Iterator Pattern

Challenge:

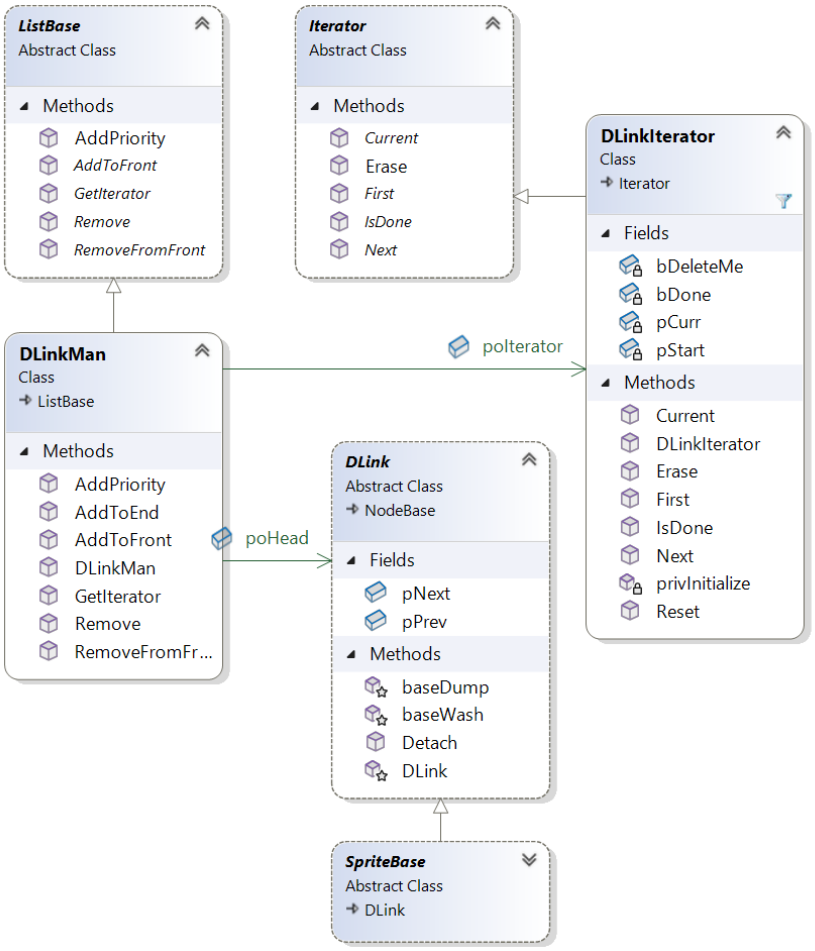
Need a simple way to access every object within all of our manager classes.

Problem:

We have many Manager classes that use different versions of the Linked List data structure, and we don't have a way to quickly and easily move through the list or access the contents of the list to find or remove specific nodes.

Solution:

The solution to this problem is to create an encapsulated way to move through and access each node within the linked list structures in a way that we can also add more functionality depending on the type of linked list we're using for that specific manager class. The Iterator pattern would allow us to create separate iterators that can quickly move through the lists provided, while also giving us the functionality to Find specific nodes and remove specific nodes from the lists. We can also add more functionality on Add for nodes with specific priority needs depending on how the manager classes are using them.



Pattern Description:

This pattern creates an iterator object that encapsulates all of the linked list functions into one easy to use object.

The mechanics to this pattern are we create a separate linked list iterator for the single linked and double linked list structures, then instead of manually moving through the list when we want to find or remove a node for instance, we simply get the iterator that is tied to the linked list already and set the iterator to process the list. The good news is that all the functions for add, find, and remove are all encapsulated which means that it is re-usable and scalable. Each iterator object is tied separately to the Dlink or Slink list objects, which means that we don't need to worry about which linked list type we're using with a specific manager, the process for moving through the list is exactly the same, which places all of the responsibility of the implementation onto the iterator object and not on the objects utilizing the iterator which reduces the class complexity of those object classes using the iterator object which is especially helpful when the classes need to do more advanced functionality like adding nodes to a list based on priority or time mechanics. The last advantage is that when we make a change to the iterator object, it is carried across all of the other instances of that object so that we don't have to make 1-n changes in our code base, it's one change that populates to all other iterator objects for that particular list, which is extremely helpful in scalability.

Key Object-Oriented Mechanics:

This pattern is implemented within the Dlink and Slink manager classes and is then used in place of a for loop, passing the first node in the list until it is completed, which allows for all of our manager classes to iterate through their respective lists without needing to share how the iterator is working which the manager class doesn't need to be functional. There is a pointer from the DLinkManager and SLinkManager classes that directly points to its respective iterator object. The iterator object itself can be returned through the getIterator() function that all managers have access to when they need to access the iterator object to move through their respective lists.

How it is used in Space Invaders:

In Space Invaders, we use this pattern to easily encapsulate all of our looping linked list behavior into a much easier to use interface that can be quickly re-used. For example, let's say we want to remove an Alien object from the grid because that object has been destroyed by a missile. We would need to iterate up from the alien node to the grid node (root node) and then we would need to iterate through the GameObjectNodeManager to find the GameObjectNode that matches the root node, then we would need to iterate back through the grid to find the alien node that was destroyed and then remove that node. In that one example of one game object interaction, we needed to iterate through 2 lists, 3 times which is made much easier using the iterator object being shared from the list manager to the object manager and that doesn't even account for the missile iteration we would have to do for the missile object as well.

Strategy Pattern

Challenge:

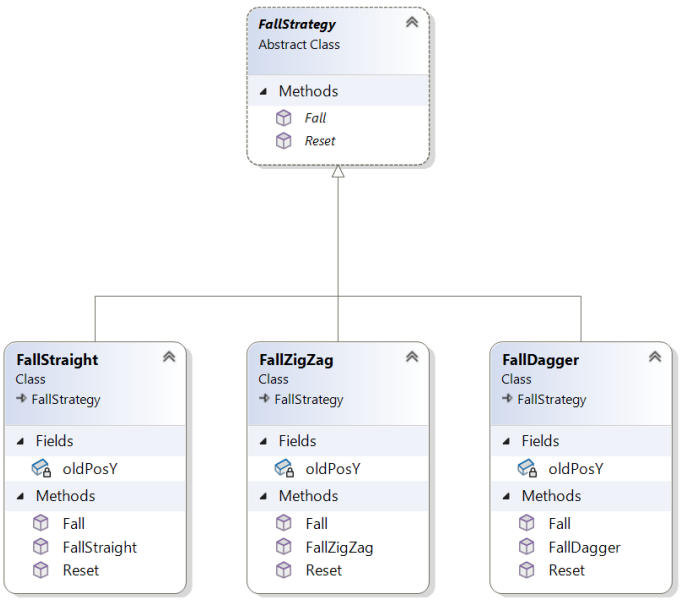
We want to impart a similar behavior to our Bomb game objects easily.

Problem:

We have several Bomb objects that we want to share similar specific behavior between but allow changes to that behavior to occur between bombs by inheriting the behavior. We also want to make this a flexible addition to the bomb classes instead of creating a subclass to deal with the behavior similarities and differences.

Solution:

We need a structure that can pass similar behaviors onto multiple objects through inheritance, without creating a separate subclass. For this, the Strategy Pattern is a perfect fit because it will allow us to abstract the methods that we need in order for the bombs to have similar behavior that can be directly changed within each bomb class. The two methods that we're going to need are the Fall method which will control all of the behavior of the bomb as it plummets towards the player/earth, and the Reset method which will reset the bomb at the correct y value of where it's dropping next;



Pattern Description:

This pattern allows us to share functionality between different game objects without creating a subclass between those objects which would increase the complexity of the objects.

The mechanics behind this pattern operate in a similar way to a state pattern but they are different in the way they are used. It allows us to define an algorithm that is encapsulated and then allows that algorithm to vary independently from the objects that are utilizing it. IN a similar way to the iterator, it encapsulates the implementation behind an easy-to-use interface that is abstracted to each of the objects deriving from it. One note for the strategy pattern is that if you use inheritance to define the behavior of the strategy, then you are stuck with that behavior which is important to note.

This pattern is also a good way to propagate an algorithm or object behavior to a lot of different classes that you want to behave similarly to each other, like the mechanics of different kinds of cars driving on a flat roadway, they should all be able to drive straight, backwards, and turn left and right because the behavior doesn't need to be defined individually for each object, but can be modified for so that how the object goes about that behavior can be different from vehicle to vehicle. One vehicle might drive straight forwards while another vehicle might approach it by zigzagging left and right by a set amount to move straight forward wh.

Key Object-Oriented Mechanics:

This pattern is implemented by creating a base strategy class that abstracts all the required behaviors that will be overridden in the classes that derive from the base class. This will allow each object to fully decide how they approach and implement the shared behaviors that are passed from the base class. As each object overrides the behavior we can see fundamentally different approaches to certain game behaviors.

How it is used in Space Invaders:

In space invaders we use this pattern to share behavior between our bomb objects as a derived strategy class. In the base FallStrategy class we define two abstract functions, one for Fall which simulates a bomb falling from the sky, and Reset which resets the bomb at the correct height to fall from. Because we pass this down to each of our bomb objects, they decide on a local level how to implement the Fall function, while maintaining the same Reset function. For instance, our FallStraight strategy will implement fall with a direct negative y direction being added to the object constantly to simulate a straight fall, while the FallZigZag strategy will implement a bit of drift from left to right that will simulate a different type of fall.

The ability to override these behaviors on an object-by-object basis, gives us full control over the behaviors that each object will possess, while still allowing them to inherit all of the behaviors from the FallStrategy class which defines the basic behaviors associated with the game object themselves, which would be a bomb in this case.

Factory Pattern

Challenge:

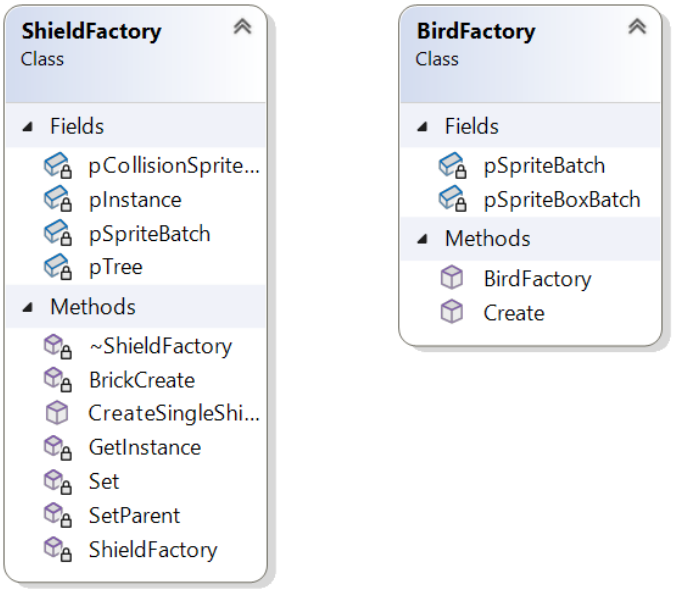
We want to create a large amount of game objects quickly and cleanly.

Problem:

We have to handle the creation of 55 Alien game objects and 4 shields worth of game objects that coincide with different construction parameters. Creating each object one at a time individually would be a monumental task that would quickly become tedious to program while trying to manage the size and shape of the over-arching objects themselves. We also must create copies of the shields in the exact same shape which creates an exponential problem when managing the location, size and shape of each individual shield.

Solution:

The solution to this problem is to create a class that will encapsulate the construction of many different game objects while making sure they are instantiated correctly and setup for use prior to them leaving the constructor. The Factory pattern is perfectly suited to alleviate this problem. The factory class can be used to encapsulate construction of the Alien grid by creating the grid, columns, and alien objects easily and we can also use a separate factory class for the shields which will alleviate all the same issues revolving around the shield placement and shape.



Pattern Description:

This pattern allows us to control the creation of many game objects and encapsulate the creation of those objects behind an easy-to-use interface.

The mechanics behind this pattern operate on the intent that we are creating the objects without exposing the instantiation logic and through a common interface. The factory mechanics also hide all the variation that is applied to each of the game objects during their creation and instantiation. This allows the client to create a wide variety of objects easily and without a lot of hassle because all of the creation logic is encapsulated within the factory object.

We should specify one factory per collection of game objects for instance if we were creating cars and roads we shouldn't couple them together, we would instantiate two separate factories, one to handle vehicle construction, and the other to handle road construction to separate all of the necessary variation that will exist between the subsets of those objects.

Key Object-Oriented Mechanics:

This pattern is implemented as a separate game object that encompasses all the instantiation and creation logic for whatever object type it is constructing. For example, having a separate shield factory which only constructs shield objects and alien factory that creates alien objects. Each factory has a Create function that allows the client to select which object is being constructed and define where on the screen it is being constructed to.

The encapsulation of the creation and instantiation logic allows us to easily construct many objects from a single function execution which is helpful in the instances where we are dealing with a lot of objects.

How it is used in Space Invaders:

In space invaders we use this pattern to create two separate factory objects that will handle the construction of our two largest collections of game objects on screen. One of our factories will handle the creation and instantiation of the alien objects, encompassing the alien game objects, the alien column objects, and the alien grid object. The other factory will encompass the shield creation, laying out the correct brick layout for a single shield object to be returned.

This ability to create large swathes of game objects in one go is great for instances such as the alien grid creation, where we create the grid object first, then in N column cycles we can populate the entire grid by creating an alien column object, and then creating an entire grid worth of alien objects during the same loop, all of which are properly instantiated and constructed and can be easily tied together in the grid hierarchy. The shield factory takes a different approach than the alien factory in the fact that we encapsulate the entire creation logic for one shield and output that shield object from the factory instead of outputting a single alien game object. We take this approach because the definition of a shield object will never change and must be the same across all instances of the shield objects.

Proxy Pattern

Challenge:

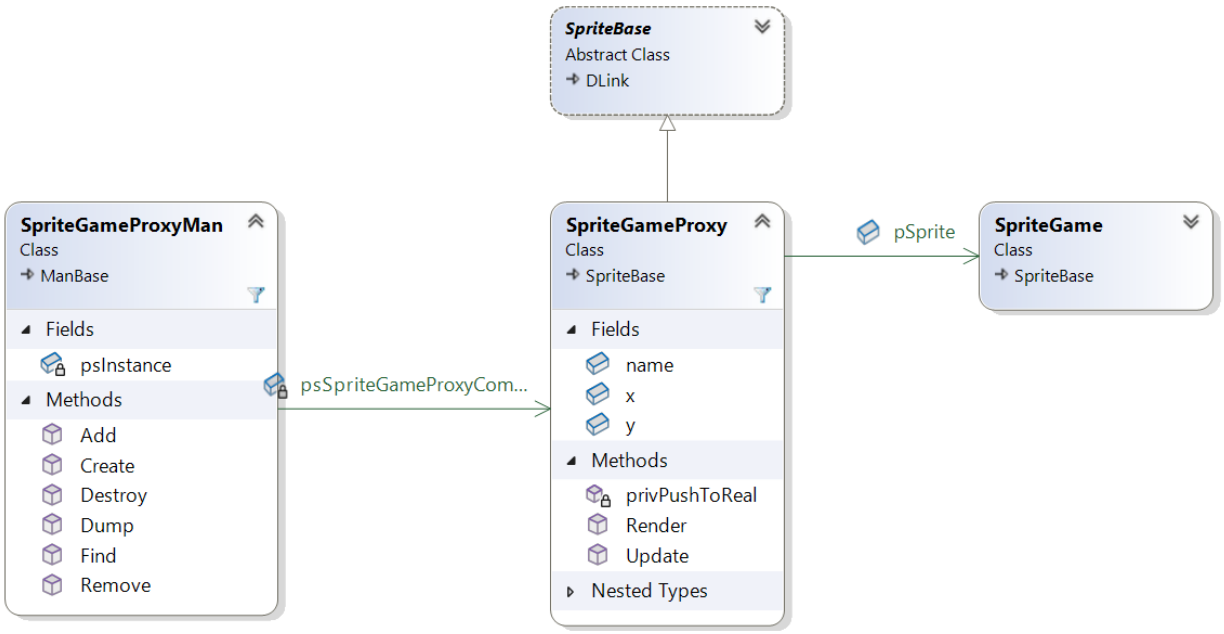
Too many heavy sprites

Problem:

We are using many interactive sprites within our game and the most numerous is the Sprite object which carries our individual Alien data. In the instance of the Alien game objects, we have 3 different classes, one for each alien type, and each of those classes encompasses the object's location x and y values, the scale of those location values, the angle of rotation, the Enumeration name of the object, the color of the object, the sprite association of the object and the collision box that is attached to the object. If we extend all that data into the 55 individual alien objects, we have a huge memory footprint during runtime that is carried over to each object. When looking closer at the Sprite objects, we only need the x and y location values from the class that are updated during movement. If we don't need to carry the entire classes data around continuously, we should look for a way to minimize the **memory footprint on these objects.**

Solution:

We should create a class that carries the x and y values from the original Sprite class and add a pointer that connects this class to the original Sprite class so that the original data can be preserved, and it can be quickly accessed from this class. We also need a way to be able to update the original class's x and y values based off any changes that occur to this class's x and y values without the huge memory allocation. The Proxy design pattern is a perfect fit for this challenge.



Pattern Description:

The Proxy Pattern allows us to create a lightweight placeholder object along with an interface that controls access to the original object's data. The interface design of this lightweight game object is the most important aspect besides the memory savings. The interface itself allows us to directly access and affect the original Sprite class as if we were using that class instead.

Through direct pointer access, we can call any of the functions that are tied to the original Sprite class as well as pushing any updates from the Proxy to the original Sprite. This allows us to still instantiate unique copies of the original sprite class which returns exponential savings on memory.

The second way that the Proxy pattern saves us memory happens during run-time. We want to access the individual alien objects, for collision detection or movement, and every time we access the object itself as a reference or when we get to the bottom of the tree, the entire object needs to be loaded into memory and the Proxy class allows us to cut that memory footprint into more exponential growth considering we are accessing these fifty-five objects every frame, and that's only one aspect of the gameplay.

Key Object-Oriented Mechanics:

This pattern is implemented as a class that shares the same base class and base class methods of the class that it's replacing. This allows the Proxy to swap easily into the place of the SpriteGame objects within our current game object hierarchies without creating ripples of refactoring that would be required if they weren't compatible switches. This also gives the Proxy class the methods it needs to Update and Render the actual SpriteGame class that it is pointing to.

This class carries a pointer which allows it to access all of the information in the much heavier SpriteGame class if an object referencing the Proxy class requires it. This pointer also gives the Proxy class the tools it needs to pass any x,y changes and Update and Rendering information to the Sprite it is replacing, allowing full functionality as if it was the original class.

How it is used in Space Invaders:

In space invaders we are using a Proxy to replace the heavier SpriteGame class and the SpriteBox class which allows us to save 110 game objects worth of memory in just the alien grid alone. We use the SpriteProxy class as a replacement for the SpriteGame class and use the pointer from SpriteProxy that points to the SpriteGame class it is associated with to push all movement of the Sprite as well as Update and Render information to the SpriteGame class. We take a similar approach with the collision boxes, using the SpriteBoxProxy class to replace the SpriteBox classes used for collision and allowing the SpriteBoxProxy class to push all updates to the real SpriteBox class that it is associated with.

State Pattern

Challenge:

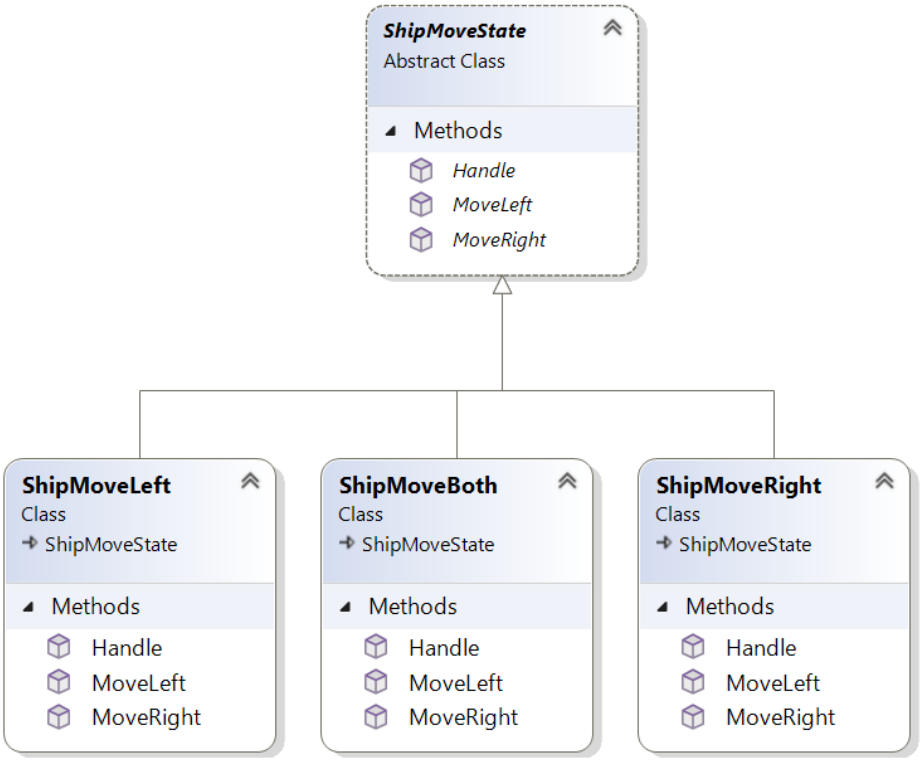
Switch between Start, Gameplay, End.

Problem:

We need to have a way to effortlessly switch between the Start splash screen, the normal gameplay loop, and the end screen without disrupting the gameplay loop. We also need a way to completely encapsulate all the associated objects and managers within those different areas, so they don't overlap and create issues between which version of the managers are being referenced by each area of the gameplay loop without creating separate classes to distinguish them. We also need a way to track if our ship has a missile ready or if it is 'reloading' and how it can move between the bumpers that are set.

Solution:

The solution is to create separate classes that can be used to carry the different objects and managers that each gameplay section will utilize during its instantiation, run-time, and deletion. The State pattern is perfectly suited for this because it allows us to set maintain an overall gameplay State manager that will both maintain the current state and transition/change the state to whatever the gameplay requires. We can also use this pattern for our ship issue which will allow us to set a missile ready or reloading state, telling the game object if it has a missile or not, and whether the ship can traverse left or right depending on whether the ship is colliding with the bumpers or not.



Pattern Description:

This pattern allows an object to alter its behavior depending on what the current internal state of the object is.

The mechanics behind this pattern replace the need for a large number of conditionals when the behavior of an object needs to change or evolve over time. This allows a single object to act as though it is a collection of multiple game objects swapping around simply by cycling through the current objects internal state at any given moment in gameplay as the need arises.

Because we tie all the associated behavior to each state, we can modify how the game object operates in several ways without having to go through the hassle of creating separate game objects and subclasses, we can simply swap the current state of the object around and get exactly the behavior we need for a given situation. We also do not delete states as we swap off of them because they are reusable and will likely define object states that are commonly associated with the objects tied to them, for example a boat would have different states than a plane.

Key Object-Oriented Mechanics:

This pattern is implemented by creating separate state objects that each have a distinct set of pre-defined behavior associated with them and then instantiating those separate state objects within the object that we want that behavior to apply to. For example, if we have a plane object, we would want a groundStop state where the plane isn't moving and it's on the ground, we would want a groundMoving state when it's taxiing on the ground and we would want a Fly state when it is in the air because the behavior for the object is inherently different between all of those states of the object.

How it is used in Space Invaders:

In space invaders, we use states to define how our player ship is behaving at any given moment during gameplay. The ship has predefined movement behavior which is reflected in the MoveBoth state, allowing the player to move both left and right freely. This behavior changes once the player encounters a bumper object, which swaps the state of the ship so that it cannot move further in the direction of the object that is it colliding with. This simulates the boundaries associated with both physical objects as well as gameplay boundaries associated with the design of the game. Instead of having to create separate ship objects, we switch the state of the singular player ship which makes the overall implementation much simpler and easier to work with.

We also use states to further define if the player ship is currently armed with a missile or if the missile is flying towards a target. When the ship is instantiated, it is set to be armed with a missile but when the missile is flying, we want the ship to be unable to fire further missiles, so we changed the behavior by swapping to the MissileFlying state which has no shooting implementation. This allows us to wait for the missile to impact either an Alien object or the wall before the player is allowed to shoot again, behavior that would be much more difficult to implement without this pattern.

Visitor Pattern

Challenge:

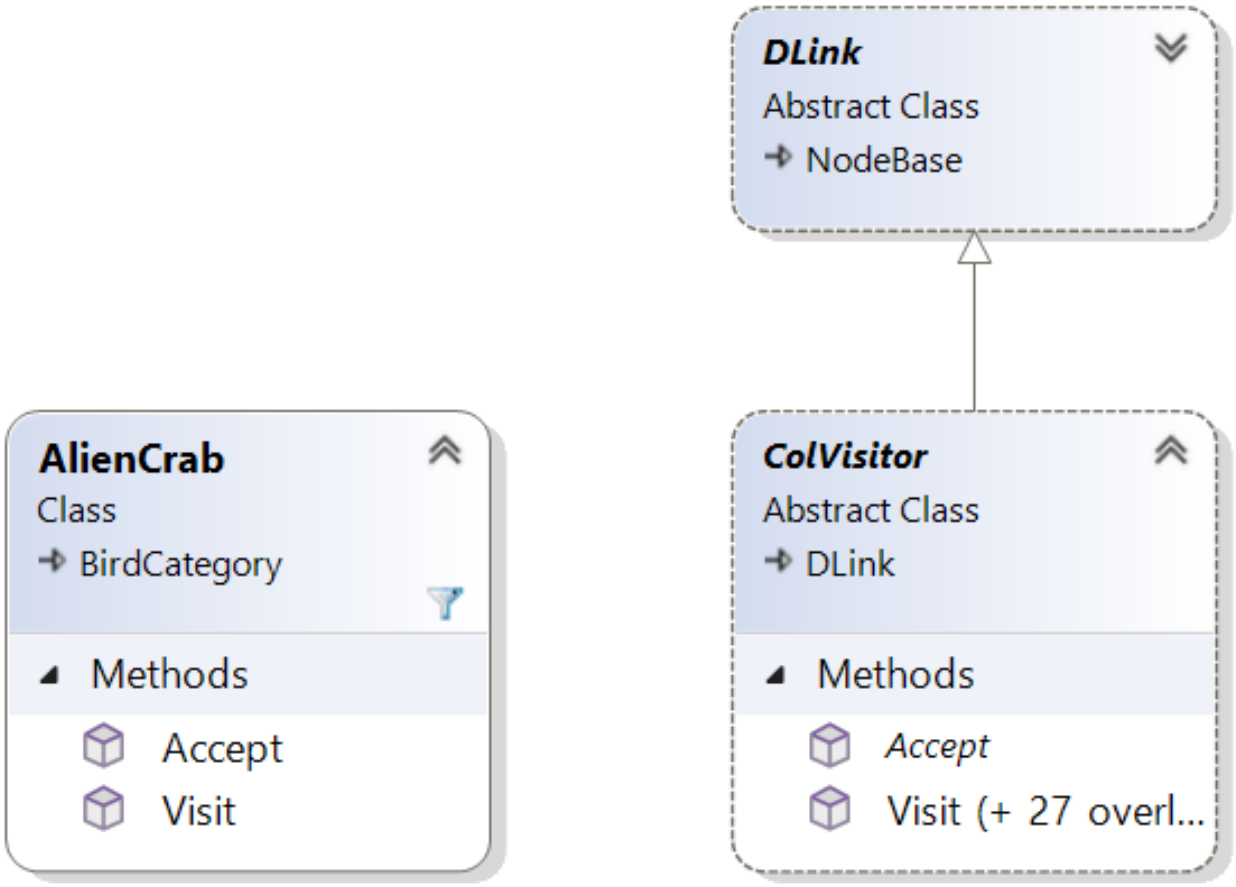
Track when collision occurs between 2 game objects.

Problem:

We need to have a system in place that can track collisions between 2 game objects as the collision occurs during gameplay. The system needs to be lightweight because we of the amount of game objects we have on screen, 55 aliens, shields, missiles, bombs, ships, that if we tried to check every collision box every frame, our game would slow down immensely.

Solution:

The solution to this problem is to create a class of objects that can track collision between specific game objects as they occur rather than on every frame of gameplay. The visitor pattern is perfectly suited for this task because it allows the game object that gets collided with to send a message out that it has been collided with a game object that has been set up to collide with it, or to ignore a 'collision' if the game object has not been setup to for a collision event. This gives us a lot of freedom when combining collision pairs of objects to decide exactly which objects should and shouldn't be sending collisions.



Pattern Description:

This pattern creates a notification system that can be passed between objects as they collide. Once an object has collided with another object, they run the Visit function and the Accept function depending on how the collision pair is set up.

The mechanics of this pattern revolve around having a base collision Visitor class which virtually implements all visit functions for the game, which are overridden in the actual class the collisions take place. Once a collision occurs, the objects that are collided can modify their behavior based on which object collides with them making this a very powerful pattern.

This interaction allows us to track when objects collide as well as setup who we want to show collision behavior, for example we wouldn't care if a wall and a floor object collide where they intersect so by pairing objects together in Visitor collision pairs, we can decide which interactions exist within our gameplay loop which allows us to define precise collision behavior between those objects. This pattern also allows us to not have to check constantly for collisions happening between all of the objects on screen every frame, because we have associated collision pair's that can be cycled through and when one is visited, they call collide and all collision behavior is applied then.

Key Object-Oriented Mechanics:

This pattern is implemented mainly through the Collision pair manager class and the collision visitor class. The collision pair manager class managers all of our collision pairs and is tested every frame to see if the collision boxes of the two root objects are colliding, if they are they fire off the respective Visit and Accept functions to modify the behavior.

Because each object has a local version of Visit and Accept, we can modify exactly which objects the base object can collide with, meaning if the collision pair fires off a collide notification message and the corresponding class has no visit or accept from the respective objects, then no collision behavior is observed beyond the collision of the boxes taking place.

How it is used in Space Invaders:

In space invaders we use the Visitor pattern to track and notify when collision happens between our active game objects. We set this up within each class, for example the alien object has a Visit from Missile and from the Wall objects because during gameplay, those are the only 2 objects that will interact with an Alien object. If the alien object were to interact with the floor wall, no collision event would occur because neither the bottom wall nor the Alien objects are associated collision pairs, nor do they have Visit/Accept functionality.

Building off of that we also have to define what behavior happens when an alien encounters a shield object and how that behavior is handled, and using the visitor system coupled with the Collision Pair Manager, this makes creating new pairs and modifying existing behavior to be quite easy.

Observer Pattern

Challenge:

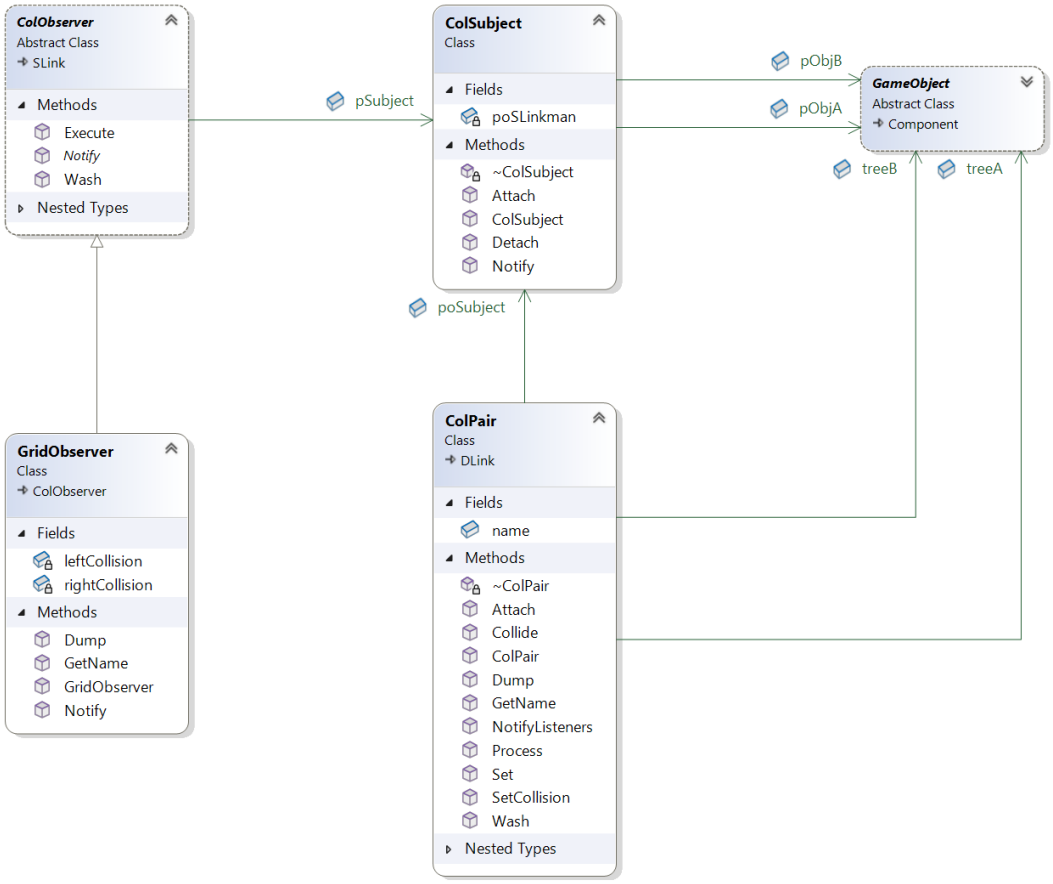
We need a way to do things when collision occurs between 2 game objects.

Problem:

We need a way to do gameplay interactions when a specific collision event occurs during gameplay. We also need a way to hold many different interactions that can be fired off when that collision occurs, while saving the interactions for future collisions between the same objects because our interaction behavior does not change or evolve through gameplay.

Solution:

The solution to this problem is to create a class that can do a set of interactions when an object collides with another object. The Observer Pattern works well in this instance because we have already created a collision messenger system with the Visitor pattern and that working in tandem with the Observer pattern will allow us to solve the issue of holding 1-many interactions within one class and firing them off when the collision is detected from a Visit between objects and a collision call. The best part of the Observer pattern is that it is only called once collision occurs, making it a passive observer instead of acting and checking every single frame.



Pattern Description:

This pattern creates an object that encapsulates behavior that we want to change or modify once the conditions for the observer occur. This pattern is almost always implemented with the visitor pattern because the visitor pattern already implements a messenger system to notify when collision occurs between two objects, and this pattern helps to add more functionality to the visitor pattern by adding object modification and behavior during the collision's that are tracked by the visitor pattern.

The mechanics behind this pattern operate through the creation of an object that will contain all the behavior that the observer will require. Then we attach the observer to the associated collision pair that the observer will become a part of, allowing it to affect the behavior of one or both the trees in the collision pair. This can be a small behavior change such as changing direction or playing a sound, or something much larger like a form change on a boss once its health goes below a certain level.

Observers are reusable, and they are continuously attached to the collision pair as long as it remains in game which makes them a good pattern for behavior that is circumstantial but might be repeated if the conditions continually come up. A good example of this would be a player character moving from land to water and the observer noticing the collision of water and player would swap the states to swimming as well as all associated behavior with the switch.

Key Object-Oriented Mechanics:

This pattern is implemented through a separate observer class that is attached to the collision pair when that collision pair is instantiated. Then when that collision pair is active, it notifies each observer for that pair one at a time, and the associated behavior is processed from the Notify function.

This system relies on the visitor pattern to announce when a collision has occurred and builds off of that message system that's already in place to provide more advanced behavior.

How it is used in Space Invaders:

In space invaders we use the observer pattern to modify game object behavior when collision occurs. This behavior modification can be a one-to-many occurrence and we build separate observers with each collision pair in mind, so missile collision pairs would share a RemoveMissileObserver but would differ depending on what object the missile hit which allows us to define object behavior independent of what the second part of the collision pair is, such as a missile exploding on contact with any object.

For example, we have a GridObserver which is called when the grid collides with either of the two boundary walls we have set-up, and this observer reverses the movement speed of the grid, while also moving the grid down 20 pixels from where it was originally. This complex behavior is easily implemented because of the observer pattern and allows us to execute that movement across the entirety of the grid only when that collision occurs.

Command Pattern

Challenge:

We need a way to make the aliens march and animate together.

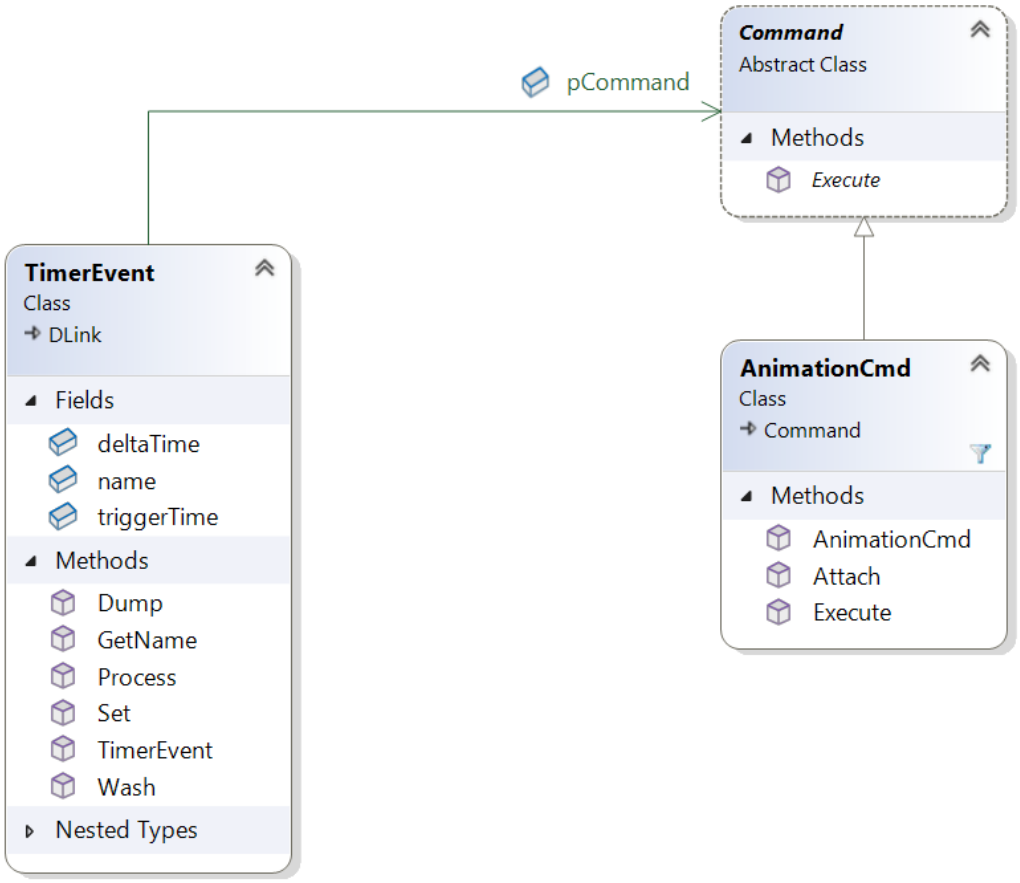
Problem:

We need an easy way to get all 55 of our aliens to animate on the same frame. We also need an easy way to move all 55 aliens on the same frame that animation occurs on. We also want a way to declare when we want these things to happen, and to track when future animations and movement occurs based on the same scalar value.

Solution:

The solution to this problem is to create a class that can be called at different time intervals, which will allow us to do certain interactions when those time intervals have passed while maintaining ease of use and modularity.

The Command pattern is perfect for this because it will allow us to create commands that are executed based on how much time has occurred in game, maintaining an instance of the Command which is reused every time the command is executed and then can be re-added with a different time interval or completely discarded if a one-time use object.



Pattern Description:

This pattern encapsulates a request into a Command object which has an execute function to process that request. When the command is fired off, the command processes the execute function which applies whatever the command has inside.

The mechanics of this pattern revolve around the modularity of the Execute function inside the command class that called it. That modularity allows the user to do anything while giving them the ability to fire that execution off once, removing it after, or keeping it and executing the command multiple times. Also, because we encapsulate the command itself, we can execute the command without passing the execution method or the command object itself.

Because of this, command pattern classes usually only encompass one behavior type per command because we want to be able to reuse the commands multiple times. This means that we will have to create separate commands that do separate things and then store those commands to be used when necessary.

Key Object-Oriented Mechanics:

This pattern is implemented in code by creating separate command classes which allow us to pass requests onto different objects. For example, if we Execute a command that fines the player $100 for committing a crime in game, we can execute that command from the Crime class without having to pass what the execute function is or what it is doing.

Because of the reusability of this pattern, it lends itself well to working with a timer or some manager that simulates time because you can set a timer on the command itself and execute when that timer has expired.

How it is used in Space Invaders:

In space invaders we are using the command pattern to execute the animation cycle of our alien sprites as well as using a separate command to move the grid in lockstep with the animation cycle. Because our commands are tied to a timer, we simply set the animation and grid movement to execute at the exact same time, simulating the aliens animating as the grid is moved across the screen.

We achieve this by implementing a TimerEvent system that tracks the system time across the entirety of the gameplay loop and fires off the commands as the setTime is less than the current time and then firing off the command. We then reattach the command to the timer event manager based on the original wait time duration added to the current time.

Another example is the alien firing command which is given to each of the columns and set off at a random interval from the bottom of the column. When the column is destroyed, we remove the command from the TimerEventManager so that it is not continuing to fire when it no longer exists. This command has more complexity but because each column has its own command, but the reusability of the command pattern makes it very easy to implement this across all of the columns instead of having to create individual instances.

Composite Pattern

Challenge:

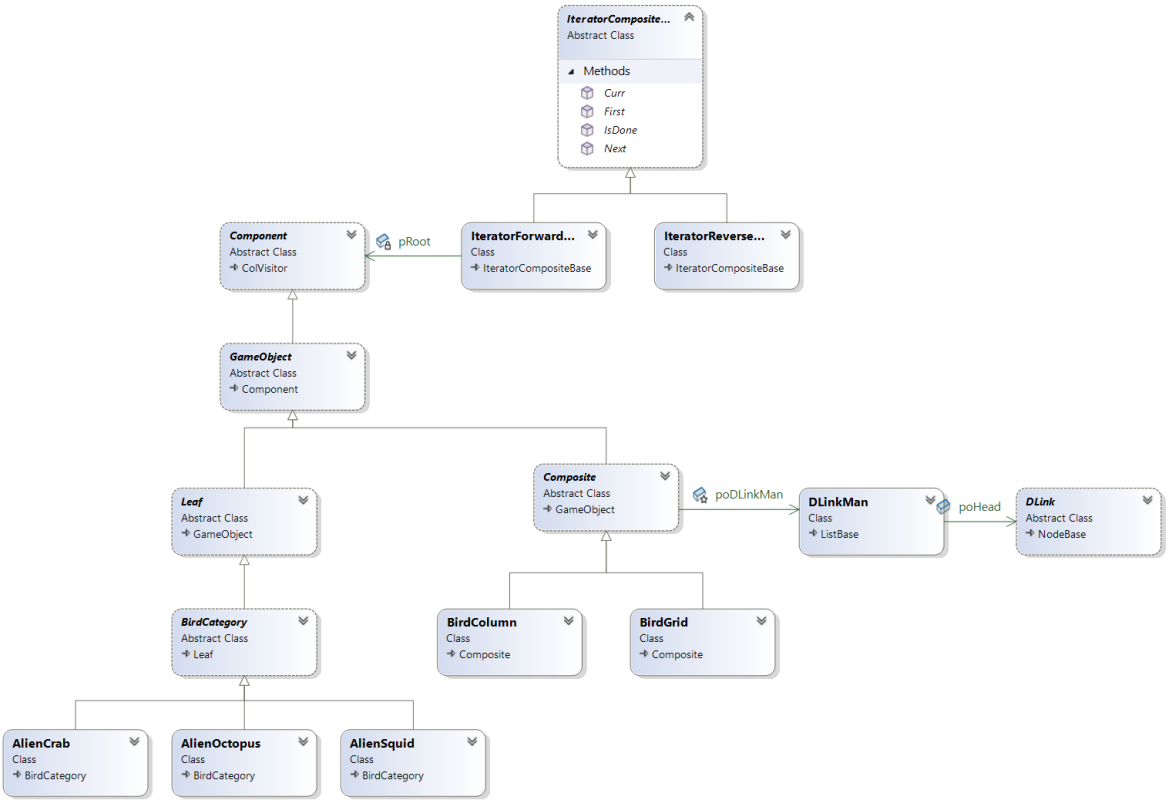
We need a way to quickly distinguish between the grid, columns, and individual aliens.

Problem:

We need a way to distinguish between if a game object is a grid, a column, or an alien object within the existing grid hierarchy. We also need a way to set up a parent/child association between the aliens and columns, and the grid and columns because it will allow us to build a hierarchy of objects that we can use to access different levels of the grid object at will and with ease.

Solution:

The solution to this problem is to create a way to distinguish between the location of a particular game object as it sits within the hierarchy of a gameplay object. The Composite pattern is perfect for this because it allows us to quickly distinguish between whether a node is a Component, which means that is has at least one object below it in the hierarchy as a child object, or as a leaf, which means it does not have an object below it in the hierarchy. The composite pattern allows us to create and use a tree structure which shares the object hierarchy that we are looking to implement between the alien grid, alien column, and alien objects.



Pattern Description:

The composite pattern creates a recursive tree structure by distinguishing an object as either a Composite or a Leaf. A composite means that the object has at least one child in the current hierarchy and a leaf designation means that the object has no children in the current hierarchy.

The mechanics for this pattern are simple but difficult to understand, basically when creating an object hierarchy, you designate if the current object is a composite, which means that the object has or will have a child component or is a leaf and we can use this hierarchy to iterate to the bottom of the list. The iteration happens as recursion because every composite has an object under it which allows you to then check if that object is a leaf or a composite and recurse down through the hierarchy until you hit a leaf node.

Key Object-Oriented Mechanics:

This pattern is implemented by creating a Composite and Leaf class, which are both derived from the base Game Object class. Because these are both derived from the base Game Object class, we can further derive all of our game objects into one of these two categories which allows us to define an object hierarchy for all objects of a certain type such as tree branches being constructed as a composite object and tree leaves being constructed as a Leaf object even if they're not tied to a specific hierarchy upon construction.

This also affects how the iterator works on these objects because they are a fundamentally different structure than our linked list data structure is so they require their own iterator.

How it is used in Space Invaders:

In space Invaders we are using the Composite pattern to define our GameObjectNode hierarchies between game objects. For example, our walls in game are constructed using a wall-group which is a composite, into a wall column, which is a composite, into a singular wall, which is a leaf node. To create this functionality, we created a separate iterator that effectively accesses the tree structure and moves through every sibling that the leaf nodes have.

We have expanded the initial functionality of our Composite pattern to allow us to GetChild from a Composite node, returning the first child node in the sequence, the GetParent from the Leaf node when we want to move back up the hierarchy, and the GetSibling function to get the other children of the parent node without having to go up a level and back down, this sequentially moves through the children. Because the iterator also carries the current node, we implemented the GetChild, GetParent, and GetSibling logic here.

We are using this to great effect in the RemoveAlienObserver hierarchy where we are deleting the alien objects. We do a series of 'if' statements to check if the current level in the heriarchy has 0 children, and if it does, we remove that object from the hierarchy, but if not, it stays. This allows the grid to clean itself up as aliens are removed, the columns are also removed. This self-cleaning ability would not be possible without the implementation of the Composite pattern.

Object Pools Pattern

Challenge:

We want to instantiate new game objects as our gameplay loop progresses, but we want to limit the number of new calls we're doing.

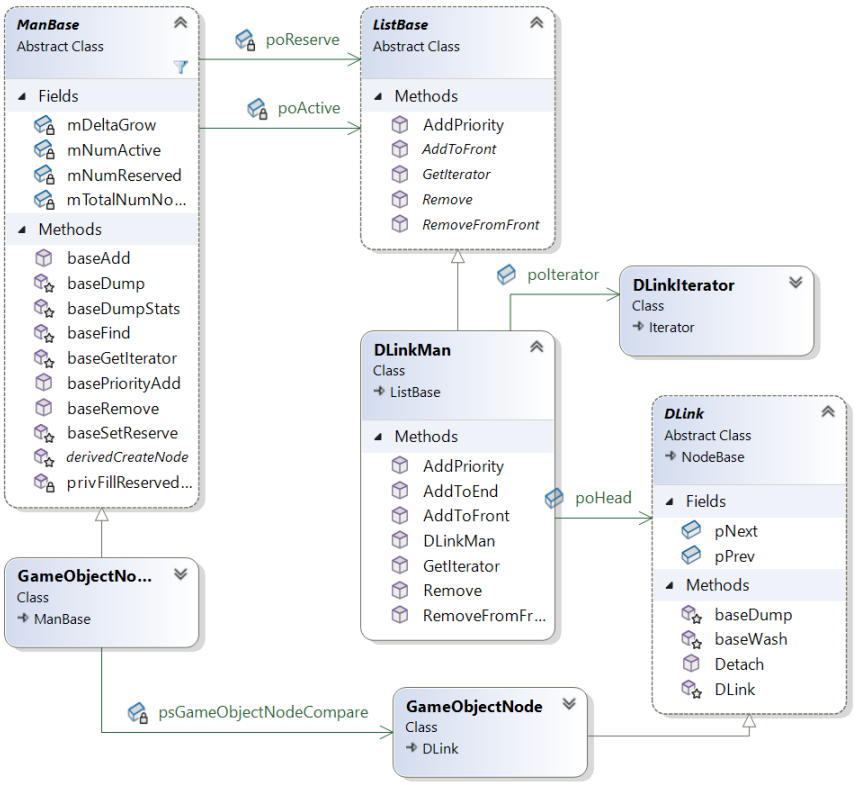
Problem:

During the gameplay loop of a typical round of Space Invaders, we are utilizing a broad number of game objects that are created, deleted, and then re-created many times before the game loop ends and while we could call new on every new instantiated object, it becomes very expensive memory-wise when we're creating a new object every time we instantiate it.

Solution:

The solution to this problem is to fully utilize the linked-list data structure that we're using and reuse the nodes once an object has been removed from gameplay, completely nullifying the new when another object takes its place.

The Object Pools pattern allows us to use separate linked list structures, which can be single linked or double linked, and use one linked list for the active list of objects, and another linked list for the reserve list of available nodes. This pool of available nodes means that we can re-link objects to nodes without having to instantiate new nodes every single time we want to add an object to the scene.



Pattern Description:

This pattern relies on using a reserve list of available nodes/objects that can be easily utilized when a new object is needed.

The mechanics behind this pattern lend themselves to the linked list structures that we're using because when we delete an active node that contains a game object, instead of deleting the node entirely and losing that allocated memory, we simply remove all the data from the node and re-attach it to the head of the reserve list while maintaining O1 time complexity. Then, when we need to add a new game object to the active list, we just pull the first reserve node, add all the data to the node from the new object, and attach it to the active list which is all encapsulated within the manager class that these objects are pooling into while maintaining the same time complexity. This pattern is extremely useful because it only calls new and grows as more nodes are needed instead of instantiating every single object in the game.

Key Object-Oriented Mechanics:

This pattern is implemented within each of the Manager classes within the game. When each Manager class is created, we instantiate 2 single or double linked list structures that are tied to the manager class, and then we utilize one of those as an active list structure and the other as a reserve list structure.

Encapsulating these lists within the managers allows them to quickly and efficiently move nodes between lists, add nodes to the active and reserve lists when needed, and remove nodes from both lists. The best part about this is that we are using the normal functions of the linked list structure in order to accomplish this pattern, but instead of deleting nodes upon removal from either the active or reserve list, we are simply hanging onto them.

Another great inclusion is that we can control how many nodes the reserve list starts with during instantiation, and how fast each of the reserve lists grow independently of each other. This is especially helpful when dealing with certain lists that either don't need to grow very fast and can start a little smaller, and those that need to start much bigger or need to grow very fast.

How it is used in Space Invaders:

In Space Invaders we use this pattern on all our managers in our Manager Base class that all our manager classes derive from. In our Game Object Node Manager for example, it allows us to add new object roots to our game object node manager without instantiating new nodes every time, as we add new objects to the active list. Once a game object root has been removed from the game entirely, we can remove the node from the active list, wash it to remove all the data, and re-add it to the reserve list. In some of our lists that don't remove a lot, or any nodes, we set the original reserve list number of nodes to be higher with a lower growth number Considering all the managers that we have within the game, which are constantly cycling their assets between active and reserve lists, the number of new calls that we're saving gives us a memory savings that is exponential in return.