WeiJian Xu
SE 456

Space Invader Project Design Document

Link To Presentation Video: https://youtu.be/sMVf7rOElbQ

Project Overview

This is the design documentation for the space invader project. The goal of this project is to implement the classic space invader game using modern design pattern concepts with C#. However, there is a set of restrictions that need to be applied. Specifically, no advance features of C# is allowed. Including the usage of generic C# array/containers. Hence, the project start-off with the implementation of its own container type, doubly linked list.

Overall Design Concept

The game engine is provided, which is called Azul. This engine provides objects that handles graphic rendering. It provides very simple classes consist of texture, image, rectangle, sprites, and box sprite objects. These objects already provide methods for rendering, so the goal is using these objects to create the game. Below is a summary of the overall design concepts being used. The details of each design pattern used will be provided in later sections.
Because C# array and containers are not allowed, the first design approach is implementing our own containers, which is doubly linked list consist of data nodes that points to previous node and the next node. Next, a generic manager is created to provide object pooling by keep tracking an active and reserve list. This way, it introduces the idea of size of a doubly linked list is introduced in this structure.

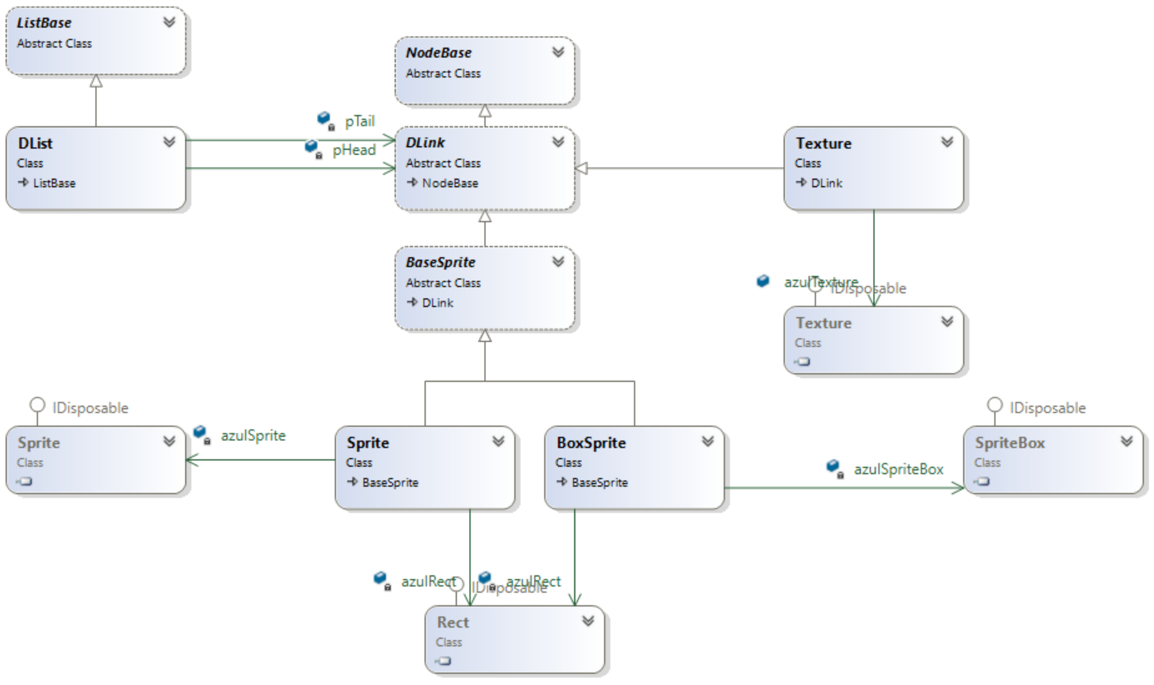
Next, the provide object types such as sprites need to fit into our design of doubly linked list, so class adapters are created to make these objects able to be added to our doubly linked list. Once these objects are incorporated, the generic manager can be extended to manage all these objects by creation of their respective managers.

Next, it is difficult to interact with many different types of objects at once, so the composite pattern is added to make everything into a tree hierarchy consist of composite and leaf, with composite can consist of many leaf nodes. This essential turn everything into a game object, and the generic manager can be extended to manage these game object nodes that could consist of various types of game object. This becomes useful since instead of dealing with sprite, sprite can wrap into a game object such as aliens, shields, etc.

Next, for things to interact with each other, the idea of collision needs to be added, so the visitor pattern is introduced to supply the collision operation for various game objects. A game object consists of the sprite and a collision box (box sprite), and a collision is defined as when two game object’s collision boxes collided with one another. With this setup, various types of game objects can be created to setup the game. To update these objects when collision occurs, collision observers are created and listen for collision events, and handle the update accordingly. The details of the design patterns that are used would be discuss in more detail below.

Adaptor

Class Diagram



General

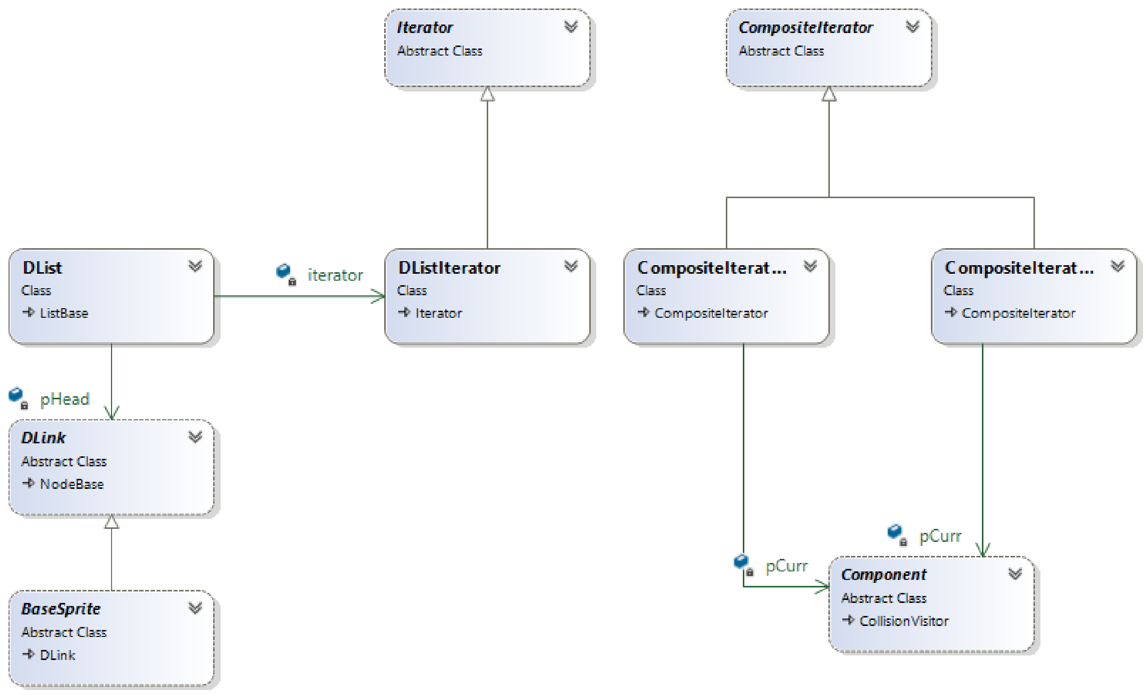
This is the class diagram that show case the idea adaptor for our game design. The idea of the adapter pattern is making object able to incorporate into an existing design. This is especially useful when objects are provided and cannot be modify.

Discussion

The Azul engine provides object such as sprites for rendering. However, these objects can’t be modify and need to be able to use within our double linked list data type, so adapter classes need to be created to wrap these object, so now we would just need to interact with the Azul object within these wrapper classes, and these wrapper classes can be modified and updated base on our needs, and this can be done without having the need to modifying the base Azul classes. Any action that needs to be done by the Azul classes would get delegated down from the wrapper class.

<u>Iterator Pattern</u>

Class Diagram



General

This is the iterator design pattern. The goal of this design pattern is when there a collection of objects being link together, there needs a way to traverse through such collection of items in a meaningful way. Having the iterator pattern allows the algorithm need to traverse such objects being wrap inside a class called the iterator. This way, the structure of how these object collections can be wrap inside the iterator class, and it becomes much easier to use.
For example, with linked list, instead of having to understand how the nodes being linked together and traverse thru each one that way, simply just call the linked list iterator class and ask for the next node. This way, the objects does not have to implement specific methods for traversal, which decouples the algorithm of the traversal to the iterator that could potentially be reused.

Discussion

Now that all provided classes are adapted into our double linked list, we need an easy way to walk thru the list. However, doubly linked list consists of nodes, which holds previous and next pointers. In order traverse through a doubly linked list, the head node needs to be always maintained, and this becomes difficult as the project gets bigger. Because of this, the iterator design pattern became very useful since the work of iterating thru the doubly linked list can package into a single iterator class. This way, with the doubly linked list contains the iterator pointer, it makes traversal thru nodes simple since all the logic of that is package inside the iterator class and can be reused multiple times.

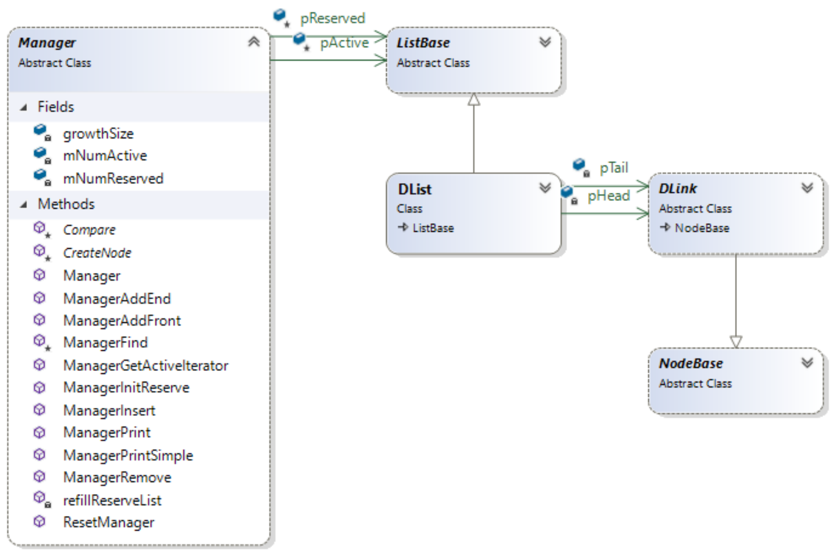
The non-iterator pattern approach of this would be constantly keeping track of head node pointer, and upon each time of traversing, the same iterating code must be re-created upon every traversal, and that is lot of work. This becomes even more useful when the collection of objects become much more complex, such as when there is the composite pattern (see Composite Pattern section), which consist of doubly linked list of trees, and re-implementing tree structure traversal every time when needed is lots of unnecessary work.

The trade-off of the pattern is that the iterator class must be implemented correctly, it has no room for mistakes since the iterator class would get re-use all over the place, and if there are any issues within the iterator class, then everything else that use it would end up breaking, so this pattern must be implemented correctly before being used. Additionally, it makes debugging code quite difficult since everything is wrap inside iterator, and it could take a while to realize it is an issue cause by the iterator.

The primary usage of the iterator pattern is to allow the game to be able to update objects in a timely manner and capable to update the correct objects. By using the iterator class, distinct objects with identification such as a specific enum name can easily be found by iterating thru the structure. This is especially usage when interacting with the Composite design pattern, where everything essentially become game objects consist composite and leaf nodes within the tree and traversing each node and updating those objects become essential for this game to function properly. Using the iterator, many of the traversing algorithm can be wrap inside the iterator class and primarily focus on the end goal instead of having to deal with node traversal.

<u>Object Pool Pattern</u>

Class Diagram



General

This is the class diagram of object pool being used inside the base manager class. The idea of object pool is there is a preloaded set of objects known as the reserve pool, and when an object is needed, instead of creating the object, it would pull from the reserve pool, and that gets added into the active pool. Once it is done being used, it would get remove from the active pool and push back to the reserve pool. The benefit of this approach is to limit the number of objects needed to be created, which would reduce the number of resources needed to be used to maintain. This also prevents unused objects since once it gets removed, it can reuse again since it is being put back into the reserve pool. By having this, there never going to end having too many objects exits and not being used, which prevent wasting resources.

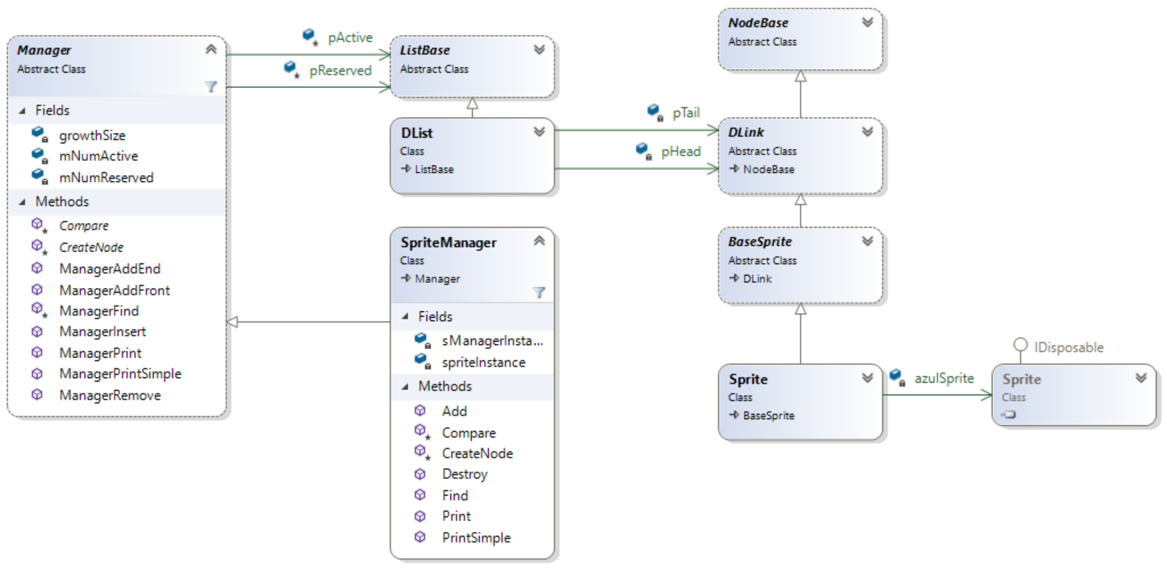
Discussion

In our game design, the doubly linked list container, DList, does not consist the idea of size. In order to traverse thru each node, there is an iterator that can be use, but there is no way to know how many nodes it has within the list until it gets to a node with its next node being null. Hence, the object pool pattern is introduced to keep track of the nodes. Inside the manager, it contains counters for the active and reserve list with a counter on number of nodes for each list. It also contains a growth size, indicating how many nodes to fill the reserve pool when more nodes are needed. Reserve list initialized with a fixed amount of node; the active list starts off empty. When a node needs to be used, it would be pull off from the reserve list and added to the active list and return the node. This way, nodes are being managed within the manager and the size of

active and reserve lists would be maintained inside the manager class. It solves the problem not knowing how many nodes there are, and nodes would never become wasted.
The non-design pattern approach to this is have this sit within the list class, and just interact with node creation on the fly. This becomes annoying when their multiple types of nodes. For example, within our game design, nodes are essentially DLink objects, which can be Image, Sprite, texture, etc. Keeping track of all these nodes is extra work that can be avoided by having this base manager class to mange them properly.

This base object pool implementation is extended for many of node manager classes, including ImageManager, SpriteManager, etc. It becomes a very useful base class because all the functionality of add/remove nodes would get delegated to this base manager class, which reduce the need of creating extra code that essentially does the same thing. The drawback to this would be when lots of nodes are needed, the manager would have to constantly refill the reserve size many times if the defined growth size is not large enough, which could have an impact to runtime performance.

<u>Template Pattern</u>



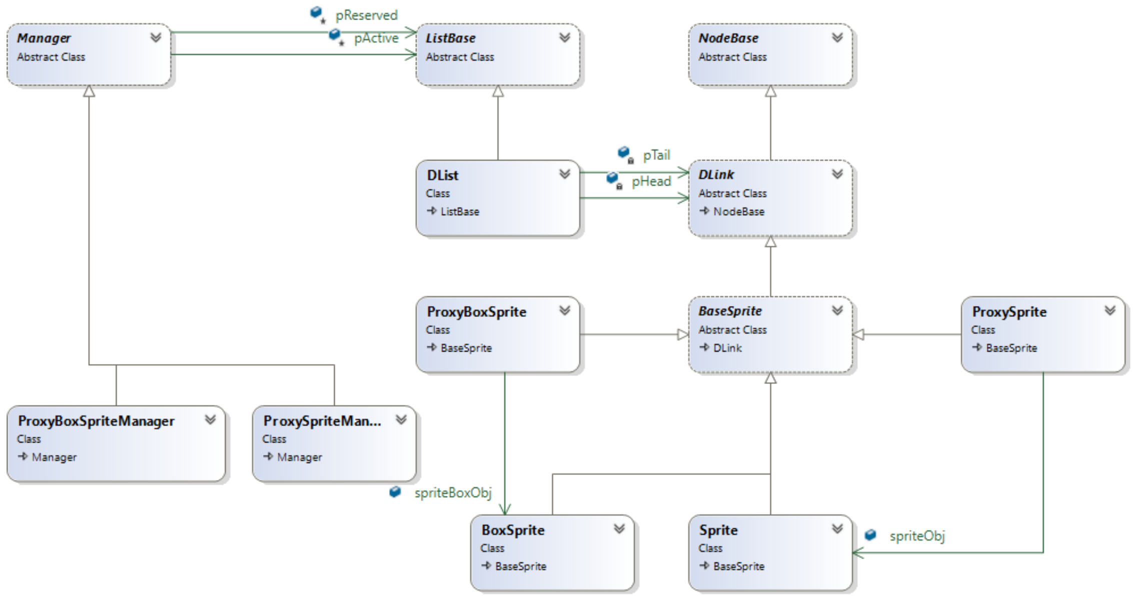
General

This is the template pattern class diagram. The template pattern provides general structure of the methods needed, and the base class can provide those functions to its sub-class by sharing it or provide it with a generic function signature and force its child class to implement such feature. This way, the base template class would always be relevant, and code that already created would no longer need to be recreated again. Additionally, it would force a common interface for objects that uses similar interface.

Discussion

With our game design, since the manager base class have already implemented object pool pattern, this can be served as a template other manager classes such as sprite manager. The class diagram above show that there are lots of similar functions that are either shared or being delegate to the sub class for implementation. This saves lots of work for each respective managers in our design since each manager no longer have to recreate the object pool pattern, it could simply just take it from the base manager class and able to reuse most of the add/remove DList operation. However, it would still require implementing its own methods of node creation since it is dealing with different types of nodes.
The non-design pattern for this would be recreate everything within each individual class, which is unnecessary and results into way too much code duplication.

Proxy Pattern



General

This is the class diagram for the proxy design pattern. The idea of the proxy design pattern is to be able to reuse same reference of a real object when replication is needed instead of constantly changing the real object, especially when the real object is complex and/or does not need to be changed. The proxy pattern would create an object that have the same interface as the real object, and every time when the real object needs to use by client, a proxy of that real object would be sent, and the actions that need to be done would have get push from the proxy to the real object. This way, if the same real object needs to be pass around, there only needs one instance of that object.

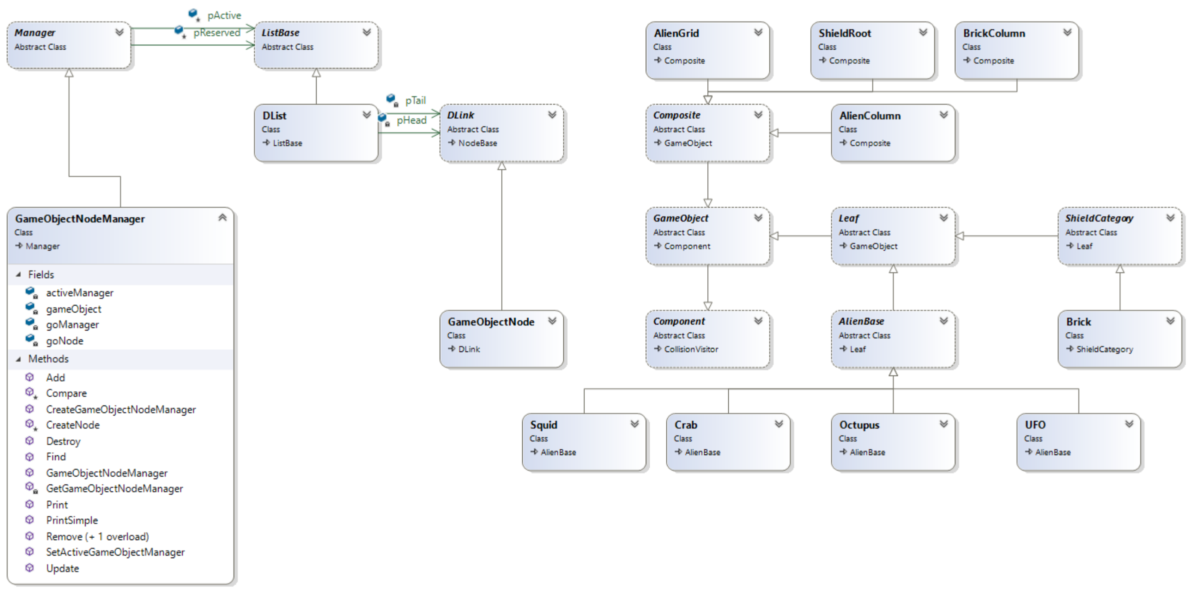
Discussion

In our game design, the alien grid consists multiple crabs, octopus, and squids. Instead having multiple sprite objects to draw all of these, it is possible to just have of each, and use proxy objects to redraw them as many times as needed. This way, it reduces the amount of actual sprite objects that need to be created, and it make deletion simpler since deleting the proxy objects would have no impact to the real objects. Also, modification to proxy sprites would not really alter the real sprite object, which reduce complexity to the sprites when is not needed.
The non-design pattern approach to this would be not using the proxy, then every single sprite would be real sprites. This would add unnecessary complexity to the sprite object as the project gets bigger, and which sprite to delete becomes hard to keep track of since every sprite is uniquely created.

This design becomes very useful when collision need to be added because proxy objects would be ones dealing with collision while the sprite objects are just responsible for rendering on to the screen, which separate the responsibility. Additionally, deletion of proxy sprites has no impact toward the sprite objects, and this can be freely with no impact to the sprites. This means that many of the same sprites can be drawn without introducing duplicate sprite objects, which reduce the amount of resource needed for the game.

<u>Composite Pattern</u>

Class Diagram



General

This is the class diagram of the composite design pattern. The goal of this pattern is to transform objects into uniform groups. By doing this, object hierarchies would turn into tree structure with leaf nodes and composite node, where composite nodes could contain other composite nodes and leaf nodes. Additionally, the client that use these composite and leaf nodes could treat them as the same type of objects, which in this case, everything essentially becomes game object type. Doing so, clients can treat object compositions just like normal object, which reduce the level of complexity when creating different game components.

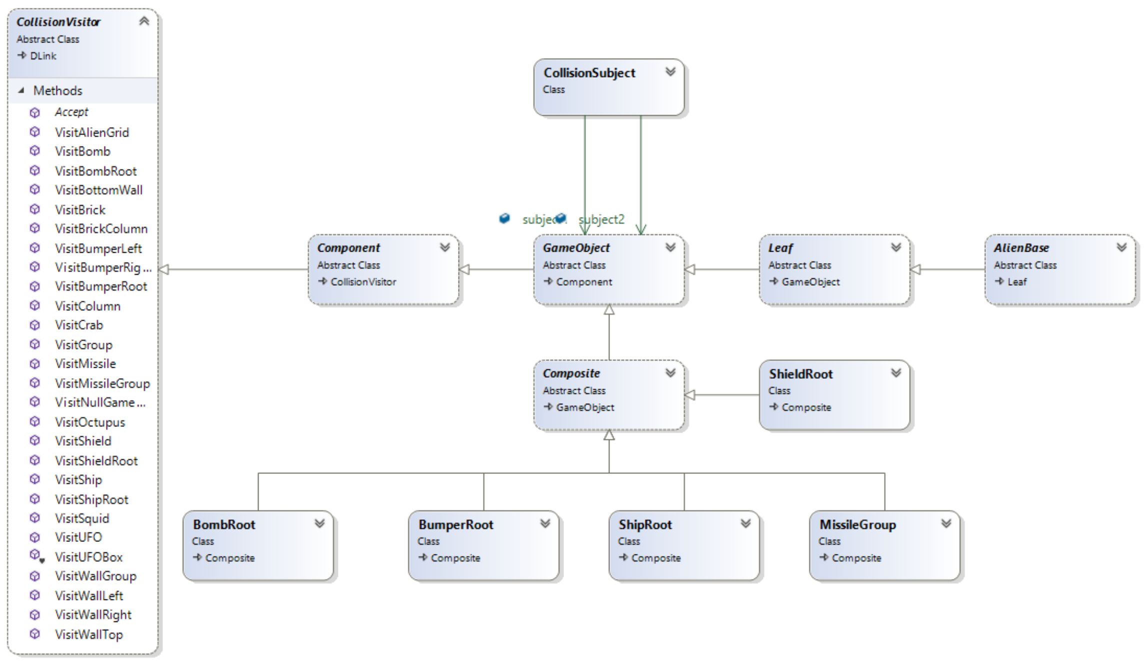
Discussion

In our game design, there consist of many different types of object that needed to be create, such as aliens, organized into a grid. Shields consist of column of bricks. All of these are different object types and implementing them individually make it difficult to interact with them in the long run since it would require type checking constantly. By having the composite pattern, the object that needs to have other component would be a composite, the objects that does not need other components would be leaf type, and which essentially is a game object. Hence everything can be treated as such. This way, construction things such as an alien column becomes easy since it would be classified as the composite type, which can contain leaf nodes of various alien types. Having implemented this design pattern, the creation of game objects such as alien grid, alien columns and alien types easy since it already incorporates the original design with the doubly linked list structure.

The non-design pattern approach would be implemented these objects individually without having a uniform type, and handling interaction between them becomes difficult since each object type is a separate case to handle. The benefit of this approach makes the implementation of collision possible, since with such design, collision would always occur between two different game objects, which would consist of checking the hierarchy of tree structure of nodes. Additionally, with all the object essentially becomes game objects, then we just have a game object node to wrap around the game objects and have game object nodes manager to manage these game object nodes, which essentially allows us to re-use our previous design of object pooling. With this extension, regarding of how many different types of game objects we need to create, it would always utilize the object pooling pattern within the game object node manager. Additionally, this becomes very useful with collision implementation using the visitor design pattern, which will be discussed in the next section.

<u>Visitor Pattern</u>

Class Diagram



General

This is the class diagram for visitor design pattern. The goal of the visitor pattern is to implement an operation for a set of uniform objects without modifying them. The goal is having a visitor that contains visiting methods, and each object that need the operation added would have an accept method and visited by methods for each case of the operation. This way, it does not require modification of the objects itself to have the operation added. This way, the overall structure of the objects would not need to be changed, only methods need to be added to have the operation implemented.

Discussion

In our game design, there are various types of game objects, such as aliens, walls, missiles, etc. An operation called collision need to be added to all these objects. Hence, the visitor pattern is introduced to solve. As shown in the class diagram, collision subject consists of two game objects. Game objects are essentially components, which consist as collision visitors. To implement collision, each of the objects that would collide would have the accept method added, as well as the visiting specific object implemented. This way, when collision is detected, the visitor would go visit the object it collided with by accepting the visit using the accept function. The handle of the collision would be done using the individual visited by functions.

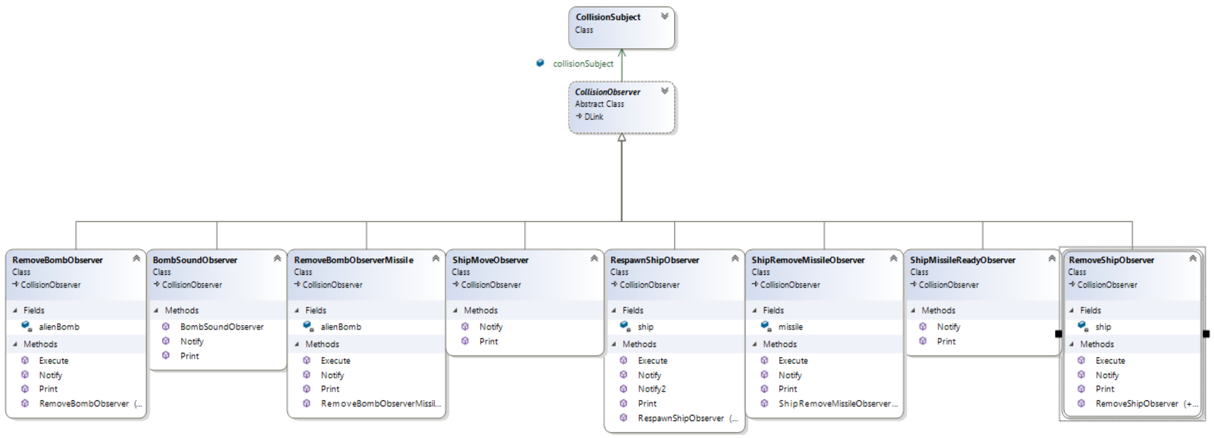
The non-design pattern approach to solve the problem of adding operation to a uniform set of objects is very difficult because each object would need to have separate functions to handle various cases of collision depending on object type. It would be very difficult to implement and hard to update and scale.

With the composite design pattern, this makes implementation of visitor very easy since everything can be defined between game objects, which is known is a tree hierarchy. Because of this, checking collision becomes checking whether each tree node has collided with one another by going down the tree structure, which is checking collision between two trees.

Additionally, the visitor defines collision between objects, and when collision occurs, objects need to be updated accordingly based on what the collision event is, and to update objects accurately and consistently, there is a need for listener to listens for collision events, and when it happens, it would need to use the appropriate handler to deal with it. This would require the design pattern called observer, which will be discussed in the next section.

<u>Observer Pattern</u>

Class Diagram



General

The is the class diagram for the observer pattern. The goal of an observer is to have flexibility of updating one to many dependencies. For example, one object's state could impact the states of many objects, and all the other objects need to know when the object that they depend on change state to update to reflect that change to stay consistent. An observer allows the dependent objects to listen for state change events and get notified accordingly to respond to such state change.

Having an observer means that it would provide notifications to objects that need to know the state change.

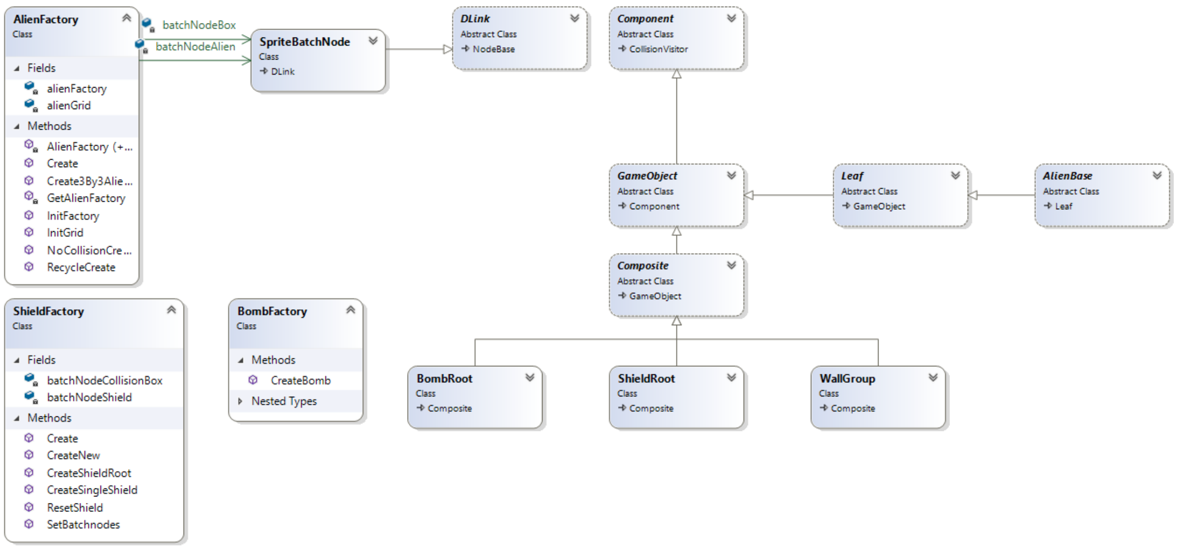
Discussion

In our game design, when collision occurs, certain objects need to get notified correctly for event to occur. For example, in our game, when bombs fall and collide with a shield object, we want an explosion sound to play, which means that we need an observer to constantly listen for collision between bombs and shields, and when they collide, it would get notified and trigger the play sound action. Without the observer, the play sound does not get notification and result the sound play event does not occur.

The non-design pattern approach to this would have the colliding objects deal with the event, but this would require every object have such feature added when it occurs. As an example, if we need the sound to play during every collision occurs, then every single collision visit would be required to have the play sound functionality, which is unnecessary. By having the observer, when game objects collided, it would just need to notify the observers knowing that those are listening for collision and can trigger the appreciate events that need to occur when it happens, such as removal of game objects and play sound, and the play sound functionality can just be implemented once and get invoked by the observer classes. It essentially allows for objects to have ways to communicate to other classes that might not be related and notify its state change to ensure consistent updates.

Factory Pattern

Class Diagram



General

This is the factory design pattern. The factory design pattern provides a single interface to create similar types of objects without having to know much about how to create those objects. It makes object creation easier since it does not require much understand of the object that it will be creating in the first place; it would simply use the factory provided methods to do so. This way, lots of the object creation logic can be wrap inside the factory does not need to be handle externally.

The goal of this pattern is simplifying the complexity of how objects need to be created so it can be used easily within the application. It reduce the need to understand the object properties before object creation

Discussion

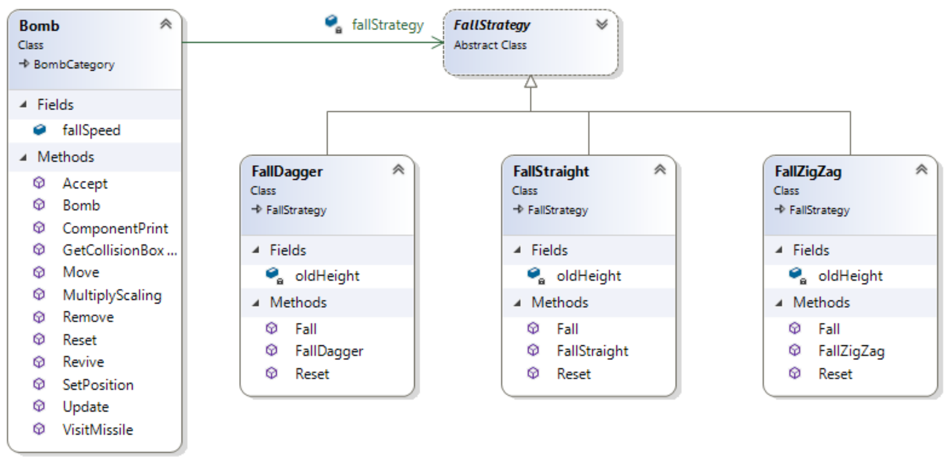
With the introduction of the composite design pattern that allows creations of various game object types. It creates the problem of there are many different components need to be created for the game to function. Mostly, the objects fall into three categories, aliens, shields, and bombs. There are 4 different types of aliens, 3 different types of bombs, and 9 different parts of a shield. Also, these objects need to be created into a grid structure, so the factory is provided to create these effectively. This way, the creation logic only needs to implement once within the factory, and then it could use easily throughout the application. For example, alien grid consists of numerous columns, and numerous numbers of aliens. Assemble such grid can be time consuming to do. Hence, such implementation can be package inside the alien factory, and it would handle all of work such as activating the sprite and collision sprites into the correct batch nodes instead of dealing with that within the game code. This reduces the complexity of the game code logic and make things separated, which make it easier to debug.

This is different than non-design pattern approach. Without the factory pattern, any object creation would require going through all requirement of what the object needs before creation, which is lots of redundant code that need to be added when is not necessary to do so. The user would need understanding how each object work before able to create it properly.

Additionally, this ties nicely with the concept of recycling objects to preserve resources. With creation of objects comes with destruction of objects, and having the factory allows the flexibility of re-use by interacting with a ghost manager, where the factory can always look inside the ghost manager to see if any object can be recycled before creating new ones. If found within the ghost manager, then the factory can simply take that out from ghost manager and update it with the correct info instead of creating a new one.
The drawback of this pattern is the factory class can become large when the creation logic becomes more complex, and as more objects need to be created, it would require additional modification to the common factory creation interface. Also, as more objects get introduce, more factory classes need to be created, which could create additional classes to manage in the long run.

Strategy Pattern

Class Diagram



<u>**General**</u>

This is the strategy class diagram. The goal of the strategy pattern is to allow an object to have the flexibility to decide how to perform the same action in many ways. It is going to achieve this without modifying the structure of the class with the introduction of a strategy type.

The primary action of the class would delegate its action method to the strategy type, and the strategy type would be the class to implement the class action method. This way, the original class can be created and pick which ever strategy to use without modifying its structure. Also, the class can be created first, and select its strategy upon run time, which makes the design very flexible. Because the class consist of a reference of the strategy type, adding new strategy is very flexible and have no impact to the original class.

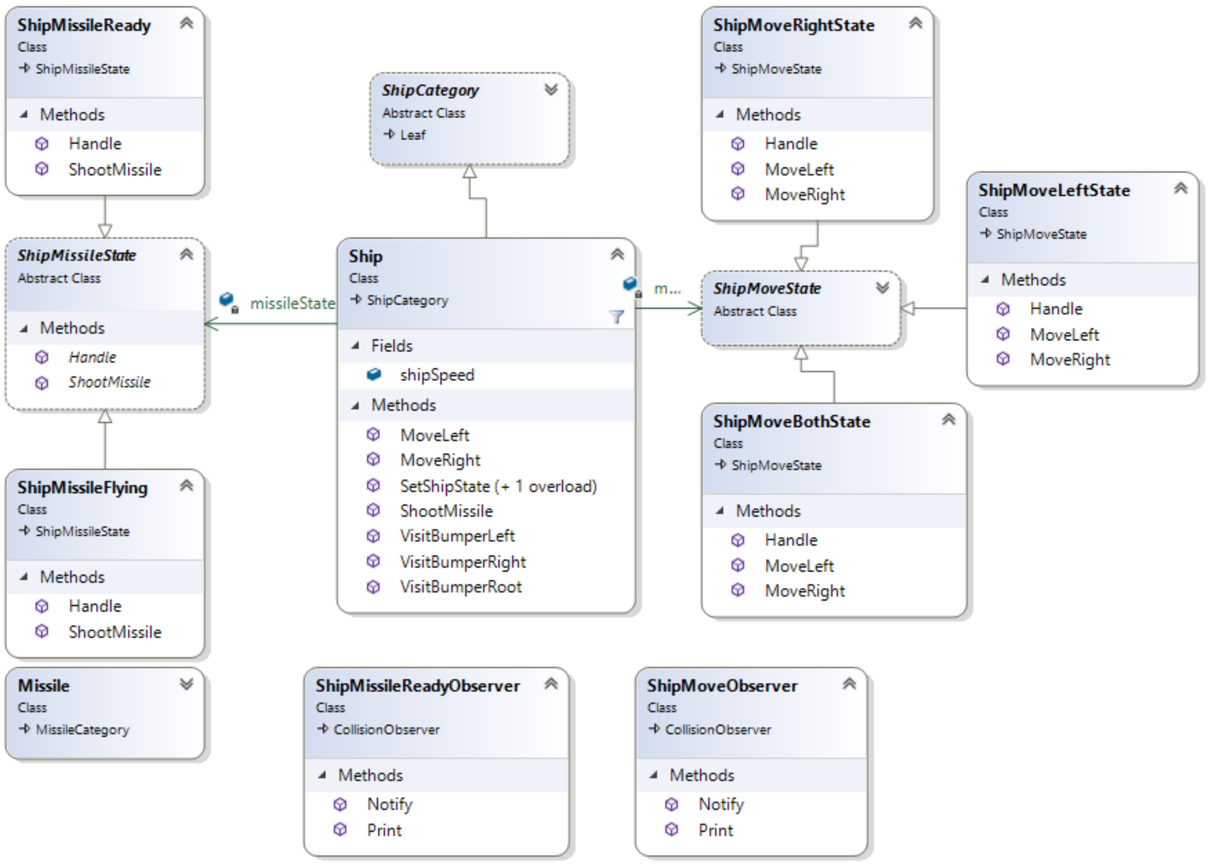
<u>**Discussion**</u>

The problem here is that the aliens drop bombs, and there are three different types of bombs that needs to be drop. The end goal is a bomb needs to be dropped by the aliens, but the type of bombs can be different, and it consist of three different types. Therefore, the strategy pattern is perfect to solve this issue. The goal is dropping a bomb, but what type of bomb does not matter. Hence, the strategy would be fall type. The bomb would have a reference to a fall type strategy and delegate the fall action to the strategy class.

In such design, it makes the code much easier to understand since once the bomb class is created, there is no modification needed if more fall types need to be introduced, it simply be adding more fall strategy type classes.
This is much different than the non-design pattern solution, which would either be implementing distinct methods for each different fall types within the bomb class or define completely new bomb classes for each fall type, and both make it difficult to maintain and require testing for all the individual bomb classes.
This becomes very useful for the factory class, since the factory would only have switch the strategy type when creating different types of bombs. Also, it works nicely with the command pattern since the strategy of what bomb type to fall can be decide at any moment, so when a drop bomb command is being run, the fall type can be decide during the creation of that command.
The main drawback with the strategy pattern is it require creation of many classes since with each strategy type, it consist of a strategy class. Even though most of these strategy classes can be simple, it could add up to many new classes if there a lot of different approach to perform the action.

State Pattern

Class Diagram



General

This is the state design pattern. The state design pattern allows an object to change its behavior when its internal state has changed. For example, an object needs to only be able to perform certain actions based on its internal state. Within each state, the object can only perform a very specific sets of actions based on state changes during run time. The state pattern allows the object to do so without having to implement a range of if statements within the object.

The idea of state pattern is that the object stores the state, and state switch occurs within each state objects, and the actual behavior changes are implemented within each state. This way, more states and behavior changes could easily be added without changing the object itself.

Discussion

The problem here is that the ship object in the space invader game have a few restrictions on missile firing and movement. For missiles, it can only fire one missile at a time, and the missile must either hit a wall or a game object before it can be fire again. For movement, it cannot move pass the two bumpers on the left and right side to get outside of the visible screen. The state pattern is used here to accomplish this is by having two types of states for the ship, one for missile and one for movement.

For missile, it has the missile ready state and missile flying state. To fire a missile, the ship would call the shoot missile function, which would call the shoot missile function within the respective state the ship is currently in for shooting off a missile. Once the missile is fired, it would switch to missile flying state. To switch the state back to missile ready, the missile would need to hit something. In the case, we have a missile ready observer that listens for missile and game object collisions, and when such collision occurs, it would notify the observer to switch the state back to missile ready to allow for missile to fire again.

For bumpers, there are three different states for movement: move left, move right, and move both, and just like the missile actions, the ship's move left and move right function delegate such function call to the respective state object that is currently in. To know when the ship has collided with a bumper, there is the ship move observer that listens for collision between ship and bumpers. When the ship hits a bumper, it would notify the observer to switch to the appropriate state to prevent further movement, and when it stops colliding, it would switch back to the move both state via the handle function.

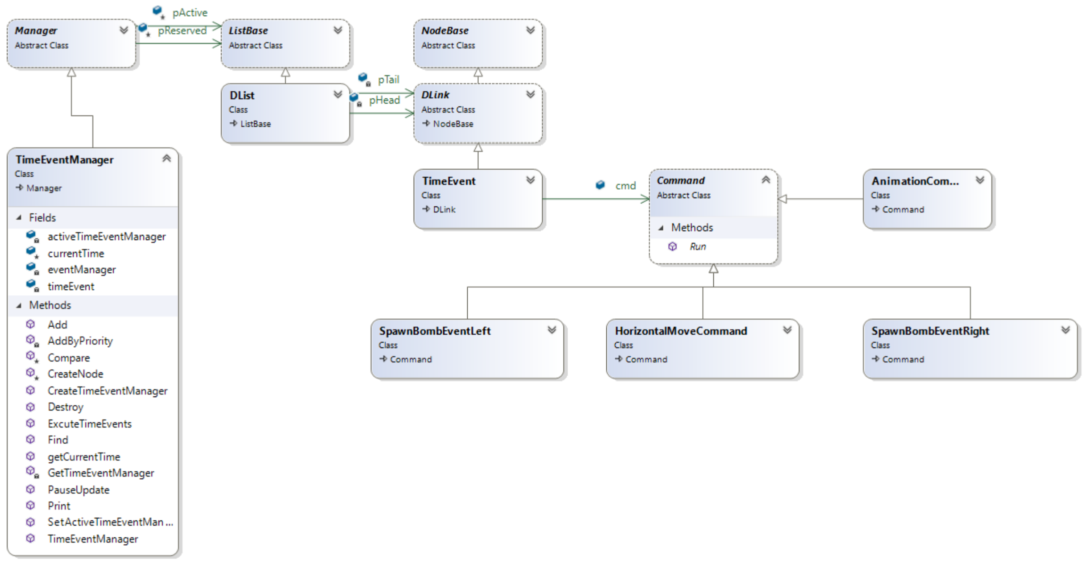
This design pattern is better than other coding solution because it decouples the ship's functionality to each individual state objects. This way, the ship no longer needs to constantly check which states it is in, it would always delegate its operations to individual states, which makes it extending to additional states much easier. Also, it removes the requirement to maintain the ship object itself since all its core functionality is implemented within each state, the ship requires no modification when more states need to be introduced, which makes it easier to maintain and prevent the ship class becomes larger and more complex.

The state design pattern requires interactions within other objects since state switch is controlled by object collisions. It works with the collision operation using the visitor design pattern, where different observers are setup to listen for different types of collision and trigger the state change appropriately. Just as mention above, the missile state change interacts with the ship missile ready observer, which listens for collisions occurs between missile and game objects, and the movement state change interacts with the ship move observer, which listens for collisions occurs between ship and bumpers.

As for trade-off for using the state design pattern, it requires creation of many state classes. As it is shown in the above UML class diagram, the ship has two different types of state changes, and each of those states requires a class to handle that specific case for delegation, which means that to implement state design pattern, more code needs to be written since for each state, it requires a class, and within that class, all operations for that specific state need to be created. Because of this, it would make the codebase more complex to understand and debug.

Command Pattern

Class Diagram



General

This is the class diagram for the command pattern. The goal of command pattern is to encapsulate request as an object. Having the command patterns means that request can be made at any moment with understand what that request do. When a command it being run, the object running the command does not need to understand what that command do or how that command will impact other objects.

Discussion

The problem is that certain action needs to run on the fly without knowing much info. In our game design, the alien movement needs to be synchronized with animations. The goal is to have the aliens to change animations within each movement step. Hence, it needs execution as a sequence of events based on time.

Because of this, the command pattern becomes useful here because by construct commands for movement an animation allows them to put inside a sequence of event execution base on a given time interval. Also, having the command pattern allows for bookkeeping of execution and can be managed easily. Simple command classes can be easily created to perform various action easily.

This ties in nicely with our game design with the introduction of time events and time event managers, where each time events consist of commands. The time event manager would keep track of list of commands using our DList containers and execute them accordingly. This way, the command would always be run in sequential ordering one at time, which provide our goal of synchronized movement of aliens with animation step by step.

However, in order to do this, the events within our time event manager must be sorted correctly with an early out, but this is easy since the base manager class contains all the methods that are needed to insert nodes at various location. Hence, by comparing the triggering time of each node during insertion, it is possible to find the correct location of the time event node by perform inserting into sorted order. This way, it makes much better run time, and it does not require the DList to be sorted every time a new time event got added into the list. Additionally, since it is using the base manager class, the object pool pattern is being maintained as well.

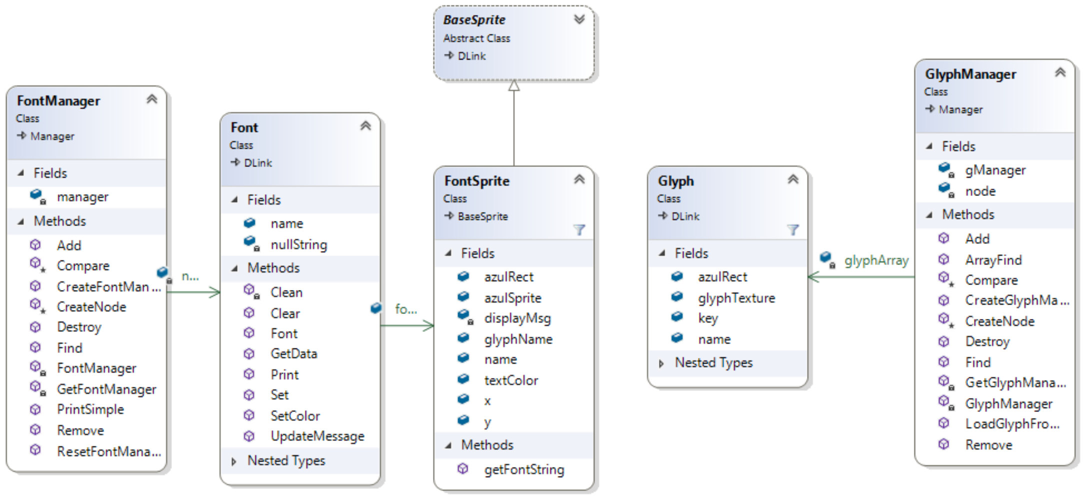
Lastly, with the introduction of time events with command pattern, observers can easily invoke state change for various object types on the fly since the observer do not need to know what the command is for or do, it would just need to know which command to add to the time event manager, and the actual execution of the command would be done. This way, observers would not need much functions and everything can be easily perform using time event thru the usage of the command design pattern.

It is quite difficult to do this without the command pattern since in order accomplish similar goal of sending request to various objects, objects would need to have some ways to communicate with each other to notify the execution. The command logic would also fallen onto each individual object itself, which is not ideal.

The drawback of having the command pattern is that when there are too many events within the DList, some might not get executed in the correct time, so it is important to keep track of the triggering time of each event. A big problem with the inserted into sorted order is that if an event starts much later, and many events would get added would start much earlier, the later event might never get run if there are constant additions of earlier events.

<u>Flyweight Pattern</u>

Class Diagram



General

This is the class diagram for flyweight pattern. The idea of flyweight is sharing a large pool of object effectively across all levels. The data within flyweight are immutable and concrete, which means it does not change, so it can be use by many different client objects at the same time. Often times, the flyweight objects initialized from start, and this save memory because it can provide the same type of reference to many clients at once instead of each client need its own copy.

Discussion

The problem in our game is font needs to be display on the screen. By using flyweight, it is possible to preload all the font characters individual into a glyph object, which is managed by the glyph manager. When a message, which consist of several character glyphs, the font sprite would reference the glyph objects to get the message render onto the screen. The benefit of this is that the font sprite can reference the same glyph object when the same character needs to be displayed. As an example, to display the message “apple”, there only needs one reference the letter “p” glyph instead of needing to have multiple “p” glyphs because the “p” character does not change no matter how many times is being display onto the screen. Because of this, it became very useful to initialize the flyweight characters consist of glyph object.

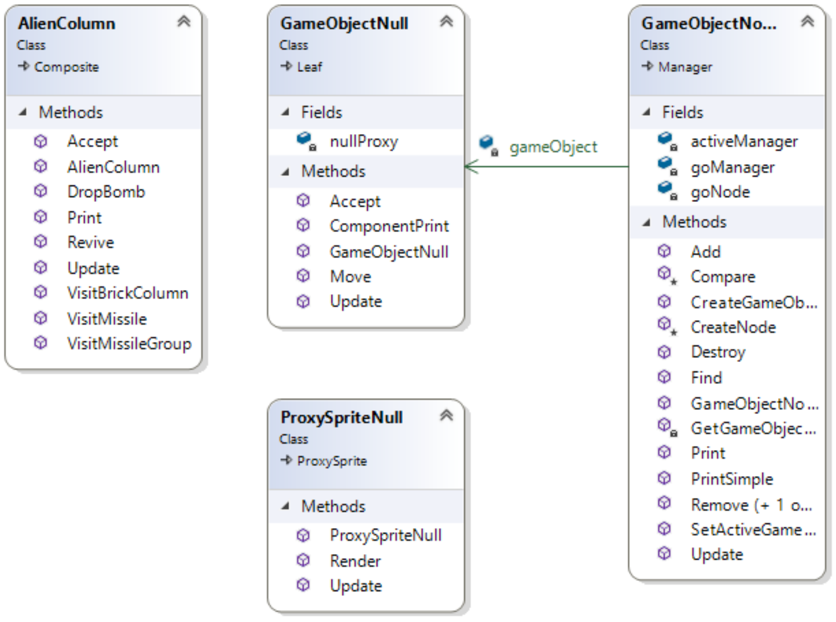
The non-design pattern approach of doing this is every time a character needs to be draw, it would have to go find the character within the texture file, construct the glyph after finding it, and then return this character to the font sprite for rendering. This would cause a major decrease in run time because it requires to find the correct glyph from the texture a string message needs to be render onto the game screen.

This interacts nicely with the attractive display of crawling characters since the glyph objects are always there, so with the usage of the font sprite manager, each font sprite characters can be added and deleted on the fly. Of course, because of our doubly linked list container, finding the correct glyph brings down runtime, so here is the only exception that allows the usage of an array to storage the glyphs for flyweight.

This is much different compare to object pool since with object pooling, it consists of the active and reserve pools of objects, and these pools is only providing to one client. For example, the object is implemented for sprite manager, and those object pools are only available to sprite manager and nobody else. On the other hand, flyweight glyphs can be use by any managers, and it does not need to manage two list, it would just have exact enough to preload everything from start.

The drawback of flyweight is that it since flyweight is shared across, this means that these objects would never get deleted once being created.

Null Object Pattern



General

This is the class diagram for null object. It is essential a class that does nothing. This prevents the handling of null reference error by having this in place within the structure.

Discussion

In our game design, certain objects are basically a representation since it does really do anything besides being a container that holds thing. Example of this is alien columns, brick columns for shields. However, these objects still exist within the tree hierarchy and needs to be updated accordingly, which is why null object is introduced. This way, things would still function without much modification. These null object classes consist of the same methods just like the respective non-null counterparts, with empty method bodies.