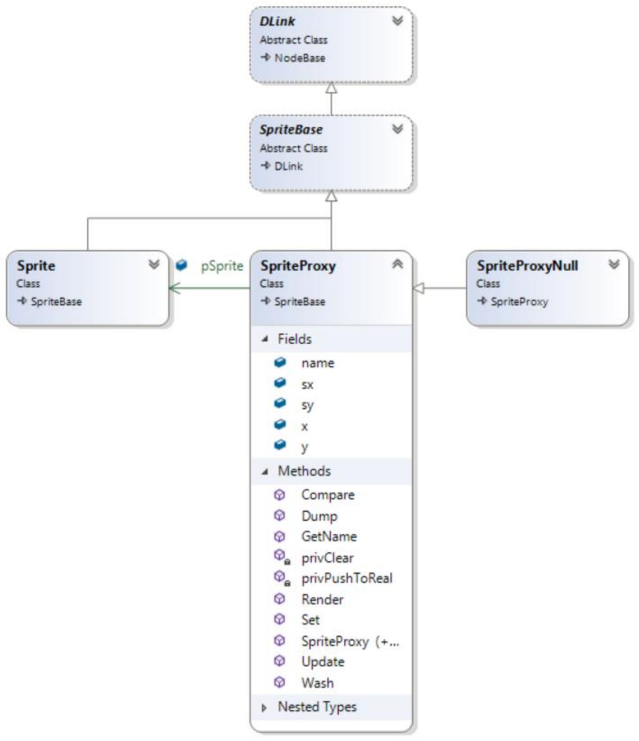
Game Design Document

Joey Domino

Proxy



Problem:

Game objects will be constantly manipulated and checking against other objects in the game. This would slow the game down considerably if game objects held every piece of data. We require a lightweight way to reference game object sprites to move, resize, swap images, and swap their colors without needing to grab every piece of data per check.

Solution:

We have created a sprite proxy class to be used in the sprite's stead. The sprite proxy's job is to store the name, x and y coordinates, width, and height of any sprite the proxy is holding. A pointer to the sprite itself is then stored within the proxy for reference when needed.

Pattern:

Let's look at an example of the proxy being created and used for an alien squid. The alien factory has called for a new squid to be created. The system first calls the alien squid constructor, which calls its base class alien category, then calling category's base class Leaf, then the game object, before reaching the component template class (see Template for more on this). After setting the component template, we step back down to the game object class to build out the object.

Our game object stores the name of the object, in this case an alien squid, the coordinates where it will be on screen, the sprite's name (also alien squid but from the Sprite's enumerator. This is important for finding the sprite to swap data of the sprite and not the game object during runtime), then calls our sprite proxy manager to add our sprite to a double linked list of sprite nodes. At first, the node is created completely empty using a null object (see Null Object) then the data is set to the node to be used in the game. A sprite box is also attached to the game object which is set up similarly to the sprite proxy. The difference being that the box is a null object with four lines outlining the empty space. This is how we will check for collisions within the game (see Visitor).

Mechanics:

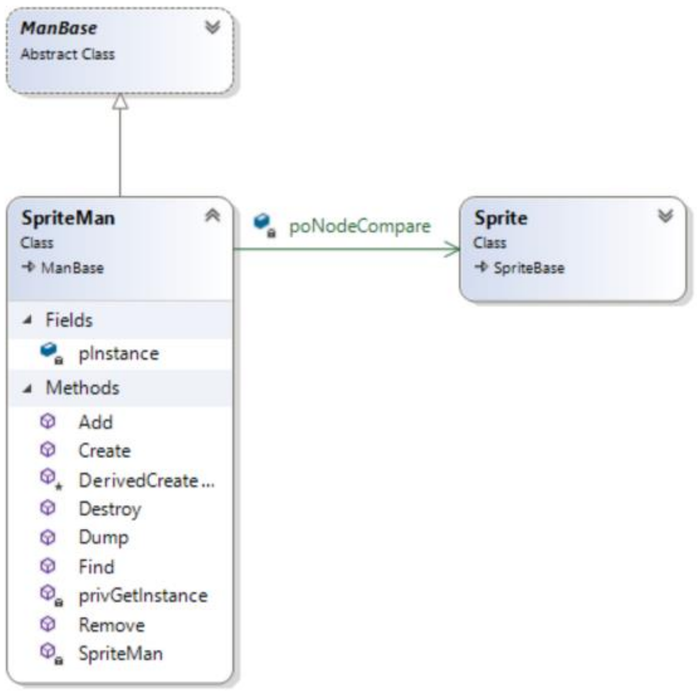
Now that our alien squid has been stored and our game object can reference it, we can manipulate the proxy during runtime. When playing the game, we need to simulate movement of the alien by altering the sprite when the squid is moving along the x and y axis. We do this by swapping its image in tandem with the move. This is possible through the timer event system (see Commands).

So, we have our timer system in place and added an animation command. This command is enacted every 1.0 float units and speeds up as the army is reduced and by however many times the player has completed a level. The animation command grabs the sprite we want to animate, and swaps its image out with another from an iterator (see Iterator) that was previously built.

What Does It Do For Space Invaders:

You may be thinking "but didn't you just completely ignore your proxy?" and that's the best part of the proxy! This swap happens behind the scenes so the game object does not need to worry about it. This allows the game to run faster as not all data is being passed around to every action. It's segmented. The sprite proxy can point back to the sprite at any time and show whatever image the sprite within it holds. It doesn't care if the image was swapped somewhere else. It just holds the data toward the sprite. To display the sprite, we just call the sprite's Render() method. The Render() method is where we push our data to the screen. This system reduces the amount of times we push to the screen, which greatly increases the game's speed.

Singleton



Problem:

Space Invaders holds many items that all require easy referencing. Sprites, sprite proxies, sprite batches, sprite boxes, images, textures, and more all require managers to function properly. These managers have been created, but they must be able to be duplicated easily.

Solution:

To solve this, we have transformed every manager that derives from the base manager template class into a singleton.

Pattern:

The pattern for a singleton is pretty simple in construction and execution. To change a class to a singleton, you must privatize its constructor and make all methods you would like any copies of the class to reference into static methods. We have created a Create() static method to act as a way to build our duplicate singleton. This allows us to separate our managers as needed to only see certain lists during runtime.

Mechanics:

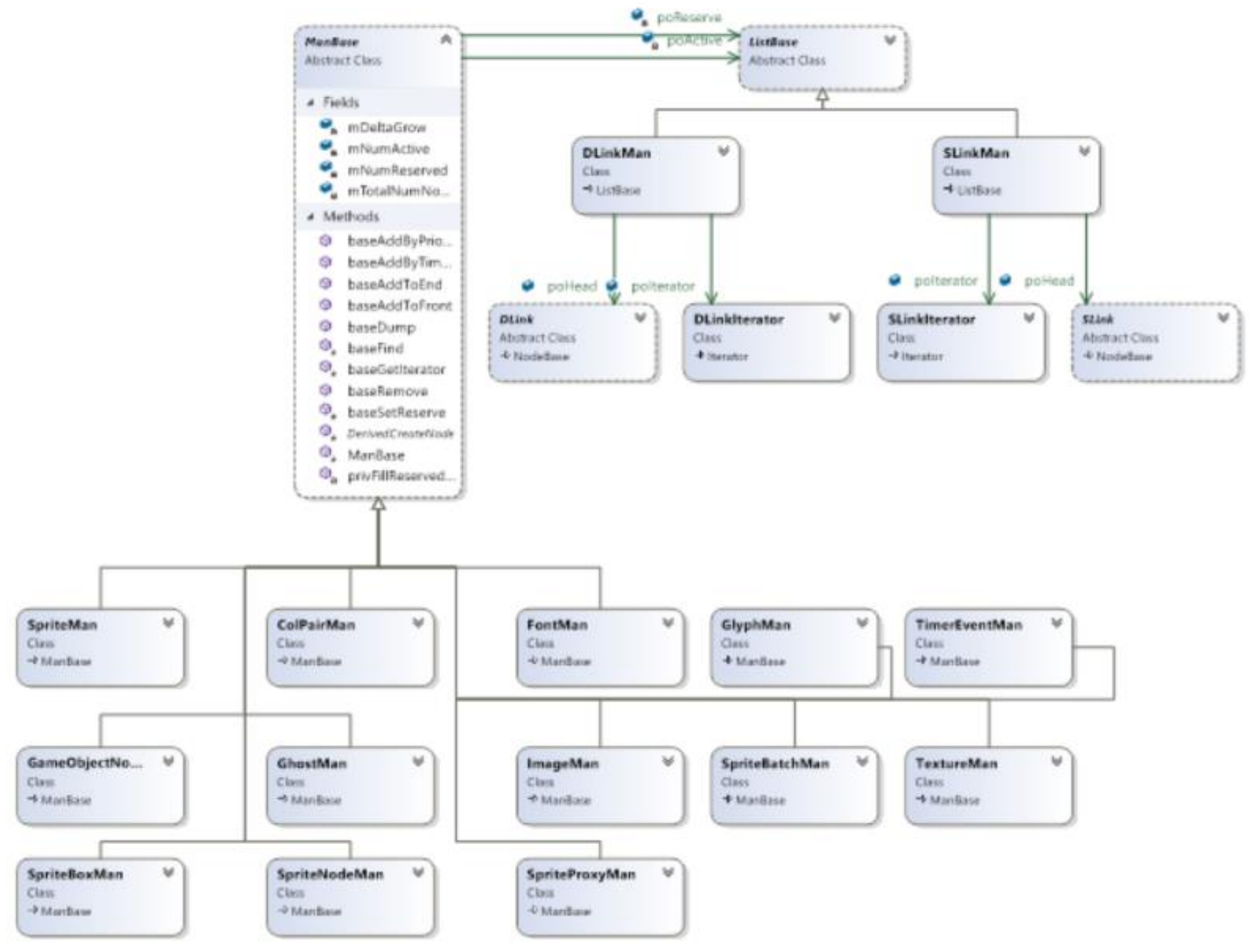
Someone has just started up the game. What do we want them to see? We need 4 sprites, the UFO, squid, octopus, and crab aliens, to show their scores. We need text fields to say what our game is, print the scores, show how many credits they have, the high score, and what to press to start playing. Then, when they play the game, we need a separate list for textures, sprites, images, game objects, texts, and more. We want to make sure the two screens do not get the same information shown.

In comes the singleton. When we load the second scene, we can create another list of managers to split the two screen's list of items so they never interact. Our current implementation was simple. We created separate sprite batch lists so only the sprites on a given scene is shown, but it would be far cleaner code wise to make a new list of managers per scene.

What Does It Do For Space Invaders:

Singletons won't do much for the player, but it cleans things up behind the scenes immensely. Like I stated above, it would be easier to manage the data in each scene if we split them up with copies of each manager the scenes require. For example, we could keep a sprite batch manager global for the scoreboard so those fields are rendered in every scene. It's a problem we faced and were unable to rectify before the deadline. If I was proactive with my setup weeks in advance, I wouldn't be suffering now with my single manager.

Object Pools



Problem:

Seeing as the game has many objects in need of referencing, we need a way to keep track of them. As stated in the singleton section, we require lists of every item in the game to find and modify them as needed. This includes when the items are removed so we can reduce the amount of times we allocate new memory during runtime.

Solution:

We have created a single and double linked list to be used by the base manager and, by relation, every manager attached to it. These are used for tracking both existing and removed nodes of any item.

Pattern:

It may look complicated, but it’s relatively simple. Each manager has two doubly or singly linked lists associated with them on creation. These lists get populated when the manager creates a new item of its respective type. So an image would add to the image manager's active list, and a sprite to the sprite manager's. Whenever we decide to remove that item from the game, it is found on their manager's active list, wiped clean so all data is nullified, and added to the reserve list for later use. Then, when we create another new object, we first check if the reserve list has any nodes able to be used. If so, take it from that list, add it to the active list, and fill it with the data we want. Otherwise, we allocate a new active node and populate that one. That's it!

Mechanics:

To explain further, let's stick with sprites. Before creating the sprite, we need to allocate a texture to reference the image from. We tell the texture manager to create a new node. If there's a reserve node, use that, otherwise make a new one and store our texture data. Now that we have our texture stored, we can grab an image from it. The image manager does so and stores the image node within its list. We can now create our sprite with the image grabbed from the texture. The sprite manager will do as the others have and either grab from its reserves or make a new node and populate it with the sprite information.

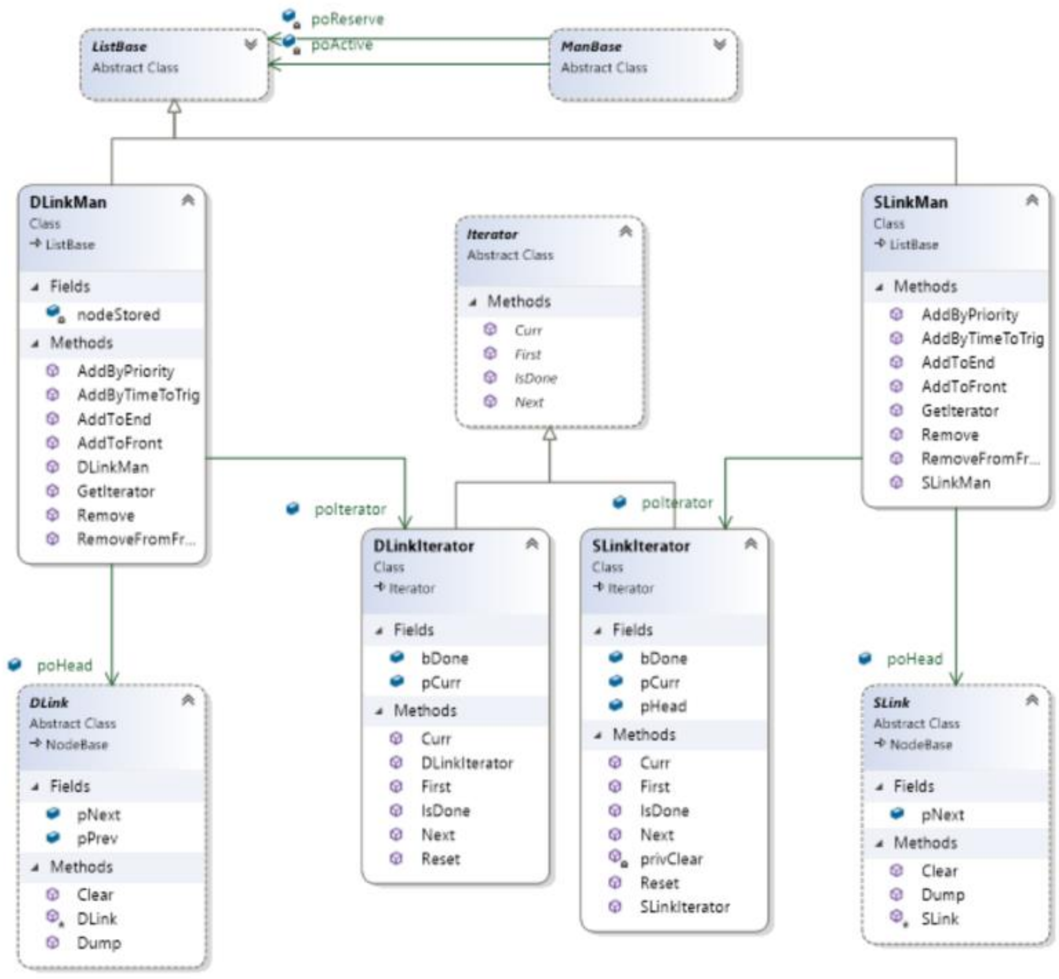
So, we have created a new alien squid sprite. Great! This system will add it to the sprite manager's active node list and allow it to be referenced. Now that the sprite is in, we need to attach it to a sprite proxy. That proxy is now also added to its own active list of sprite proxies. We then add the proxy to our game object after the game object was also created. Whew! All done…

Or are we?! To be able to see our created sprite, it needs to be added to a sprite batch list to be able to be rendered from. Now our image can be seen on screen, the game object can modify the sprite's location through the proxy, and texture…well the texture gets to sit and wait to be referenced again for a new image to be grabbed from it. Simple!

What Does It Do For Space Invaders:

This insanely complicated system of lists allows the game to easily reference, modify, remove, and create new nodes while also reducing the amount of new allocation during runtime. This is insanely important as the reduction of news will greatly increase the speed of our game. Imagine if each image added was a particle on the screen. If we didn't allocate these ahead of time, the game would chug horribly when spawning them as the system would be trying to allocate hundreds if not thousands of new elements while the game is running. Instead, we can now store any removed items and use those instead of creating new items.

Iterator



Problem:

When creating our lists, we need a way to step through them to find the node we are looking for. We then need to be able to remove the node on request.

Solution:

We have created, as shown above, doubly and singly linked list iterators.

Pattern:

The pattern here is simple. The iterator itself is a template (see Template for more) for the doubly and singly list iterators. These are then referenced in the manager base for every manager derived from it to use.

Mechanics:

We'll continue with sprite creation for this example. When a sprite is created, we call upon the doubly linked iterator to add it to the list where desired. We have 3 ways to add and item to a list: to the front, end, or by priority.

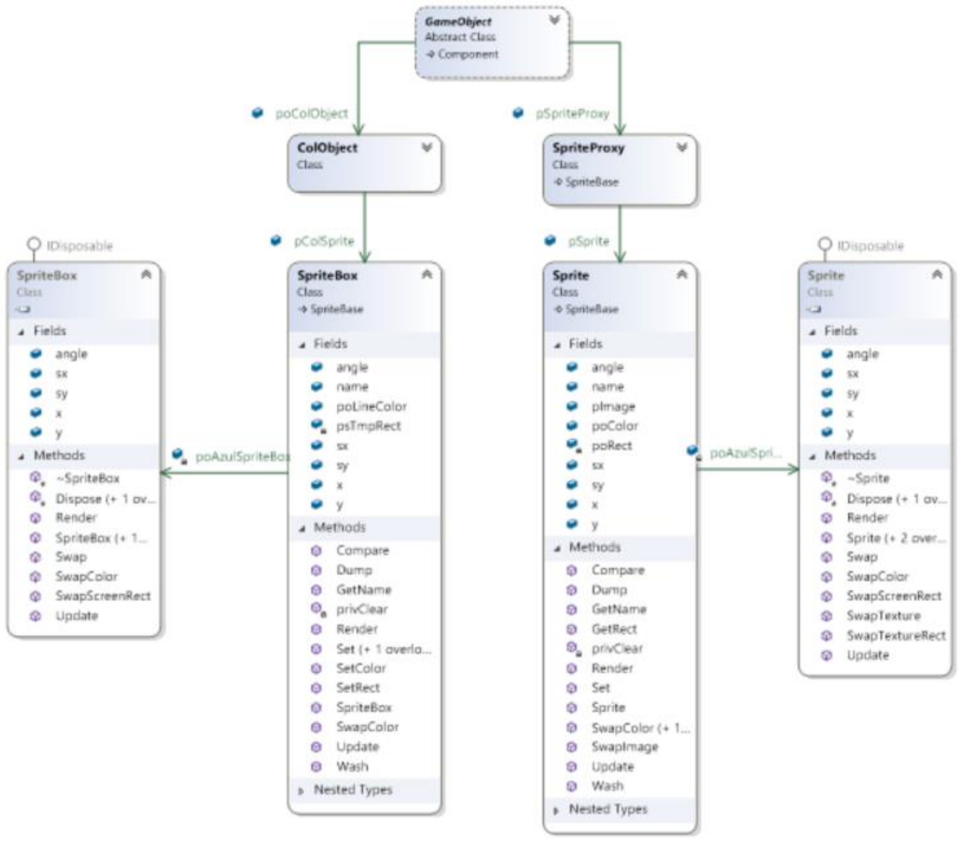
When passing in a node via adding to the front, the iterator will replace whatever the head node is and set the incoming node to the head and the previous head node to the new node's next pointer. If adding to the end, it will first check if empty and add as the head. Otherwise, it will step through the entire list and add the node once the current end node's next pointer is null. It will set the incoming node to the new last node and update its previous pointer to the list's old last node.

Adding a node by priority will store the node inside the list at a given location between previously existing nodes by referencing the priority integer passed alongside it. This is primarily used for rendering so we can decide what to render above something else.

What Does It Do For Space Invaders:

This system acts as the backbone of the entire game. Without this system, the entire game won't function. The iterator system allows the game to keep track of every item in the game and clear or add nodes as required. It's surprising how such a simple system can be so integral to a game being able to run at all.

Adapter



Problem:

We require a way to push data to the screen using given Azul methods from classes that otherwise could not call them.

Solution:

We have created adapter systems to allow such methods to be called at any time in our game.

Pattern:

The system here is pretty simple. Any given game object has a sprite proxy and collision object that need to be rendered to screen. However, game objects cannot reference Azul commands as they are not of the same type. Azul would throw errors left and right if we tried to do that. To get around this problem, we have added a Render() method for both sprite boxes and sprites. This method pushes the x and y coordinates, the sprite or box's width and height, and an image or box to the screen at that location. This allows us to modify any game object's information and push it to the screen at any point during runtime.

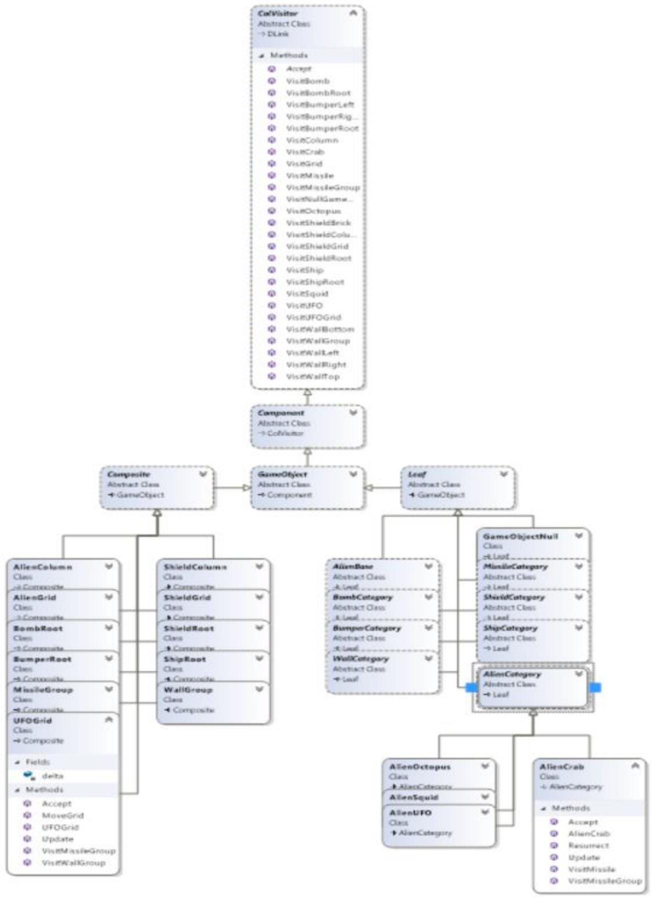
Mechanics:

Let's say we want to update a sprites location on the screen. We first need to grab the game object by finding it in the game object node manager. Once found, we can then step into the game object to get its sprite proxy. We can call Render() from the proxy here. This method will push the proxy's data to the sprite, then call sprite's Render() which will update Azul's information. Pretty simple to type out, but one false move in the code and you're in error city.

What Does It Do For Space Invaders:

This is another method to reduce the amount of times we call the screen. Reducing that allows the game to run at a much faster rate. That's pretty much it.

Template



Problem:

We need a system in place that allows multiple classes to redefine certain methods without changing the structure.

Solution:

We have created a template system that allows classes to override methods so long as they derive from the abstract template class.

Pattern:

One of the major components of the game is to allow game objects to collide with each other. We remedied this problem with a seemingly complex, but actually relatively simple system. The ColVisitor template is an abstract method that holds all types of collisions possible in the game. It is then up to each game object to handle their own respective collisions by overriding the abstract methods put in the ColVisitor class.

Mechanics:

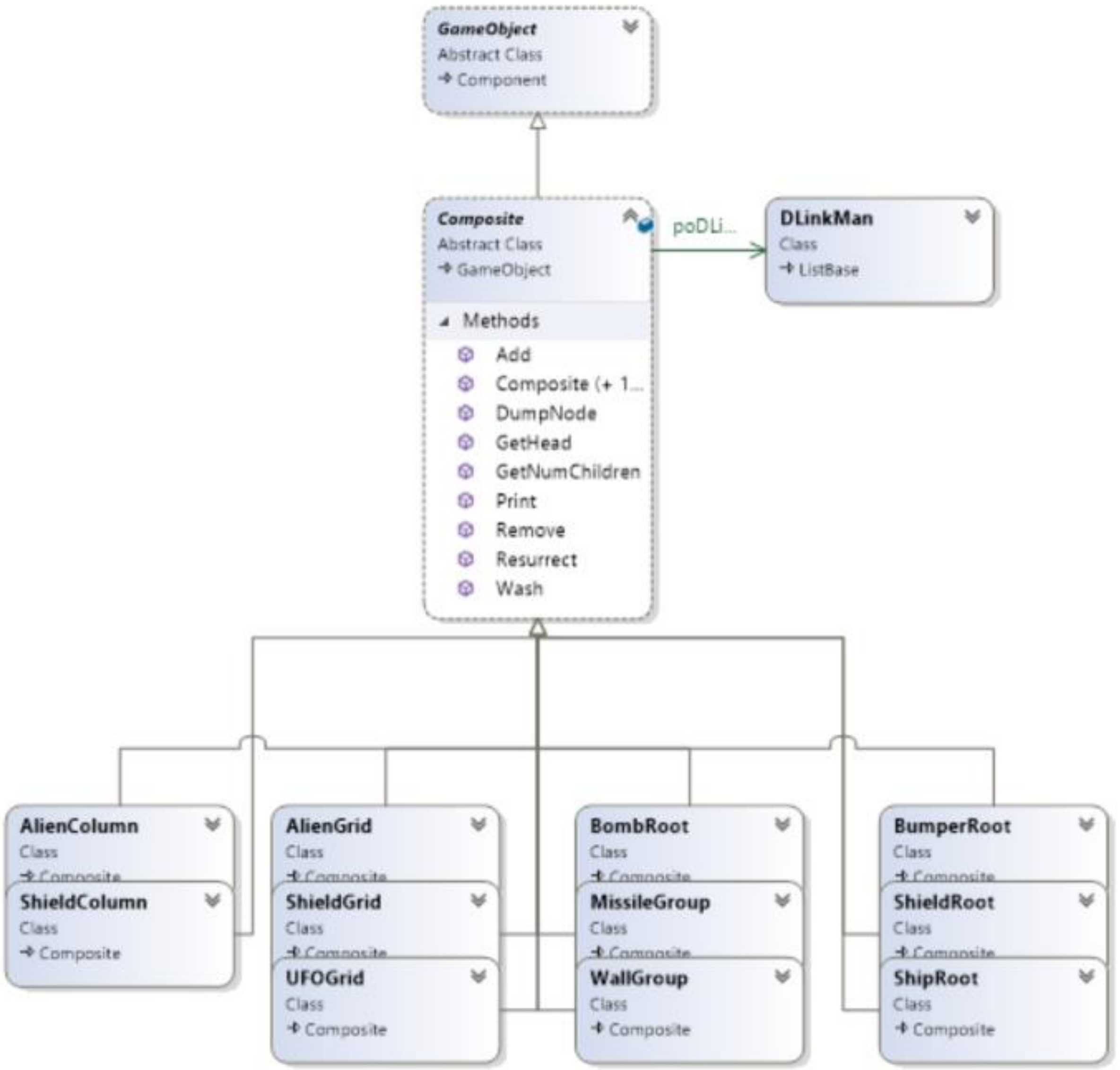
To explain this clearly, let's look at how the grid bounces off of the walls to know which way to move and when to move down. The walls are created as box sprites and placed within a wall category leaf as game objects. Each of these wall types, being top, bottom, left, and right walls, need to contain an accept override method along with any visit x game object methods required to function. Same goes for the alien grid.

So in this case, we have VisitGrid for the left, right, and bottom walls and VisitWallGroup for the grid. When colliding, we check if the two boxes intersect. We do so by looking at each child within the wallgroup in this case and see if those collide with the grid. On hit, we then need to notify listeners (see Listeners below) and execute each of those commands. In this case, we only need to call the GridObserver observer class which will invert our movement on left and right wall hits as well as move the grid down once based on the deltaY set within the Grid's class. If the grid hits the bottom, we would remove all timer events as the game has ended and add in our GameOver event. This will end the game and bring us back to the main select screen.

What Does It Do For Space Invaders:

The template system allows the game to interact with different game objects underneath a similar field. Without this system, collisions would be far more complicated to enact. With this system in place, we can easily alter objects during runtime of the game and keep the flow of the game going without any need to stop.

Composite



Problem:

We need a way to keep track of game objects that are grouped together on the screen.

Solution:

We have created a derived game object class called a composite. This abstract class will hold multiple derived classes of null_object (See null_object) types with only collision boxes within them.

Pattern:

When building out the alien army, we need to keep track of them as one large unit. To achieve this, we have created a series of composite columns that hold 5 aliens in each, 2 octopus, 2 crabs, and 1 squid. These columns, 11 in total, are then added to a grid composite. This grid composite is used to check collisions against walls to handle movement as was covered above.

Mechanics:

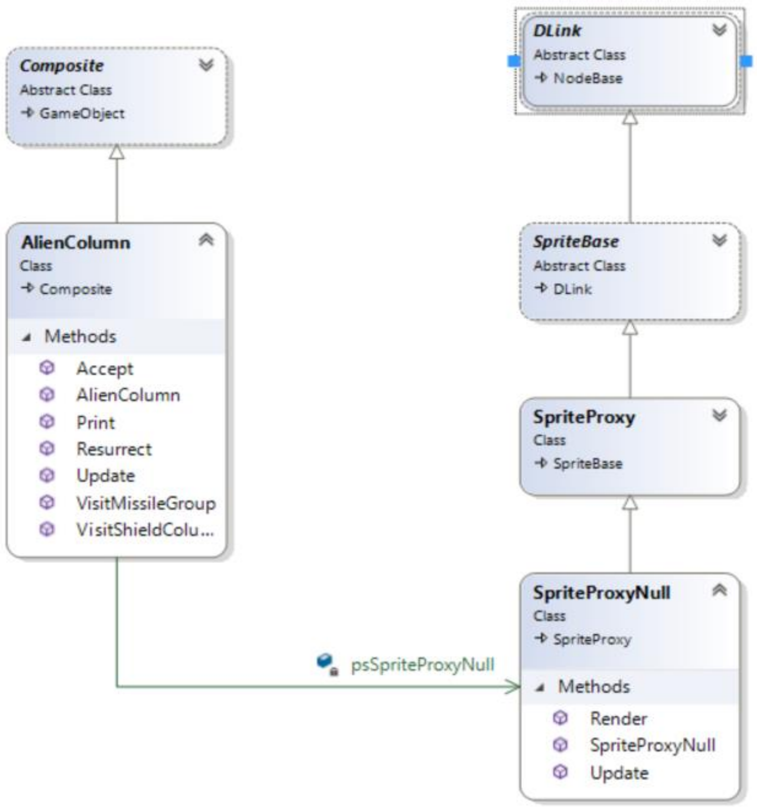
In game, the composite objects are used mainly to check collisions of various types. As was mentioned, the grid will check for collisions against the wallGroup composite object. The wallGroup composite contains 4 wall box sprites that are used as collision objects. When the grid hits the left or right walls inside the wallGroup, the grid is told to move down and invert its direction.

As another example, the columns are checked when spawning bombs (see Strategy for more on bombs). We have created a bomb spawn command (see Commands) that will randomly choose one of the column composites to then spawn and drop a bomb beneath them. We used the column instead of a particular alien to reduce the amount of code required to drop the bombs. If we went with checking the aliens, we would have to search through the entire column to find the lowest alien in the list which is unnecessary with our current setup.

What Does It Do For Space Invaders:

Composites allow the game to easily collide objects with each other. If these composites were not in place, we would need to search our entire game object node manager for any collisions with every object. That is very resource intensive and completely ridiculous. This system reduces what would be hundreds of checks a frame to roughly 20 at most.

Null Object



Problem:

We need a way to create objects even when they may not have a sprite associated with them.

Solution:

We have created multiple variations of null object types to be used on initialization of any object.

Pattern:

The system for null objects is a simple one. Just like creating a sprite proxy or sprite, the null object is merely an object with nothing attached.

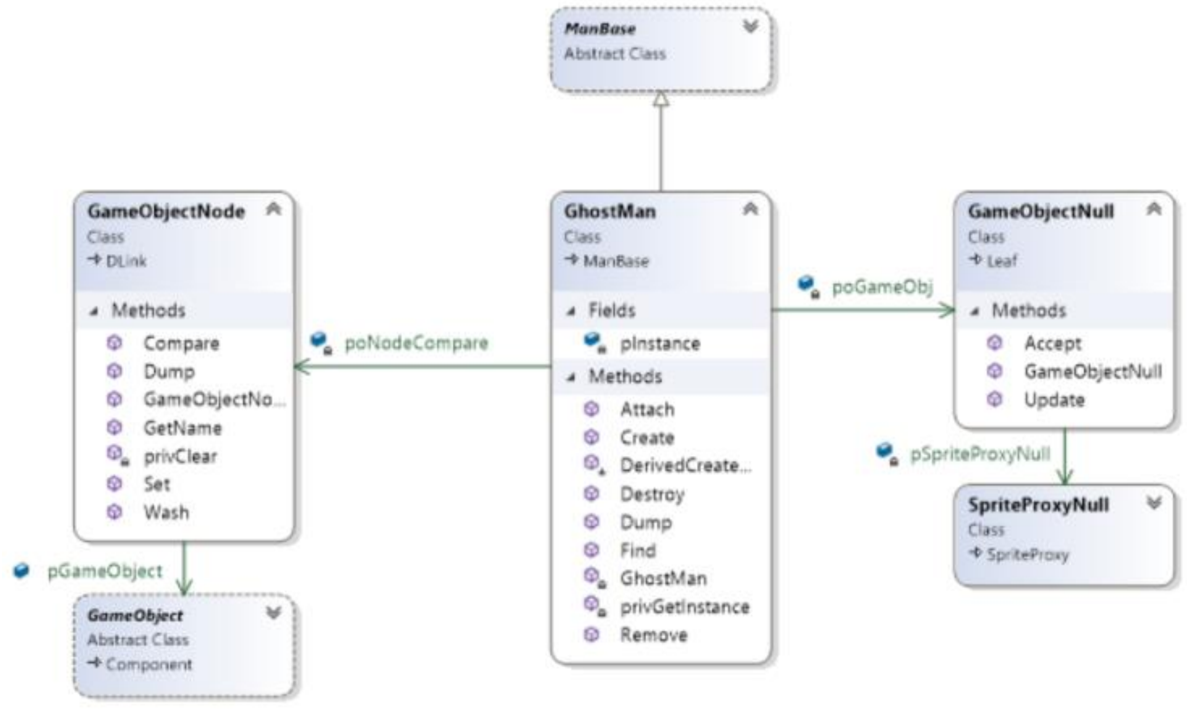
Mechanics:

Let's look at alien column creation as the UML diagram shows. When we create a new alien column, it is initialized with a null sprite proxy. This follows the same flow as a sprite proxy, but instead of setting any data, it merely passes on that the proxy is null. This in turn also sets the sprite as null. This allows for easy and rapid creation of objects to either be filled in later, like a game object, or not, like the column. The column then gets populated with a collision box before being populated with the 5 aliens as stated above.

What Does It Do For Space Invaders:

This is another optimization tool. Creating objects with null object data allows us to front load our active lists with empty nodes that can be filled with data shortly after. This speeds up the game on load and during runtime whenever we may need to create a new object in real time.

Flyweight



Problem:

We need a way to store game objects when they are removed from the game.

Solution:

We created a flyweight system to handle the removal and re-adding of game object nodes.

Pattern:

The flyweight system functions much the same as every other object pool system. What makes this pattern special is its use. Whenever a game object is removed from the game, it is cleared from the sprite batch and game object node manager before being added to our flyweight, the ghost manager. This manager's only job is to store all killed game objects for resurrection at a later time.

Mechanics:

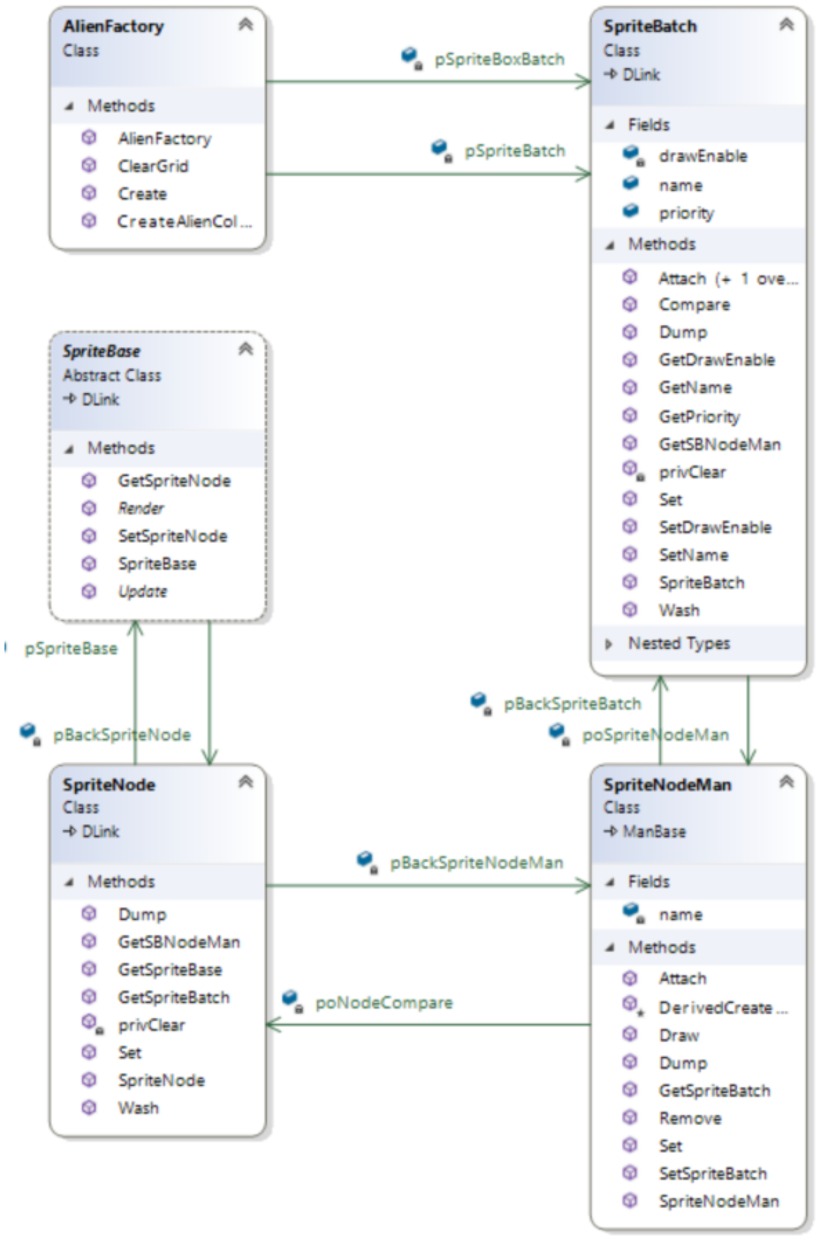
The quickest example of this is the player. If the player is hit by an alien or a bomb, the player dies. An animation is played and shortly after, the player's ship is killed. This is done by removing it entirely as stated in the pattern. After a few seconds, we call the Resurrect() command within the ship class. This will first attempt to find the ship within the ghost manager. If it does, the ship will be taken off that list and re-added to the game object node manager along with all of its relevant data. The ship will then have its states change (see State) to be ready to fire and move, allowing the player to continue playing.

What Does It Do For Space Invaders:

The flyweight is integral to the game's ability to be recursive. It drastically reduces the amount of new allocations made by the game to nearly 0 after the game is first booted, as all data will have already been loaded in. The main use in terms of gameplay is the resetting of the level and after losing the game.

When all aliens are killed, the game pauses for a moment, and all aliens are resurrected along with their columns. The same happens with the shields. However, we have put in place a system that also checks for any existing shields to save time resurrecting them. If they already exist, there's no need to remove them just to bring them back!

Factory



Problem:

We need a system to create our alien grid and shields automatically.

Solution:

We have created two factories to handle creating aliens and the four shields that spawn just above the player.

Pattern:

The UML diagram for the factory is very misleading. This system takes in multiple game objects and churns out a fully built out grid of, in this case, our alien army. Both factories have a public Create() method that takes in a series of arguments to create a slew of game objects, activate their sprite in the sprite batch and their box in the box sprite batch, create the column the game object is going into, add the game object into them, then finally add the columns into a grid to finally return the finished grid game object.

Mechanics:

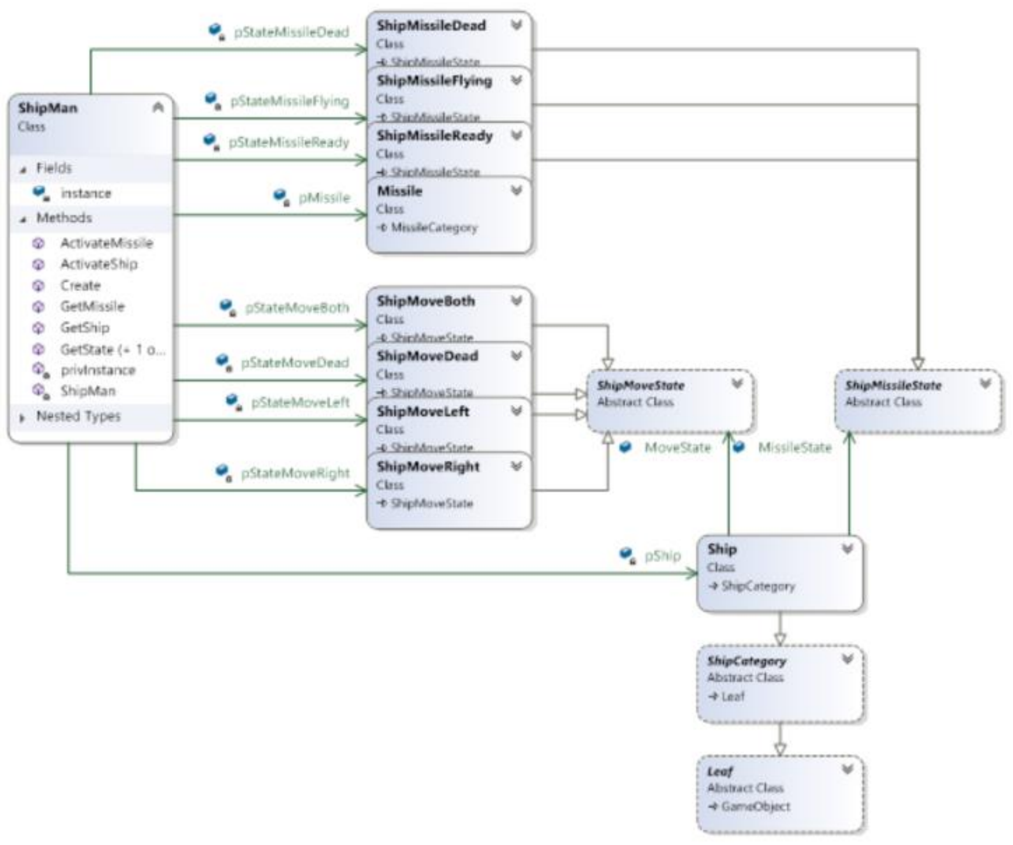
Let’s look at the creation of shields in this example. Our rendition of the shield factory creates one column at a time and returns them to be added to a grid, which in turn is added to a root as well. We require a root node, which acts as a sort of grid of grids, because when these shields are removed, their grids are as well. If we didn’t wrap all 4 shields within a root node, we would throw an error when one of the shields is fully shot down. As stated above, this factory and the alien factory both check for existing objects before creating new ones.

The special case here is if the brick already exists, it does not get removed. All resurrected or newly built bricks are placed around whatever may still live on screen. This further enhances our resource management as there is no reason to remove game objects from the screen if they don’t need to be.

What Does It Do For Space Invaders:

Gameplay wise, nothing. The factory exists to make building out the shields and alien army quickly and easily programmatically. The player will only see the finished product after everything has been built. This does make resetting levels far quicker as we only need to call the factory to recreate our grid in one line of code rather than duplicating over a hundred lines to do the same thing in a different location.

State



Problem:

We need a system to control the various states the ship can be in.

Solution:

We created a state system to set and change what setting the ship is in.

Pattern:

This pattern may look complicated, but it's really multiples of the same class with minor changes. The state system is a pattern used to remove a lot of cluttered conditional statements. Instead of them cluttering a class, we created multiple small classes that are linked to a template. When certain conditions apply, which are set via our collision system (see Visitor), ship observers (see Observer), or a timer event (see Command), these states are changed which then immediately alters how the ship is handled.

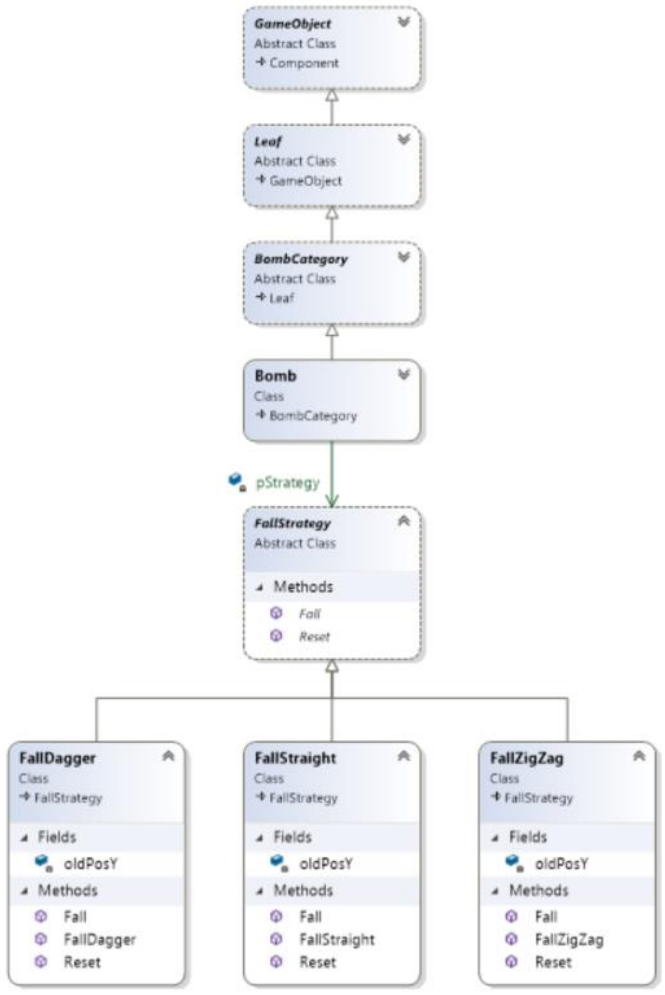
Mechanics:

Our ship is not allowed to hit the corners of the game screen. To make sure this doesn't happen, we have place in slightly larger box collision sprites we called bumpers at the bottom of the screen. When the player's ship collides with either of these bumpers, an observer fires off to turn off the direction they are moving in. So if they hit the right bumper, the can no longer move right. If they hit the left bumper, the can no longer move left. Once they move away from either bumper, the state changes back to a MoveBoth() state, allowing the player free movement in either direction.

What Does It Do For Space Invaders:

This restricts the player from going out of bounds or too far too the left or right than we would like them to go. It also restricts their firing so they cannot rapid fire shots and win easily.

Strategy



Problem:

We need a way to allow any bomb type to fall a particular way.

Solution:

We have created a fall strategy that every bomb must follow.

Pattern:

The strategy pattern is very similar to the state pattern. However, the strategy pattern allows us to change our objects behavior by pairing it with a different game object.

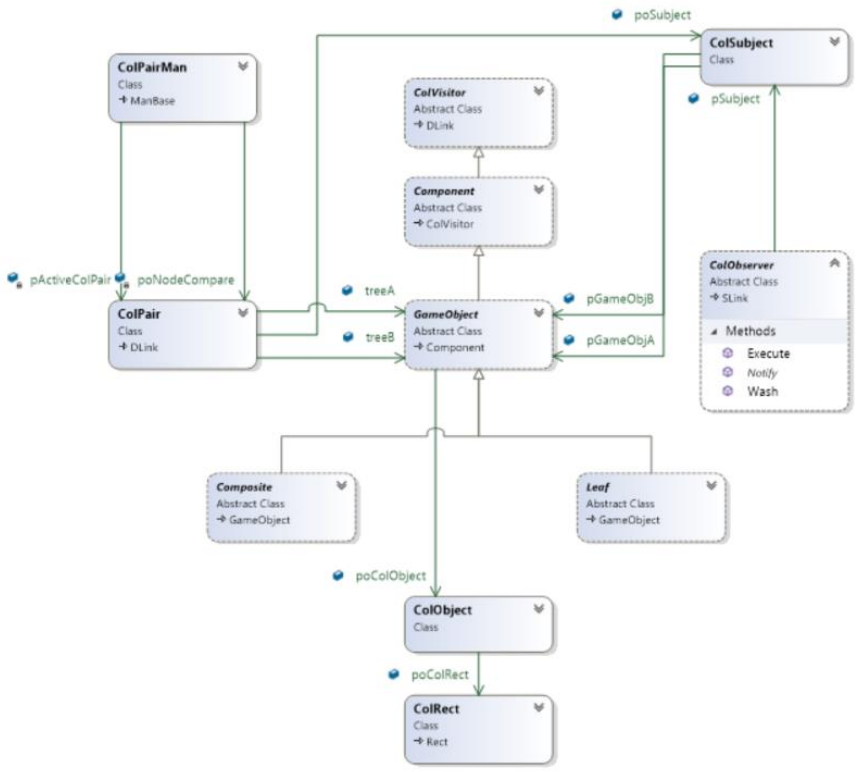
Mechanics:

As what may be obvious in the UML diagram, we have 3 different types of bombs that can be dropped. Each of these bombs have different properties when falling. Instead of giving them a state to all follow, we set our fall strategy alongside the type of bomb game object we create. So a zigzag bomb game object uses the zigzag fall strategy. The same goes for the straight and dagger types. This allows easy implementation with low code overhead. Once this is set in place, as Todd Howard would say, it just works!

What Does It Do For Space Invaders:

The fall strategy allows us to spawn various types of bombs using the same codebase. For the player, it allows great variety when playing to keep the game interesting. It'd be pretty boring only seeing the same bomb drop every time.

Visitor



Problem:

We need a way to check for and catch collisions with any game object we desire.

Solution:

We created a visitor system that every game object must follow to be properly collided with.

Pattern:

The pattern for this one is quite the beast. Every game object derives from the template class ColVisitor. ColVisitor holds an abstract method for every type of collision the game has. It is then up to every game object to handle their own collisions and accepting of being collided with for this system to work.

Mechanics:

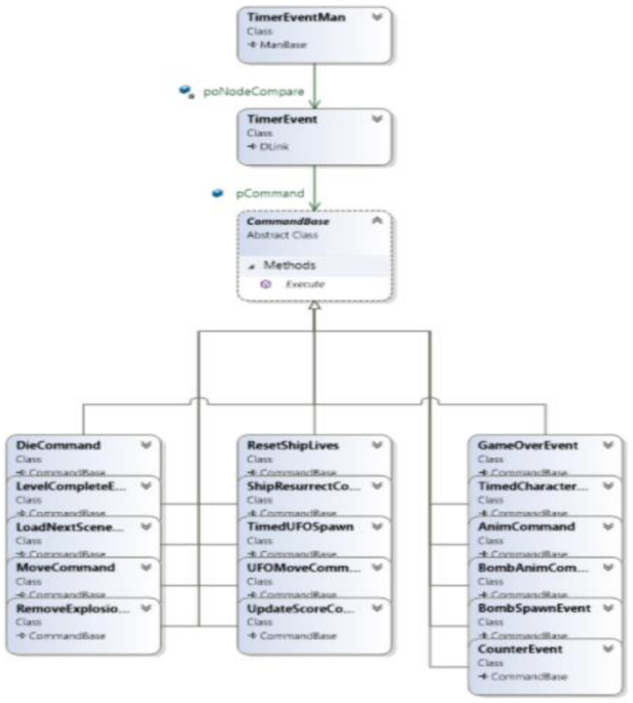
Let's walk through a collision with a missile colliding with an alien. When the ship fires a missile, the game object is added to a missile group. You can think of the group as a type of grid. Every frame the missile group is checked to see if it has collided with the shield root, wall group, or alien grid. This is checked by ColPair's Collide() method. On collision, one game object calls Accept() on the other, which then will either notify observer listeners about the collision or dive deeper to the children of one of the object to then check again if those two objects collide.

This continues until either the objects collide or they don't. Then the cycle continues. Returning to the missile; if the missile group collides with the alien grid, then we dive deeper into the alien grid to see if the missile group has collided with a column. If that has occurred, we take the child of both objects to then see if the missile itself has collided with one of the aliens in the column. If it has, then we notify the observer listeners to do whatever we want them to do. In this case, we will remove the alien from the column, remove the missile, play a death sound, replace the alien with a splat sprite on the screen, and reset the ship state to be ready to fire again.

What Does It Do For Space Invaders:

This is where the actual game comes together. This system drives everything the player sees on screen. Without the visitor pattern, we have no game. The visitor pattern handles everything that can happen in the game. This system in tandem with the observer and command systems (see observer and command below) are the brains of the game.

Command



Problem:

We need a system to allow recurring commands to occur over differing amounts of time.

Solution:

We created a command system that uses a local timer to track when to execute commands.

Pattern:

This pattern is relatively simple. To start off, we need a timer for the game scene. This will be used to keep track of when to execute our commands. Second, we need a timer event manager to track our list of commands to execute. With those in place, we loop through our game's update method and constantly grab the global time to set our current time.

Now that we have our time, check it against when we want to execute our command. If the current time minus the global time is greater than or equal to the time to execute the command, execute it. Some commands will reset themselves into the list to simulate movement or animation. Others will be one off commands.

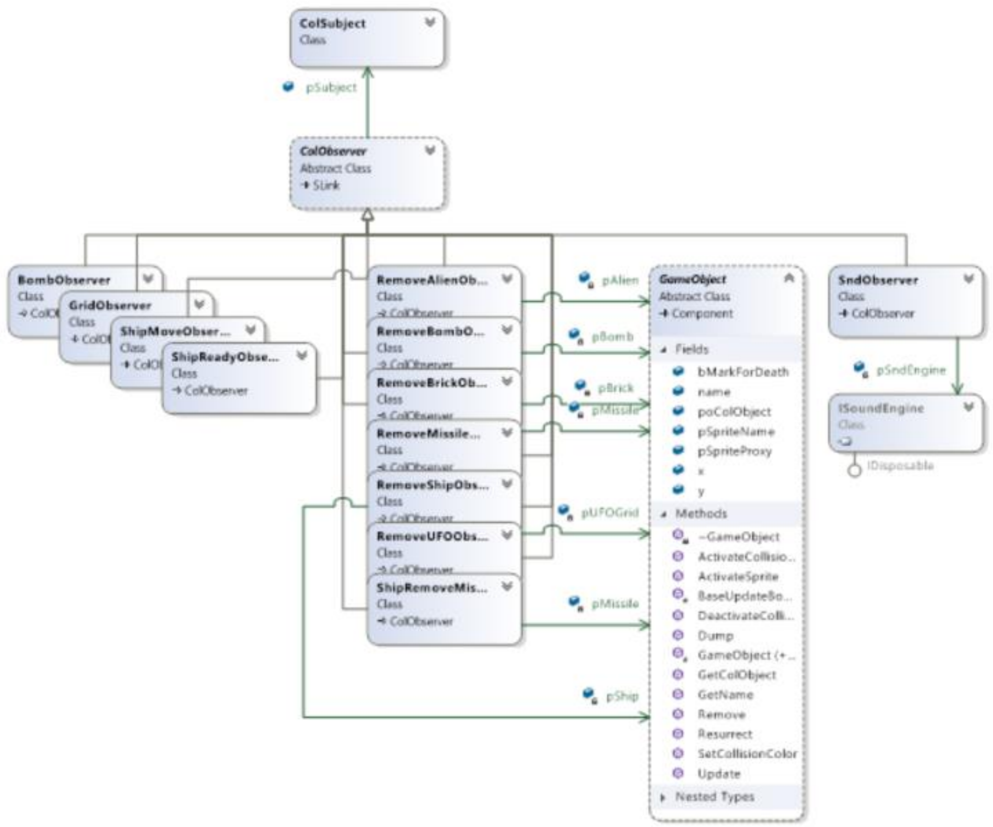
Mechanics:

Let's look at our animation commands. When we create an alien sprite, we also set up an animation command for each of them so that every x seconds, in our case every 1.0 float units at first, swap the image with another of a similar type. On the next call, swap the image with the original and repeat again. This loop will continue forever until the alien is killed.

What Does It Do For Space Invaders:

This system is primarily used to handle the killing of aliens, moving of the alien grid, and removal of bombs on collision with something. In essence, the event manager handles most visual alterations that happen in the game. However, it can be used for anything we need it to do. For example: the timer event manager is used on death to remove the player's ship and respawn it. This system helps make the game feel alive and reactive to the player alongside the visitor pattern.

Observer



Problem:

We need a pattern that can catch when collisions occur.

Solution:

We created an observer pattern to be attached to collision pairings to handle any collision reactions required.

Pattern:

The observer pattern is used in tandem with the visitor pattern. When a visitor has identified a collision, it notifies listeners attached to that particular collision pair. These observers could do anything they like. These are singular events that occur once per collision, but multiple observers can be attached to a single collision pair.

Mechanics:

An easy example of this would be the missile group colliding against the alien grid. As was mentioned above, the visitor pattern will do its thing. Then, when the missile game object properly collides with an alien game object, the visitor pattern will notify all listeners attached to that collision. So in this case, we would be calling RemoveMissileObserver(), ShipReadyObserver(), and RemoveAlienObserver().

Each of these observers handle different systems required for the game to continue. The remove missile observer will remove the missile from the screen. The ship ready observer will allow the ship to shoot again. Finally, the remove alien observer will remove the alien that was hit and replace it with a splat image for a moment before being removed via a command.

What Does It Do For Space Invaders:

The observer pattern gives the player the ability to properly kill all aliens to win the game, move, and shoot their gun. Without this system, the player has nothing to do but wait for the alien grid to reach the bottom and crash the game. I say crash the game because the observer pattern handles those collisions as well! This pattern is integral to making the game playable in any form.