Matt DiGianfilippo
www.dgdesigned.net

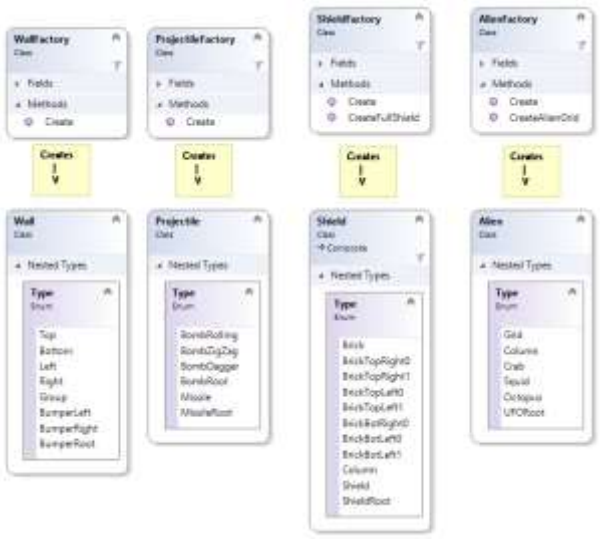
Design Pattern: Factory

Problem: The game will require creating game objects which have multiple parameters (position, scale, image, etc) and methods will have to be called after creation (like attaching the sprites to sprite batches). I need a simpler interface to create these objects and not have to worry about the rest.

Solution: Create factory classes which will do all of the tedious leg-work for me. They will be specific to the type of object and can only create those objects. After creation, they will run the after-creation methods and return the newly created object.

Pattern Characteristics:

- Creational Pattern
- Create objects without exposing the logic to create said object
- Simplify creation calls



The Factory pattern creates objects without showing all of the small details to the user. The pattern allows for easy creation through a similar method call for each factory.

The goal of a factory is to simplify the interface to create objects. The factory keeps all of the knowledge on how to actually create the object behind closed doors.

Each factory should specialize in its own type of objects and only create those types.

In Space Invaders, during the initialization stage, I had to create a lot of the same type of objects (aliens, walls, shields, etc). By creating a factory, I was able to simplify all of the instantiation logic. For example, I was able to call CreateAlienGrid() with the X and Y positions and the function returned a full alien grid ready for use. Within the Shield Factory, I created a CreateFullShield() function which returned a full shield. I could have also created a function to return a full set of shields but didn't feel the need for that in my game.

The job of my factories in my game was not only to create the game objects but also call the methods to attach the game object's game sprite and collision sprite to their respective factory. This helped simply my creation pattern. In the few places I didn't use a factory in my code (ex - UFO spawning), the code is too verbose. It should really be one call so that when

reading body of the command method, you don't get lost in the details of code that doesn't matter.

I was a little weary of using factories at first but once I thought about the practical use of this pattern with a multi-user development team, being able to compartmentalize each type of object into its own factory made a lot of sense. It allows multiple users to be working on their own part of the project without having to step on each other's toes compared to if the instantiation of all objects was in a single factory class.

Matt DiGianfilippo
www.dgdesigned.net

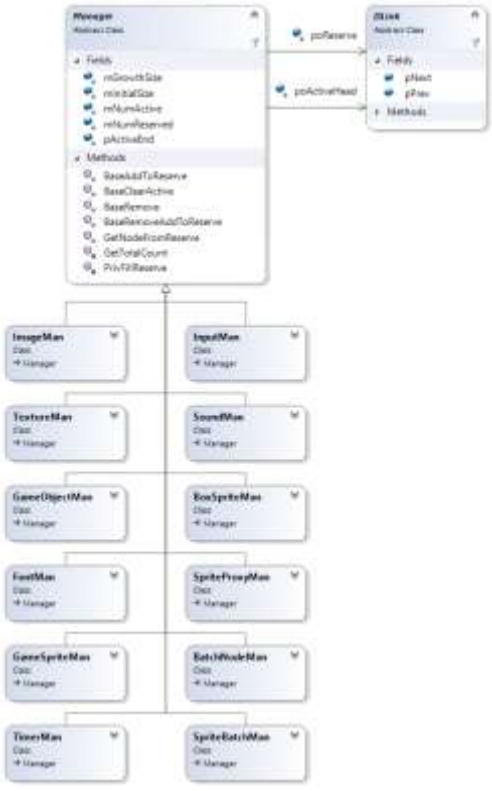
Design Pattern: Object Pool

Problem: New calls are expensive and we want to reduce these calls as much as possible while the game is running to create a smooth experience.

Solution: Create a large pool of objects and re-use them when we no longer need them rather than freeing them from memory

Pattern Characteristics:

- Creational Pattern
- Reduce new calls by reusing objects that are expensive to create
- Pooled objects are mutable and can only be used by one object at a time



Object pooling is the use of re-using objects that are expensive computationally to create using new. Rather than freeing the memory, the no-longer-needed object is rinsed of its instance data and reset back to factory settings. It is now ready to be re-used when another instance of that classes is needed.

This pattern is useful because it removes many new calls from the code. Instead of having news throughout the code, just ask the manager for an object and it'll give you a squeaky-clean object for you to use for its entirety. When you're done with it, just tell the Manager to recycle it.

Having created my own memory management system in the past, I've learned how costly calling new is in a real time system. It's much better to limit our new calls to as few as possible and during initialization and then re-use those objects over and over.

The ability to reset the objects to an uninitialized state also allows to ensure data isn't able to be accessed by clients that should not see this information.

In Space Invaders, all of my managers create an initial list of pooled objects on the reserved lists during construction. If we need another object but there is none left on the reserve, we create a group of them based on the growth rate.

While not related to pooling, these managers use static functions for adding/removing to their active/reserve lists contained in my DLink class. There are overloaded methods for

add/remove (head only and head/tail lists) which I thought would allow me to re-use the functions elsewhere. I did end up using them with some of my classes that manage basic lists but aren't full-fledge pooling managers which allowed me to not have to repeat methods I already wrote.

While many of my pooling managers were contained in their own scene instance, I had a few true Singleton managers. These managers included my Input, Texture, Image, SpriteBatch and Sprite managers. For my SpriteBatch Manager, it recycled sprite batches from the play and demo states that were no longer needed. These batches had varying amounts of sprites on the batches. For example, my debug sprite batch contained the collision sprites for every game object on the screen. On the other end, temporary graphics batch only had a few sprites on it at a time. When recycling these sprite batches, I recycled them in the opposite order of creation. This allowed for my states to always re-use the batches that were used for the same type of objects. This helped save a little bit on resource consumption as my program did not have to constantly keep running out of reserve nodes while the game was running.

Matt DiGianfilippo
www.dgdesigned.net

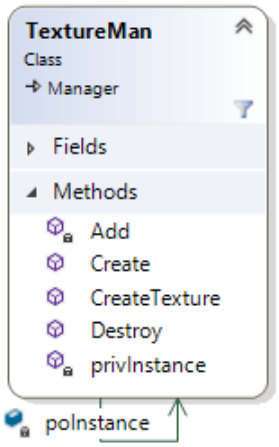
Design Pattern: Singleton

Problem: Sometimes we need to access data from anywhere and do not want to use global variables.

Solution: With singletons, we can access our data using a stance method from anywhere and guarantee there is only one of these instances within the program

Pattern Characteristics:

- One and only one instance
- Accessible from anywhere using static class methods



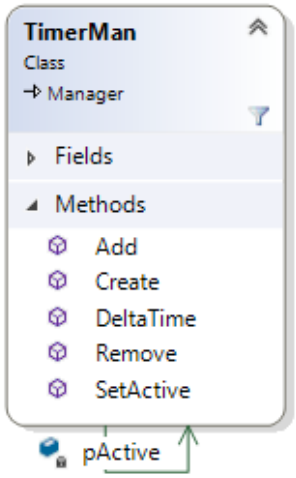
Singletons are classes which have only one instance and other classes have access to the single instance through static methods. With the static methods, the singleton can be accessed from anywhere without any reference to the instance. These static methods can either directly return the instance to call methods on it or have the instance abstracted away (not directly accessible to the public).

Singletons can either be instantiated lazily using just-in-time initialization, or a method like Create() must be called before being able to use the singleton for the first time. The constructors for singletons are made non-public so they cannot be called directly.

The Singleton pattern is pretty straight forward. Static class methods are available from anywhere. They belong to the class and are not specific to any instance of the class. This allows us to call any of the static methods from anywhere.

For my singletons, I created the singletons ahead of time using a Create() function which essentially initialized the singleton. I hid the true singleton instance away and never created a public method that returns the singleton. Because I only really ever had to use the singleton for a single method call, I felt like this was a good choice.

With all of our managers used throughout Space Invaders, we needed access to them from any class. Singletons did their job and allowed us to access the managers when needed. The usual warning about Singletons is to not overuse them. I was worried at the start of the project that we/I would fall into this issue. Looking back on the project now, I feel like we created just the right amount and only for the necessary reasons.



While the UML to the left is not, by the book, a Singleton - the idea of using a static switch function to change the "active" instance was in my eyes incredibly clever and I wanted to bring attention to it. The class did still contain a singleton, but it was used only to guarantee that the class was initialized before using and never used publicly. I had never considered the idea of switching the globally-available instance before. The static methods of Add/Remove/Find/etc. still allow us to access the needed data from anywhere, but we're able to switch that instance with the SetActive() method without the other objects knowing a switch occurred. From the viewpoint of the other objects, there is just a single instance that it is talking to.

This was incredibly important when using scene states and switching back and forth between them. The scene controlled which "static" instances should be active and allowed us to keep our managers within each scene contained within their own environment without any knowledge of the outside world. Also, not by normal singleton standards, the default constructor is made public as there are multiple instances of this class created (one for each scene).

In the case of the Timer Manager, seen in the UML, it allowed each scene to have its own scene time (the elapsed time of when that scene was active). Behind the scenes, each timer manager still kept track of what the real application time was (needed to calculate delta time between frames), but if an object ever needed the current time, it would only return the scene's time. This allowed me to keep each scene's time events separate from one another. It also allowed me to not have to re-iterate through each timed event item and update its trigger time when entering that scene.

Matt DiGianfilippo
www.dgdesigned.net

Design Pattern: Command

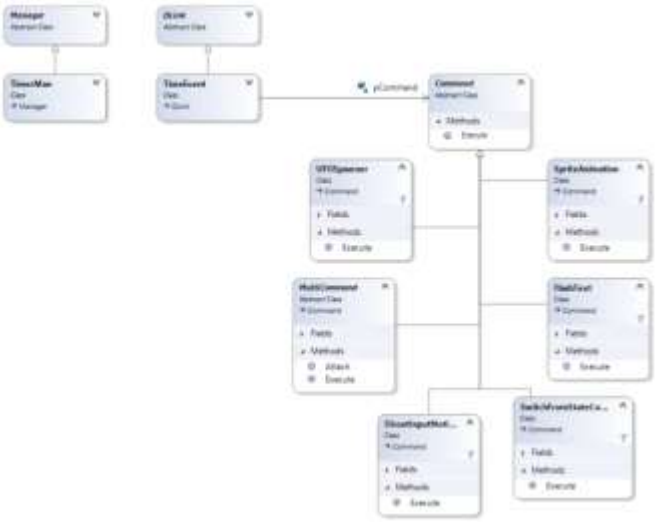
Problem: In Space Invaders, we need to have different events firing off at designated times. For example, the alien grid animates and moves at a specific interval, it is not a fluid movement.

Solution: Create a timer manager that manages time and allows me to maintain a list of commands that will fire off when they are set to be called

Pattern Characteristics:

- An action that is encapsulated as an object
- Can be added to a list or handed around with references

The command pattern is a way to encapsulate an event or action into a single method call. These calls could have an undo function to revert their changes but they don't have to. That's it. The whole point of this pattern is really for any object to hold a reference to a command.



Within Space Invaders, I used the command pattern for my timed events. Each TimeEvent had a time to trigger and a reference to a command. When enough time had elapsed in the scene, the Timer Manager would trigger the TimeEvent's command and remove it from the list.

Some examples of use-scenario were for moving the alien grid, animating the aliens, and spawning UFOs and bombs.

With my command pattern, I also created a multi-commands class. These commands contained a linked list of mini-commands (which in essence were just DLinks with a pointer to a command). I found this useful for the alien grid as multiple events had to occur at once. I also had to re-add these events to list based on the current speed of the alien grid.

I had a bit of love/hate with this pattern. For the simple commands, this pattern worked great. I found myself needing to create complicated actions like delayed switching of states. I required all of my state switching to occur after all events in my Delayed Manager. This required creating a TimeEvent pointing to a command. This command would then add an item onto the Delayed Manager which then needed its own command to actually switch the state. Postmortem, I wish I changed the functionality of this to be more streamlined and generalized so that I didn't have so many concrete commands.

Design Pattern: Composite

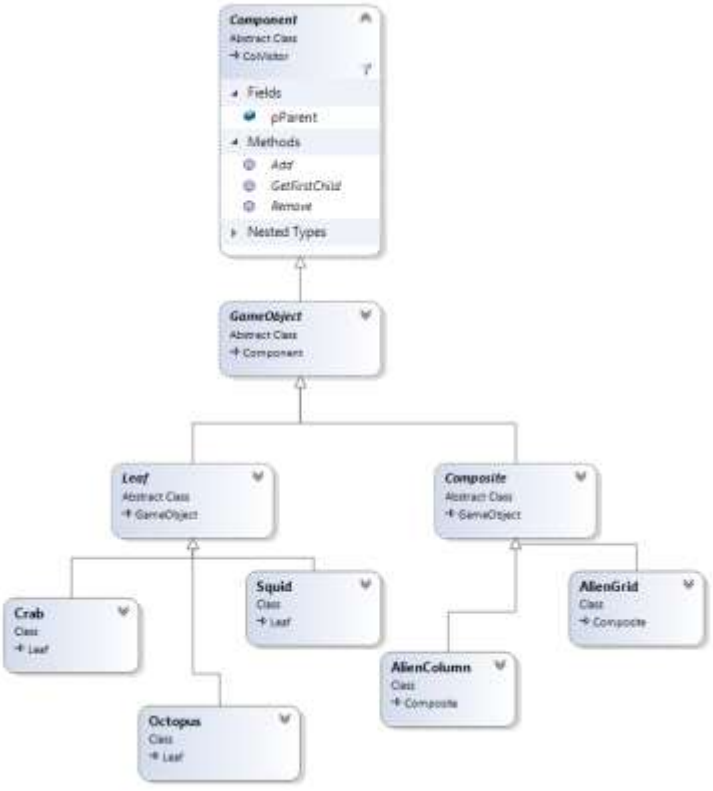
Problem: We need to organize our game objects in a hierarchy to optimize performance with collision detection and create a structural relationship of groupings

Solution: Using the composite pattern, we can create a tree that allows us to iterate through all objects, create a hierarchy of objects and create structure to our list of game objects while treating all game objects equally

Pattern Characteristics:

- Create a hierarchal structure using trees
- Every composite can contain many leaves and/or composites
- Allows for iteration over the entire structure
- Allows the ability to treat each component equally

The composite pattern is a structural pattern that allows the program to treat all objects on the tree uniformly. Each object is derived from the component class which contains functions for both leaves and composites. Only composites can have children (a list of components). These children, due to inheritance, can be either composites themselves or leaves.



Each composite object has a hold of its own list of components. The component doesn't know who they are, but it has a list of them. Leaves do not have any children.

The main intent of the composite pattern is to create order and semblance to what could otherwise be a chaotic structure with many different objects. It also allows us to treat each object in the tree the same. No edge cases. The reason that we can treat every composite and leaf the same is they all derive from the same class "Component." This means that even though leaves can't have children, it still needs to implement the method.

The composite pattern could be thought of as the folders within a hard drive or operating system. We have the main composite object (the root path). Within that, we could have many folders or files. The folders would be composites with their own set of folders and files. Even the most complicated and ugly file systems (I'm looking at anyone who puts all of their files on their desktop) can be shown in an organized way. In this example, the files would be leaves as they cannot have any children.

Normally with this pattern, the composite objects are simply containers. In Space Invaders, we actually wanted to act upon these composite objects. The composites needed to update their bounding box for collision detection. By using the composite's collision box, we could optimize our collision checks without having to iterate through every child each time.

When a function (such as Render()) was called, within its own Render() function, it would loop through its list of components and Render any children. Since many of our composites contained other composites (ShieldRoot and AlienGrid, for example), this would allow us to iterate over every object in the list just by calling Render() on the root composite.

With the composite pattern, I could create one event to move the grid and, iteratively, I could move the entire alien grid as one cohesive unit. The alien grid composite object would control the direction the aliens would move. This delta would change when it collided with a wall. Now instead of traveling right, the entire unit would travel left (after having moved down due to the collision).

One method I found helpful with the composite pattern was that during removal, I would let the child object notify their parent they are being removed. This allowed me to generalize composite removal and let the class decide what to do. By default, the composite object would do nothing. Each composite object could override this and handle the situation differently. With my shield, my trees were four objects deep while my alien grid was only three. Instead of creating specialized removal commands, I let the composite object decide what it should do when it had no children left. The roots always stayed on the Game Object Manager while the other composite objects would remove themselves if there were no children left.

Matt DiGianfilippo
www.dgdesigned.net

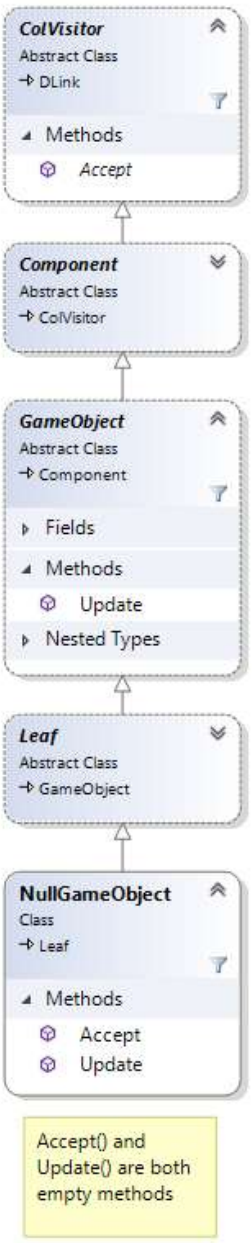
Design Pattern: Null Object

Problem: Sometimes we don't want our game objects to do or have certain characteristics. We need a way to remove these edge-case scenarios

Solution: Using null objects, we were able to call Render() on our composite game objects without actually rendering them

Pattern Characteristics:

- Get rid of edge cases
- Allow for "nothing" to happen when called



The null object replaces the need to check if a reference is null. Calling a method on a null value will result in a crash of the program. Rather than needing to check if the reference is null, we can instead call the methods with the null object. These methods will do nothing as they have no body. The null object sits there as a placeholder for a null or nullptr value while removing the conditional check to see if the value is null.

The main intent for the null object is to get rid of the possibility of a reference or pointer being null. By eliminating this possibility, we can remove the conditional check before acting upon the reference.

While I only used the NullGameObject displayed in the UML to the left for comparison checks, I did use null game sprites throughout the game for composite objects (alien grid/column, shield root/column, etc.). These null object game sprites acted the same as a normal game sprite except that the width and height of the null game sprite were 0. Because these values were 0, they were never drawn. This had the same result as a "normal" null object. This version of a null object is hard to document in a UML as the variable values are what makes it a null object.

Using the null object with our composites, I was able to have an object in the reference location for a game sprite without having to worry about the edge-case of not rendering an image. This allowed me to still treat these game objects as a regular game object, one with a game sprite and a collision sprite, without any conditional checks.

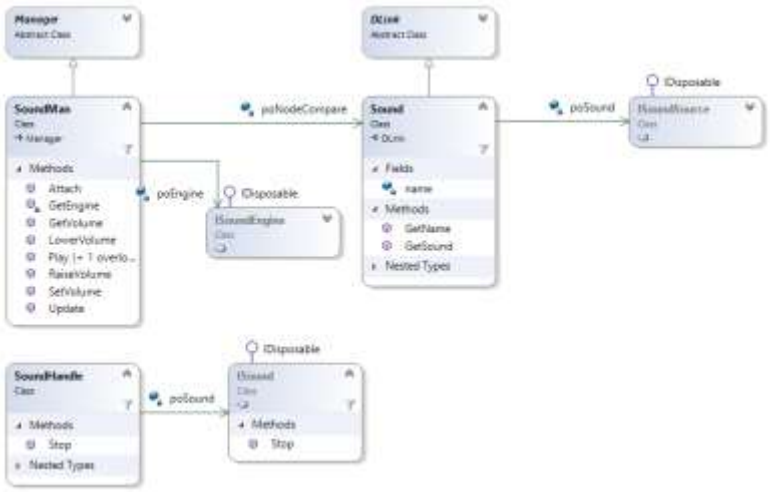
Design Pattern: Adapter

Problem: When creating Space Invaders, we had to use two external libraries: irrKlang (Audio Engine) and Azul (Game Engine). With Azul, all of the variables and functions for its classes were public. We would need to block access to the public class variables either with setters and getters or blocking them entirely. The audio engine for irrKlang had many features, most of which we didn't need to deal with for this project. We needed to simplify the interface for it and hide them so our game would be less library dependent in our code.

Solution: Created classes that would adapt the library specific objects into classes we can use with our managers using the Adapter pattern. This simplified my interface, abstracted away the low-level data.

Pattern Characteristics:

- Adapt one interface to another
- Encapsulate other classes into your own class to contain the dependency for the future



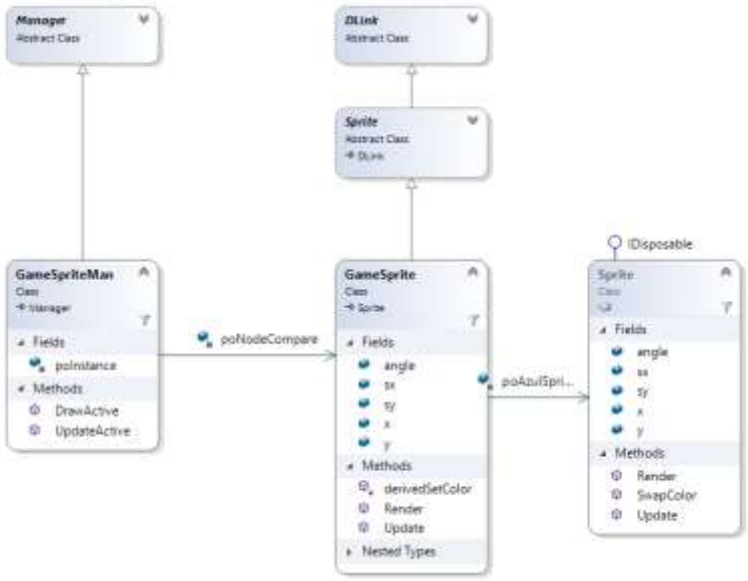
The adapter pattern bridges an interface between two objects. It allows us to let classes work together that normally wouldn't. This includes the ability to hide the internal structure of an external library in case that library changes in the future (or the external library is changed with a different library). This pattern helps minimize the time having to update classes and methods when a lower level structure has changed. In large programs and systems, this would be a mess. The adapter pattern also acts as a middle-man where I can limit or block access to any variables or functions that may be public in the library.

I find the adapter and proxy patterns to be similar as they are used for similar reasons; to act as an intermediary between the real object. The adapter pattern solves my issue by hiding the true data between my class that controls what my program wants to be done. I do not have to worry about what type of class is internally being used because my Sound adapter

holds the reference to the external library's object. If any changes were made to the library in terms of API or using a different library, I would be able to change my adapter without haven't to make any changes to other code. The adapter pattern also allows me to block access to any public variables or functions that the external library has that I do no want to expose in my game.

Within Space Invaders, sounds effects were a must. The alien sound, explosions, missile firing, and the UFO's sound all needed to be replicated. I also wanted to be able to change the volume of the game without changing the system volume. My Sound class adapted the library's internal objects into something I could use. When I wanted to play a sound, I would call SoundManager.Sound(). This method was overloaded to either take the enum (if I did not have a handle to the sound) or my sound adapter. I could then focus most of my time on the other parts of the game.

Another sub-issue I had was for looping sounds. The audio engine returns a handle to the currently playing sound. In order to hide the external library, I created another class that acted as a wrapper containing a reference to this object before returning it back to my game. Its only purpose is to be able to call the Stop() method to stop the sound from looping.



With the Azul engine, to render anything I had to use Azul's Sprite, BoxSprite, and SolidBoxSprite classes. The Azul engine makes all of the class variables public. In order to hide them away behind a wall, I created adapters GameSprite and BoxSprite. This allowed me to ensure the instance values were updated through setters and getters (or not at all).

Matt DiGianfilippo
www.dgdesigned.net

Design Pattern: Proxy

Problem: Many repeated images will be drawn on the screen at once. We will need a way to keep the memory footprint down while rendering all of these objects

Solution: With the proxy pattern, we can create more instances of a lighter object that push the instance data to the full, heavier object.

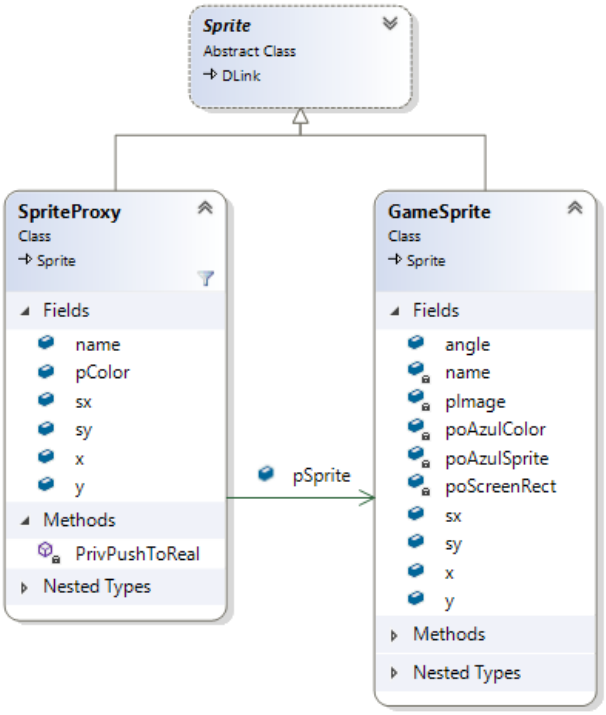
Pattern Characteristics:

- Use a placeholder object in place of the real object
- Lightweight proxy points to the heavy fuller object
- Acts as a protective layer between the client and the real object

Proxies are, as the name applies, placeholders for other objects that later push the values to the real object. In my opinion, they are very similar to adapters. Both design patterns introduce a layer of abstraction. Proxies, like adapters, can be used to restrict access to certain variables and method calls.

Basically, the pattern is about creating a middle-man between your client and your real object. That's it. You can do whatever you want to the middle-man but only he can choose what data/method calls are to be sent to the real object and when. This allows us to keep a better handle of when we are actually modifying our real objects.

A real-world example would be a body guard. He's always watching and if you ever want to talk to the important person, you have to talk to the body guard first.



While they don't have to be, in our scenario our proxies were lightweight versions of their fuller counterparts. The proxies only included variables for values we would need to change like scale, location, and color of the sprite.

We would create many proxies and just a few actual real objects. This helped keep our memory footprint down.

One extra benefit is that we don't actually push the data to the real object until the moment we need to draw it. When it is time to render the sprite, we push the data to the real object, overwriting any previous values. Because of this, we are using the real game sprites like a flyweight; all of our proxies are using the same full objects to render themselves.

One interesting side-effect of the proxy pattern is that we could have another object *change* what sprite the proxy was pointing to. This idea allowed us to create animations.

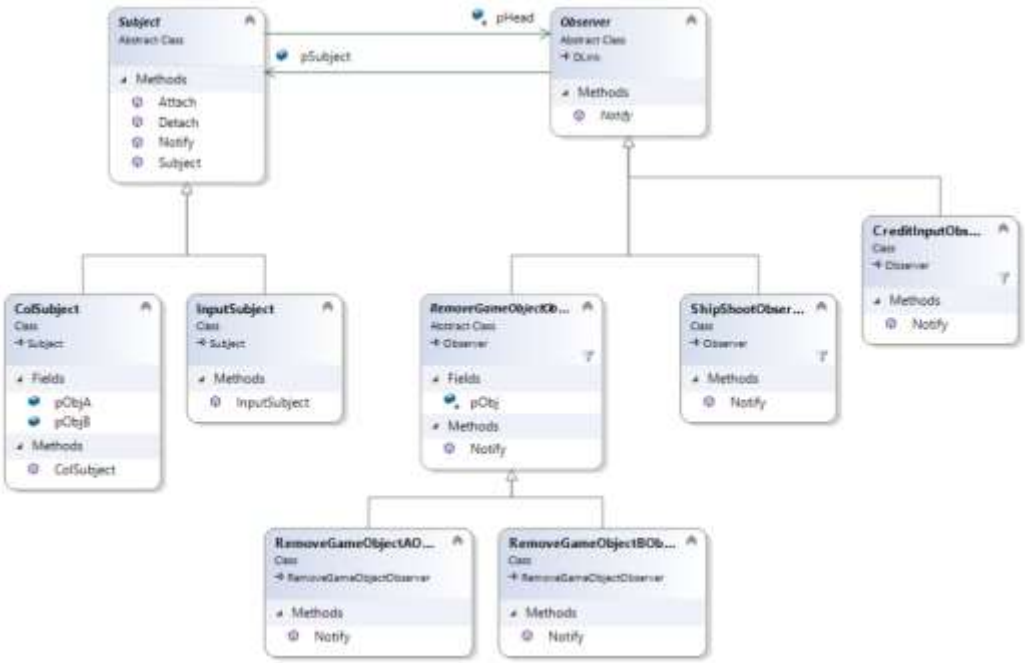
Design Pattern: Observer

Problem: Within the game, actions will need to occur when certain events happen without constantly checking if an event has occurred. We don't know when the events will occur and we don't need the event to know who cares about the event. Just let everyone who wants to know be notified

Solution: Use the observer pattern which allows any actions that should care about an event to attach themselves to a list that will be notified when the event happens. The event doesn't know who is on its list, but whoever is, will be notified that it happened.

Pattern Characteristics:

- One to many relationship
- Ability to notify any object that cares when certain events occur without needing to know who those objects are
- Encapsulates an action as an object



The Observer pattern allows for a number of objects to be notified when an event occurs without the notifying subject needing to know who it's notifying. The subject keeps a list of observers that wish to be notified of the event. When the event occurs, Subject.Notify() is called which goes through its list and calls Notify() on each observer object. Any observer that wishes to be notified, just needs to attach themselves to the subject.

The pattern is basically a necessity in any real-time environment. Anything that involves interactions needs the observer pattern. This includes key inputs, collisions, updating values on the screen and more.

With Space Invaders, we largely used this pattern for collisions and inputs. When a game object collided with another, it would populate which two game objects collided and notify anyone that cared. Even with a relatively simple game (in today's terms) in Space Invaders, I was stacking as many as 8 observers on an event.

With my naming scheme for colliding objects (alphabetically sorted), I found myself needing to sometimes remove the first object, sometimes remove the second object, and sometimes remove both. Originally, I started writing "pair" collision observers and found that to be both messy and a hassle. I then came up with, in my mind, a great idea of creating an abstract function that contained the meat of the Notify() code. The derived classes would then just change which object the executed code was being operated on (and then call the base.Notify()). With the object reference filled in, the base Notify() method would act upon the correct object.

For example, I created a RemoveGameObject class which would mark a game object to be removed and then add it to the remove list. Derived from that were two classes: one which set the object to ObjectA and the other to ObjectB. Once I made this change to my observer-creation style, using this pattern was a breeze and almost automated. I could add either (or both) removals to a single subject no matter which one came first in the alphabet.

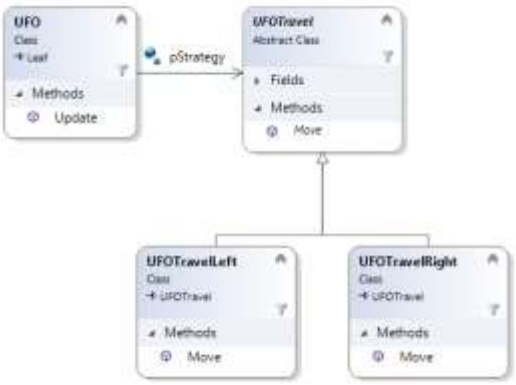
There were many other scenarios where I needed to use the observer. Inputs for the ship, knowing when the alien grid was empty and needed to be reset for the next level, when the player's ship was destroyed and had to switch states, and many more.

I also used the Observer pattern in order to find out when the current wave of Aliens have been cleared. My Alien Grid class contained a subject which I attached a "Next Level" observer to. Every time an Alien Column was removed, it would notify the grid about this. The Alien Grid would check to see if any children were left and if there were none, it would know it was time to let the observer know that we need to create the next wave (and prepare for it).

Design Pattern: Strategy

Problem: Game objects within the game will need different behaviors. The UFO has to be able to start from the left and move right and also start from the right and move left. We have three different types of bombs but don't want to create a lot of derived classes off of bomb.

Solution: Using the strategy pattern, we can change how the UFO and bombs internally work without creating multiple derived classes.

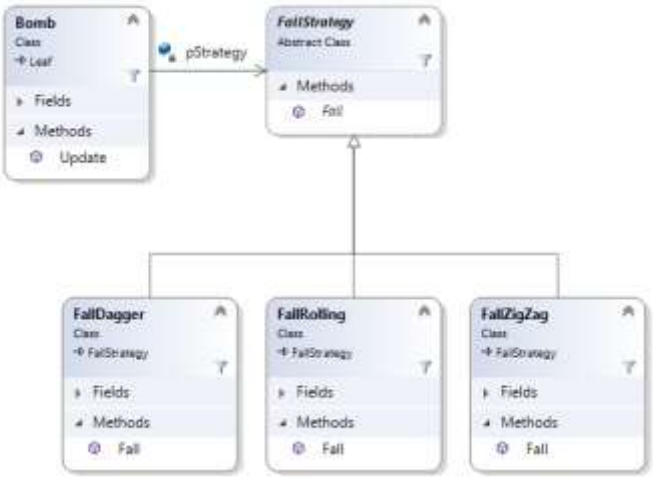


Pattern Characteristics:

- Change the internal behavior of an object permanently
- Alternative to class inheritance
- Expandable without affecting the main object's class

The strategy pattern allows different specialized behavior without affecting the main class. This is important as this means the strategy pattern is scalable without making any changes to the main class.

The strategy pattern is similar to the state pattern except that the strategy never changes. Once an object has been initialized with the strategy, it will not change its behavior. Rather than deriving new classes from the main class, new behaviors are created with the strategy.



Within Space Invaders, our UFO needed two different behaviors. Rather than creating a UFOLeft and UFORight class, by using the strategy pattern, we could change the update behavior of the UFO. One strategy would start on the left side of the screen and move right. The other would start on the right and move left across the screen.

Expanding this idea with our bombs, we were able to use completely different sprites as well. The Dagger bomb would use a cross-style sprite and flip vertically over itself. The Zig-Zag bomb would use a lightning-shaped sprite and flip its image horizontally. The rolling bomb did nothing except fall straight down. To the gamer, these all look like completely different bombs but they are in fact from the same class.

I want to try to use this pattern more often than creating new derived classes. For Space Invaders, there were only a few instances where it made sense but I'm hoping in the future with more complex games, it will come in handy.

For example, I feel like this pattern will be useful in future games that will involve many small items and/or powerups. Instead of constantly deriving from a single class, just change the behavior within the object. Temporary (or permanent) attributes could be applied using different strategies without having to derive new classes from the main item class.

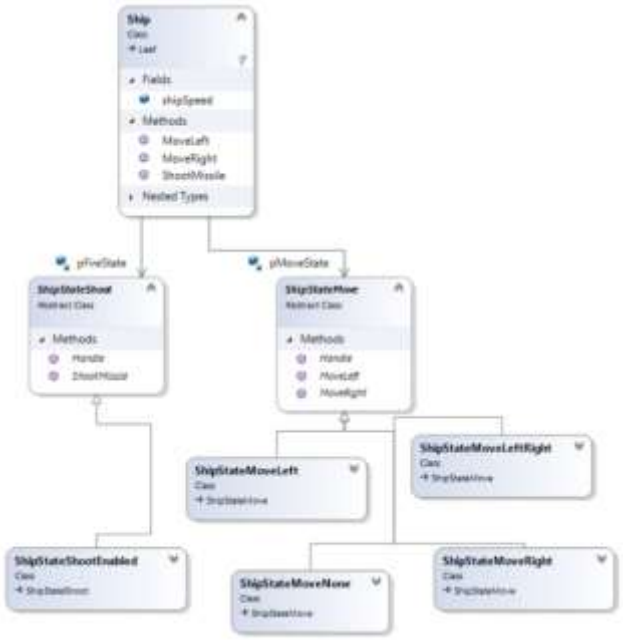
Design Pattern: State

Problem: Some of my classes will have certain conditions where their behavior will change. We need to allow these conditions to be met while reducing the use of conditional statements as much as possible.

Solution: Create abstract states for any classes that need them. The derived states will override the abstract methods and allow the instances to change their internal behavior without any conditional statements

Pattern Characteristics:

- Change the internal behavior of an object temporarily
- Remove conditional statements
- Encapsulate what the object can do at that moment



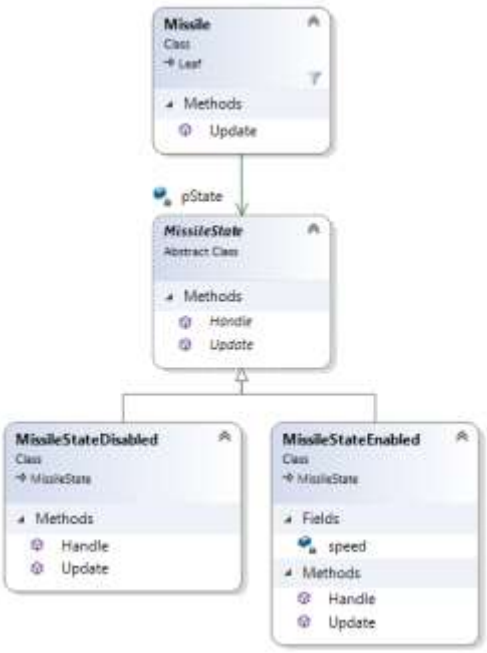
The state pattern allows us to change the behavior of an instance by changing its internal state without any conditionals.

Just like the strategy pattern, the state pattern allows us to completely change how an object behaves. The only difference is we can keep changing this behavior whenever we want.

These changes of states are normally handled with a Handle() function but I found this not to work for more complicated scenarios. For example, when our ship collided with a bumper, which state was it supposed to switch to? The ship didn't know who it collided with, only that it collided. The observer that was watching the event knew though and that observer, in my opinion, should be the one to make the decision. I felt like the better option was to allow the observer of the collision decide for the object.

This goes slightly against the rules of the state pattern but I found during development that there are a couple of options: introduce conditionals or allow an external object to change an object's state. As long as you're careful with your code, I feel like the second option is okay.

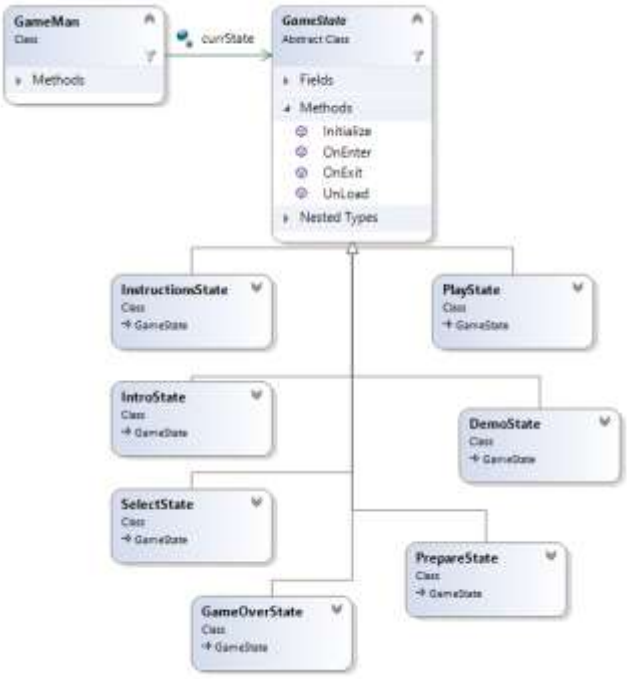
While our entire game is running in Debug mode, if we could remove as many possible logic branches in our code, we could allow the compiler to really do its magic with the Release code. This would really make our code fly. States can help us with this.



With Space Invaders, the most obvious use case for the state pattern is our ship. We need it to move left and right only within a designated area and be able to only fire a missile one at a time. When the ship collides with one of the two bumpers in the game, the ship will change its state to only move in the opposite direction.

With the ship shooting a missile example, we had two states. One's ShootMissile() function actually called for the missile to be fired while the other's method body was completely empty. It didn't matter how many times I pressed space, that missile would not fire unless the ship was in the ShootEnabled state.

In order to recycle the missile efficiently, we also needed to implement an active/disabled state for the missile. When it's enabled, it'll fly upwards and be able to collide with any objects. Rather than constantly adding and removing it from the missile's root, we could just move its position to a corner off of the screen. When disabled, the update method will do nothing as it has no body.



Our entire game is also using states to control which screen is currently active. All of the other screens are in a "frozen" state where nothing is updating and nothing is drawing to the screen. Within my game I have multiple screens: Instructions, Introduction, Select, Play, Demo, Game Over, and a transition screen. By using states, I was able to encapsulate each scene into its own state. Every scene required its own clean up and preparation methods to be called when entering and leaving the state.

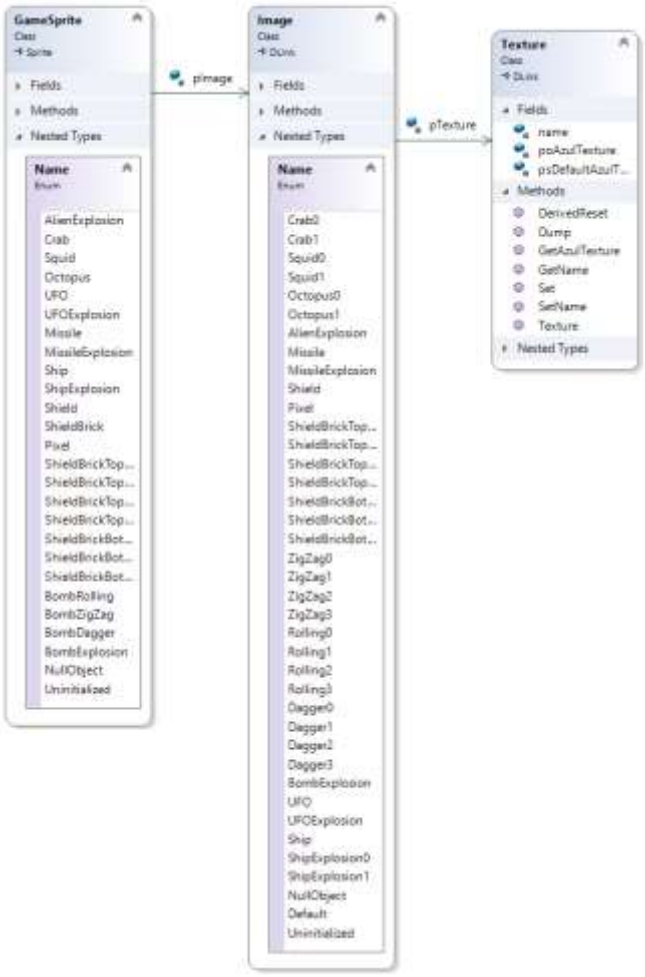
Design Pattern: Flyweight

Problem: I need to re-use the same sprite sheets over and over as they contain multiple images

Solution: Use flyweights as a read-only object where I can point to different parts of the sprite sheet

Pattern Characteristics:

- Object sharing between many different objects
- Reduce duplication of the same objects
- Flyweight objects can be created lazily or up front



Flyweights are objects that can be used my multiple other objects at the same time. These shared objects can either be created lazily whenever needed or be pre-populated ahead of time. The whole goal for this pattern is to share an object instead of creating many of the same objects.

A textbook example of flyweights is for a type system; the letter characteristics will never change and can be re-used by any object that needs to draw characters to the screen.

The gist of this pattern is that if you're going to be using the same object over and over again, share it around. Don't be a resource hog and create multiple copies of the same exact thing.

The flyweights can either be only created when needed, which would save on memory size, or be created up front. Within Space Invaders, we went with the method of creating them all at once up front during the initialization process of the game.

In Space Invaders, we used flyweights for our font systems. We also used flyweights with our textures and even our images. Our sprites also could be considered flyweights.

When loading the texture into the game, our textures contained many different images. Our images then pointed to specific coordinates on the texture along with the width and height of the image. No modifications were ever made to the texture or images. They could be thought of as “read only.”

With our sprites, we were sharing them between many different sprite proxies. One difference though is that were changing instance variables on the sprite but that was okay as every proxy wrote their own data to the sprite before rendering it. For example, we would change the position of the sprite as well as the color.

By using a combination of flyweights and proxies, were able to implement a design that allowed use to re-use and share resources in order to minimize the memory footprint.

Postmortem

Overall, I'm happy with my homage to Space Invaders. There are always areas where I could have improved and made a better product, but that's a never-ending story. This was also the first game I've finished to completion and never abandoned mid-development.

The key to using design patterns is to keep as high of a level in your classes as you can when writing the methods and patterns. Only until you absolutely have to, then go to the concrete classes. This idea didn't click for me until the last week of the project; I had always wanted to get into gritty details of what the object is supposed to be doing without looking at the generalized idea. This thought process will help me with future program development.

While I used the pattern in a few places, I wish I had implemented my iterators sooner. By the time that we started talking about composite objects, I had a large amount of methods that looped through lists using regular while loop iteration. The few places where I did implement them, they helped out tremendously. For example, when removing my shields (in order to create new ones), I created a simple loop for each child. The loop would grab the next sibling and call remove on that. With composites, these method calls worked iteratively until we remove all of shield bricks. Maybe once I have some free time, I'll go back through and make these changes. That and work on making the reverse iterator a true reverse iterator…

The observer pattern was a savior in this game. As I mentioned in the pattern section, I originally was creating paired event observers (ex – RemoveAlienRemoveMissile) due to the fact that sometimes the object was in the A slot and other times it was in B. Once I came up with the idea of abstracting the removal and then having the derived classes choose which object in the subject should be removed, it was a lot simpler. I could have even gone a step further and created strategies for this. By using a strategy, I could have had an A and B strategy and re-use them for every observer that had to apply events to either object. This would have cut down on my number of derived classes I created.

One issue I ran into with the observer though was creating a delayed scene state change. Keeping consistent with the real Space Invaders, there was a delay between switching players or getting a new life. This required adding a delay to the switching of scenes. In order to do this, I had to create an observer who watched over the ship collisions. When the event would trigger, it would then add a delayed command to the Timer Manager. That delayed command would then add a command to the Late Update Manager to switch states. That required a lot of classes in order to complete the "simple" action of switching a scene. I couldn't have the state switch on the Delayed Manager because a stray object may need to have been removed after the Switch State command was added to the Delayed Manager.

My idea of using DLink to contain all of my Add/Remove static functions worked out great. Because I had to manage lists in places other than the pooling managers, I could re-use

these methods easily and know they were already working correctly. This reduced rewriting methods all over my program.

The composite pattern combined with the visitor pattern was excellent for collision detection. Our Collision Pair Manager only had to check about 11 collisions each loop. Many of these collision pairs never had their roots colliding until a leaf object was actually colliding so we had an easy out with many of them. The visitor pattern allowed me to iterate in either direction when checking a collision. I always chose whichever composite at worst would have fewer children. One issue I ran into during my shield stress test was that I had to make sure I was iterating the children equally. By this, I mean once I was in the column of a composite, I should also be comparing columns to columns. This was noticeable when my stress test was only removing one brick when many missiles were colliding with different parts of the shield.

I wasn't happy with my randomized bomb method. I created three different timer events (one for each type of bomb) and would randomly spawn a bomb. This part of my method worked great, but I had no way of knowing which alien column had already recently fired a bomb. This was evident when there were only a few columns left when bombs were spawning from the same column on right on top of each other.

A couple of small glitches that bothered me was sometimes the Aliens' animation cycle would repeat when entering a new scene. This was caused by the previous state having just cycled to that specific sprite. With our animations only being two frames, the likelihood of this occurring was pretty common but hopefully not too noticeable. One solution was to create a separate Image Manager for each scene but I felt like that was too heavy handed. Another glitch was my alien collision with the wall. The alien's movement was on a timer while collision detection occurred every loop. I wanted to avoid conditionals, so I went with a non-standard move of moving the grid down and then across in a single cycle. The part I mostly dislike is it breaks the consistent flow of the alien grid.

I absolutely despise my sprite batch groups I created for each state. This would be the first thing I would change with my project. Rather than having a single sprite batch manager which each scene added/removed their batches to, I think I would have preferred having a common sprite batch manager and a bunch of scene sprite batch manager. The only sprite batch I was really sharing, which required the unified batch manager, was the text-based graphics. This could have been drawn onto the screen after the scene batch manager ran. This would have simplified my game object creation. In my version, I had to implement a sprite batch group strategy per scene. This over-complicated the process of adding/removing anything from the screen during gameplay. It also affected my use of factories as I had to find the correct batch to add the sprites to per scene. My non-playable states also re-used the same batches for all of the objects on screen in order to try to reduce the number of sprite batches but it only created edge-cases.