Space Invaders

Architecture Design Document

The purpose of this document is to explain the reasoning and use of various design patterns to create the data driven architecture that was implemented in the Space Invaders Project.

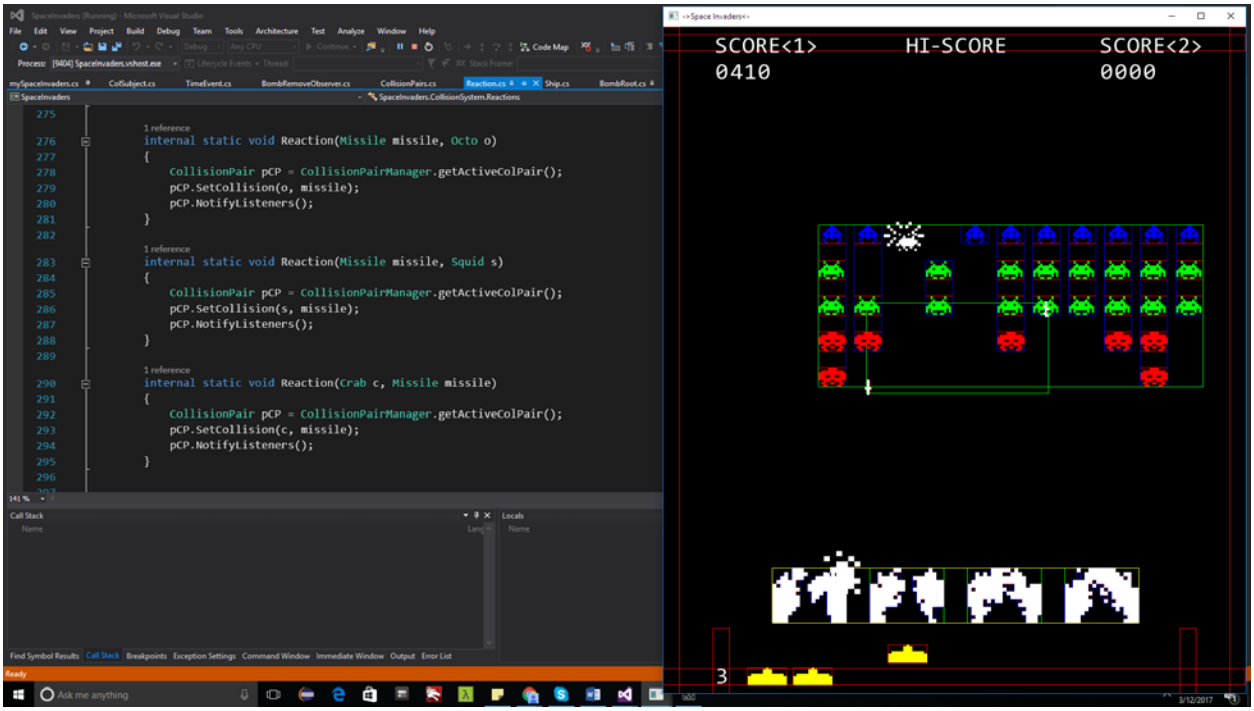


Table of Contents

The Game Layout: Using the Azul Engine you create a class that inherits the Azul.Game abstract class that contains 4 abstract methods you need to implement to make a functional game.

Initialize () : Allows the engine to perform any initialization it needs to before starting to run, Called only once It is the first method in the game that is called and it allows you to load any non-graphic content as it is called before openGL is started, here I set the game window size, name and background color.

LoadContent() : Allows you to load all of the additional content needed for your game to run, called only once it is the second method in the game that is called and it is called after openGL is started so you can load up any graphical content and any objects that are required by the game. I use this to do most of the setup for my game so I can save time later

Update() : Allows you to update data, process transformations and input, animation etc. Called once per frame this is where a bulk of the processing for the game occurs.

Render() : This is where you draw graphics to the screen, called once per frame, used for only rendering the graphics of the game.

With only 4 methods I can use in the engine it presents my first problem immediately.

Problem 1 – Organization

Who holds all the games data?

What oversees an objects lifetime?

Where does my data live?

With 100's of objects needed for gameplay where do all the references go?

When an object is no longer needed, what happens to it?

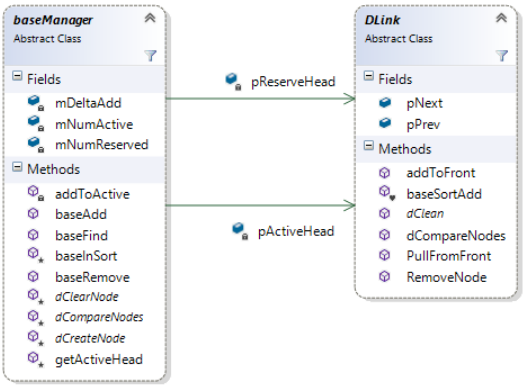
The solution – Object Managers

Managers can hold and organize data, and avoid hardcoding data elements. Having managers be singletons allow for easy access anywhere in the program where they may be needed and I don't have to maintain multiple lists of one type of object. This allows me to let the managers do the a majority of the work handling the game objects allowing me to focus more time on game behavior and less time worrying about book keeping. As a bonus the managers allow early creation getting a vast majority of the calls to new out of the way in the LoadContent() phase saving precious processor time during the game loop and since they are also object pools If necessary I have a few extra of each object always standing by. I also avoid time in garbage collection and additional new calls down the line because as objects are destroyed in game, the resources are not released back to the system, instead they are put into reserve so say at the end of a wave of aliens the aliens can be simply revived from the reserve pools bypassing the complex construction of higher level game objects.

This also allows me an additional avenue of optimization down the road, I can track the stats of object creation during development if I am running into situations where I am not creating enough reserve Sprites I can tweak the data, from the default 5 starting objects to what the test runs are showing me the game uses on an average run. I can also tweak how many extra are created if the reserve pool were to run out and additional ones are needed, currently default of 3 extra are created if the reserve pool is empty and an additional object is needed. Easily tuning the performance of the game by only changing the value of the variables passed to the manager's constructor. Which could be done dynamically by saving the stats from a file and by tracking additional new calls during the update loop so the file can be updated, until the optimal numbers are reached. So, the game optimizes itself the more a user plays it.

Easier said than done, right? With the simple application of a few design patterns this comes together quickly. Using the Template Pattern, Singleton Pattern and Object Pool Pattern's I lay the foundation for the rest of the game.

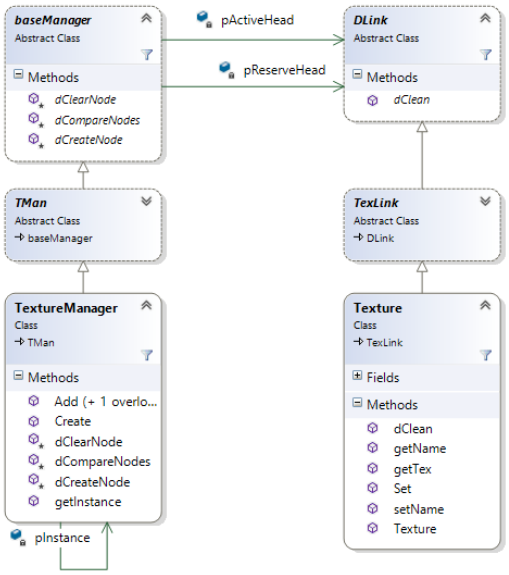
So first we lay the foundation for the manager system, it consists of the baseManager that provides most of the functionality for handling the lists of objects so we don't have to rewrite this code for every manager, and the DLink class contains ALL the code for a doubly linked list system to completely isolate the linked list implementation from our game objects. So we decouple the list logic from the game logic. The game objects never know that they are part of a collection at all. The act of defining an algorthims structure and relying on a concrete class to handle the particulars of a task brings us to our first design pattern. The Template Pattern.



Base classes for the Object managers

{Template Pattern}

The Template pattern defines the skeleton of an algorithm differing some steps to subclasses. The template methods let the subclasses redefine certain steps of an algorithm without changing the algorithms structure.



Texture Manager and Texture Object
Note: TMan,TexLink and other classes that derive from baseManager and DLink that lie between concrete classes are abstract classes that do nothing mechanically but are used to generate clear diagrams as you will see shortly.

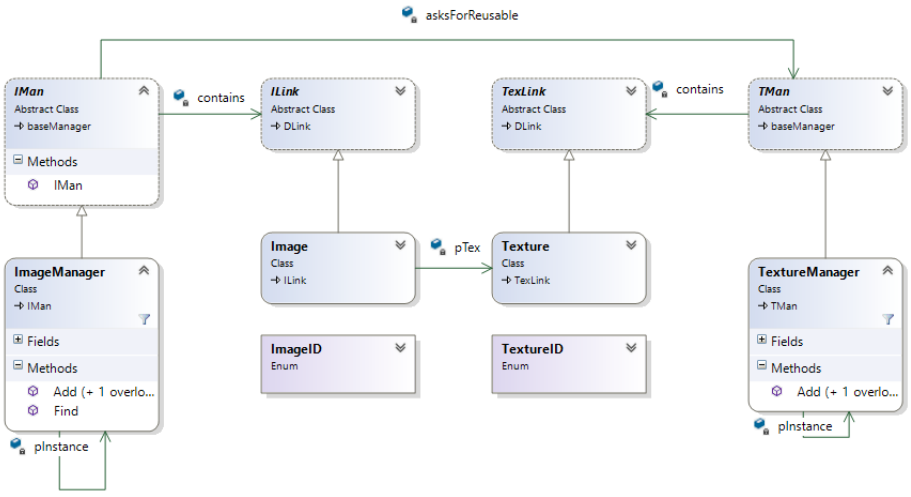
You can think of a template pattern like a team in real life. The abstract class is paired with a concrete class. Instructions are given to the team and the team figures out how to accomplish the task.

The template pattern in software acts much like a team and is a pattern that many new programmers will likely be taught before they are taught anything about design patterns. Anytime they create an abstract class with abstract methods and have a concrete class override those abstract methods, they are using the template pattern. Though an important step to note that many beginner programmers are likely to overlook is to refactor the code once a problem is understood, moving shared code from the concrete classes into the abstract class to avoid code duplication. This allows the team to work together, the abstract class executing shared functionality and the concrete implementation handling only the specifics of a given task.

In my Space Invaders, I make use of the template pattern numerous times because it is so closely related to the fundamentals of object orientated programming. Being fundamental it is worth noting the way that it is applied to the object managers of the game. Most of the work of managing an object is done in the base manager abstract class. The concrete Manager implements three important methods. dCreateNode() which implements the creation of the concrete DLink derived class so the reserve list can be populated on load. dCompareNodes(DLink , DLink) which implements the comparison of concrete DLink classes. And dClearNode(DLink) that calls the dClean() method in the concrete link class passed to it before reuse to ensure no stale data is introduced into the simulation. The concept of reusing an existing object leads us nicely into the next pattern the ObjectPool Pattern.

{Object Pool}

An Object Pool manages an objects lifetime. Allowing the reuse and sharing of expensive objects.



The Image Manager creates images that reuse textures by asking the texture manager for the texture the Image needs. This allows for many objects to share a single loaded texture.

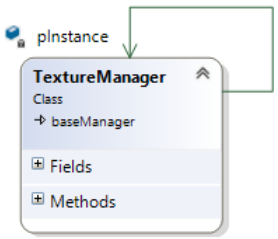
You can think of an Object Pool like a library, when you need a book you can ask the librarian finds the book and they will give you the book. When you are done, you give it back and walk away. The librarian puts it back on a shelf and if someone else needs it the librarian goes back and gets it off the shelf and gives it to them. It would be absurd if the librarian went off and burned the book when you returned it.

In software, The Object Pool pattern eliminates the absurdity of excess creation and destruction of objects and gives an easy interface to retrieve the objects when needed. An object pool manages the lifetime of objects, when objects are no longer needed instead of being deleted or freed back to the system, an object pool holds these objects in reserve avoiding additional and expensive calls to new and delete for improved performance. In large systems, and especially real time systems eliminating calls to new and delete can provide noticeable performance gains.

In my Space Invaders, you can find an object pool for nearly every class in the system, Textures, Images, gameSprites, boxSprites, proxies and more are all pooled to improve performance and simplify object management and ownership. This allows for a significantly reduced number of calls to new and since the object pools are integrated into the various managers there is an extra bonus as it assists in the creation of objects by allowing tracking of active objects for reuse like textures can be used by multiple images, multiple sprites can use an image… etc. Preventing duplicate objects from being created and taking up valuable memory. Needing to ensure no duplicates are created and providing access to the object pools leads us into the next pattern the Singleton Pattern.

{Singleton}

A singleton is a way of ensuring a class has only a single instance, and provides a global point of access to that instance.



The Texture Manager Is a Singleton it holds a refrence to itself.

You can think of the singleton pattern like a bank account, if you have an account at the bank of money, you can go to any physical bank, access it online, use a debit card and it's all one single account that all the transactions post to. If a bank account wasn't a singleton and you walked into the bank of money on fake street and deposited 200$ and then went to the bank of money on imaginary lane that 200$ wouldn't be there. It would be very difficult to keep track of where your money is, what charges are being posted to what account, what all the separate balances are. Just keeping track of what accounts you have open and what accounts you closed becomes a serious task as time goes on.

The Singleton Pattern in software acts in the same way as a bank account, you have one instance of a class and you can access that unique class from anywhere, alternatively the singleton can be used to control access as well. It is extreme in its simplicity, and is sometimes considered to be an Anti-Pattern. Where a class contains a static pointer to an instance of that class and the constructor is made to be private. The class is then responsible for instantiating itself, and ensuring no other instances can be created.

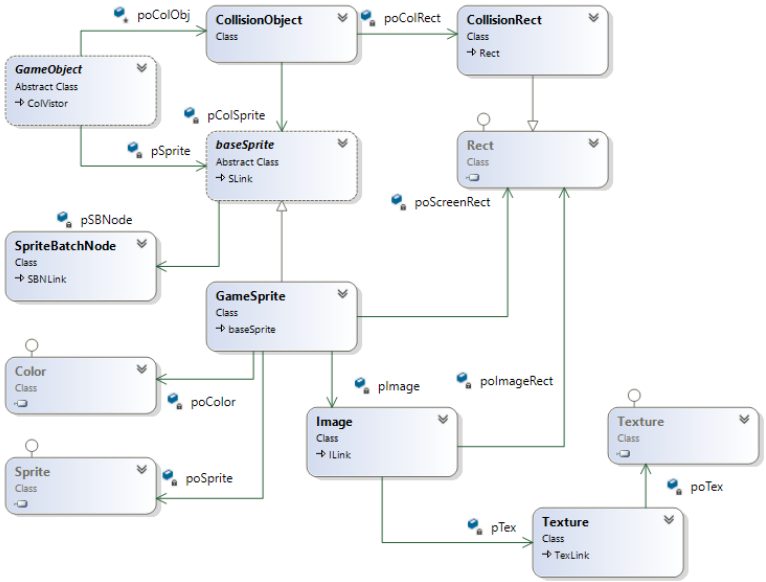
In my Space Invaders, the singleton pattern is used often, it is used for nearly every object pool and object manager to ensure that I have single instances of those objects, so I don't end up with excess objects, or losing track of managed objects due to not having the pointer to the instance of a given manager that has the object I need. It allows me to make a single call to the game object manager to update all my game objects, and a single call to sprite batch manager to render all my sprites to the screen. Now that I can organize and manage my different types of objects, I need a clean way to create them which leads me to my next problem.

Problem 2 – Creation.

I need A few Different types of aliens: Crab, Octopus, Squid but many different instances of them, a 5 x 11 grid means I need 55 aliens total. Same with Shields where I have 4 identical shields that are duplicated in different positions with many internal parts the shields at the time of writing contain 924 pieces spread across the 4 shields. Each of these pieces has additional moving parts under the hood. So, with Game objects being numerous and complicated to build with many moving parts, on different levels how do I get the objects that I need?

The solution – Adaptors and Factories

First focusing on making objects that were both useful to me and meaningful to the game engine I needed to build my objects on top of the foundation classes of the Azul Engine, I get 5 classes in the engine that are my way of communicating to the engine, Texture, Rect, Color, Sprite and SpriteBox. With just these 5 classes to work with I need to start building high level objects to simplify control, to accomplish this I, start building up to having a singular game object that I will use for most my operations. After Completing this I am left with an easy to use but complicated to build Game Object class that provides me all the functionality I need in an easy to use interface. But since all the pieces that I needed for all the systems to work meant that the Game object was much complicated under the hood. Which means that I was in desperate need of a way to let data drive object creation, building factories to allow object creation to be easily parameterized was an obvious choice at this stage. Building upon the foundation that I lay in solving the first problem I created additional managers for the objects that I needed. Which leads me to the first pattern to solve this problem the Adaptor Pattern.

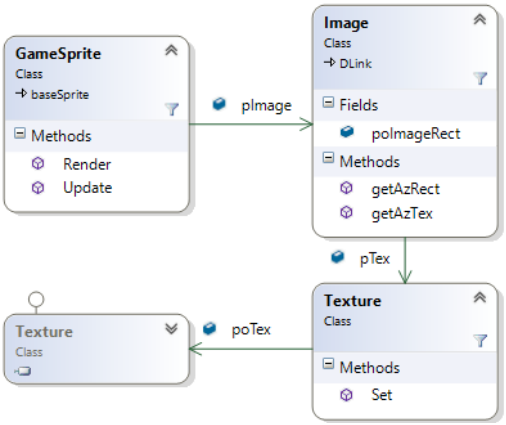


The game object and all the objects to make it work. Layers of adaptors allows me simple control over the complex object.

{Adaptor}

The adaptor is used to convert the interface of a class.

You can think of the Adaptor like adaptors for cables, the little things you stick on the end to allow you to do things like turn Digital signals to analog ones. Without it you could never connect a new device to an older TV. But because of the adapter the device can send digital video to an analog TV, and neither the TV or the Source Device need to have any knowledge of the adapter.



The Simplest example of an adaptor in my project.

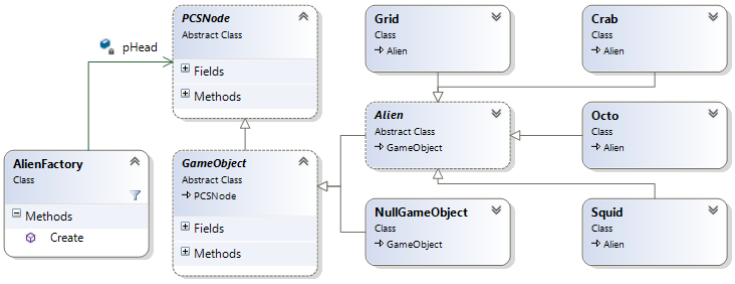
In software, it performs the same function of its real-world counterpart, it allows you to take an existing class that just doesn't quite fit your needs and add the logic needed to make it work for your system. So, the Target doesn't need to know about the "adaptee" and the "adaptee" doesn't need to know about the target or the adapter.

In space invaders I am using Adaptors to Adapt the game engine low level classes to forms more suitable for my needs. So, the Azul.Texure, Azul.Sprite, Azul.SpriteBox, and Azul.Rect all provide the underlying implementations and connection to the game engine that I need but are not very friendly to use "out of the box", so the adaptors allow me to build and use my more complex objects in a simpler way because the adaptors contain the underlying Azul objects and then the objects are built in a nested way.

For example, the Texture class is the simplest adapter in this project and is just a class whose only member data is a pointer to an Azul.Texture. A Texture alone doesn't quite get me anything functional it just allows me to load graphical data into the engine to use this graphical data I need to tell the game engine what portion of the graphical data I want. Having an Image class allows me to combine my adapted texture class and an Underlying Azul.Rect into something more useful, which is a pointer to a texture and what area of that texture I want to display. An example of a more complicated adapter is the GameSprite class, which contains a pointer to an Image class and three other Azul Objects: an Azul.Sprite, an Azul.Color and an additional Azul.Rect. This class allows me to use a collection of underlying Azul objects necessary to display the sprite on screen through one unified interface that can be used to change any necessary data in the underlying structure that can be used without knowledge of the way that the game engine or it's classes are implemented and the game engine never knows about the adapter.

{Factory}

A Factory is used to simplify the creation of objects.



The Alien Factory is in charge of creating Either a Derived Class of Alien or a NullGameObject on Error.

You can think of the Factory pattern the same way you think of a Factory in real life. When you buy a cellphone, you can say I want to buy a black cellphone that has 64 gigs of internal storage. You don't need to build it yourself or even know how it's built, you don't specify what type of camera it has, or any of the other tiny details that go into making a modern cellphone. You pick the options you care about and the rest is taken care of in the factory.

In Software, the Factory Pattern acts very much like a real-world factory, where you can get an object that satisfies your needs by simply specifying a few options, how the object is created and the objects it might depend on or how they are created are out of your control. This eases a client's burden when it needs a new Concrete Product. Very complex objects can be created by passing a small number of parameters.

In my Space Invaders, the factory pattern is used to simplify the creation of game objects, because game objects rely on so many underlying classes for their use, it is important to provide an easy to use interface for creating new objects. It saves development time by reducing the amount of code a client needs to get a new object, allows for the decoupling of client code from object creation code so a change to the factory can change how a concrete product or which type of concrete product is created without having to change every use of that object in client code. I also use the managers for object reuse and once a high-level object is created it is automatically placed into the proper manager so I make use of the systems I have already built. For example, when creating the grid of aliens, I can call the factory and pass the Enum of the sprite I want it to show and where I want it to start and the factory takes care of which alien type is created along with all the complexities that goes into making the high-level object. The first prototype used a game Sprite for every object on the screen and although this worked just fine it was very inefficient which leads me to the next problem I ran into, the desire to reuse sprites.

Problem 3 – Reuse sprites

I have just a handful of unique objects but need many instances of them.

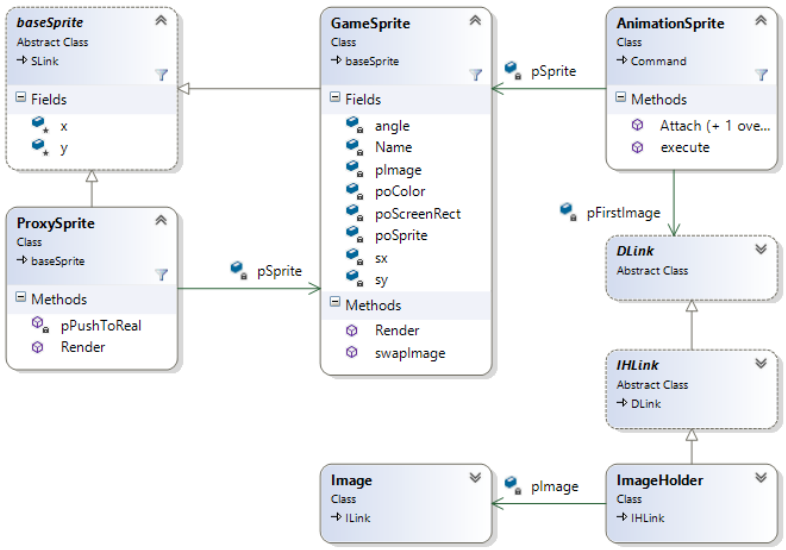
I want to animate each sprite but I don't want to animate each object in the grid separately.

I want to automate the animation process.

I want to avoid duplicate objects and redundant data.

The solution – Proxies and Flyweight

Proxy objects reuse the existing underlying Game Sprite, instancing only the data that is different, in this case just the x and y position. The proxy points to the full object and just overwrites the differing data on demand, in this case during the render call. This allows a single game sprite to be shared by many game objects, saving a considerable amount of memory per game object. As a bonus this allows the animation Object to bypass the proxy sprite and allow change of the image in a single location, in this case the gameSprite the proxy points too, and all proxies pointing to that gameSprite automatically are animated without any additional work since they use the game sprite they point to render themselves to the screen they always use the correct frame of animation. While proxies allow me to reuse portions of data there are some cases when I need to reuse a complete object.

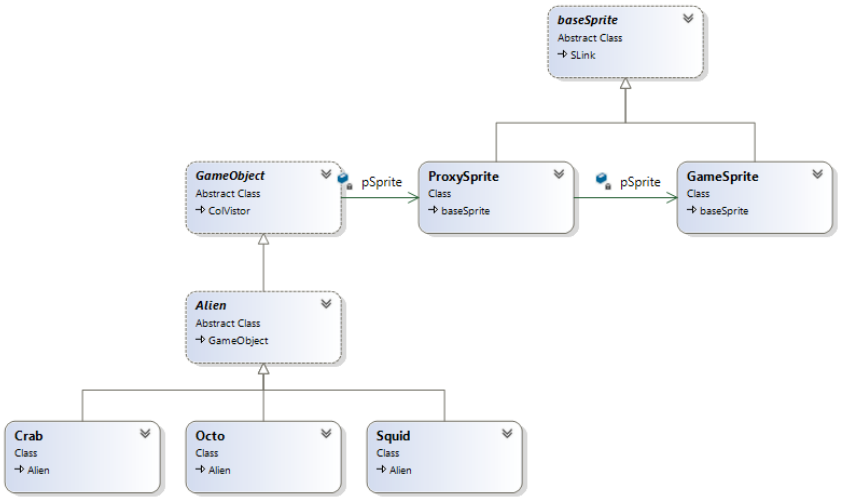


Animation Sprite acts on the few GameSprites and the proxy sprites don't need to be updated to apply the change in image.

{Proxy}

The Proxy pattern provides a point of access or placeholder for another object.

A proxy pattern uses objects to act as an interface for a large or expensive object that can simply foreword data to the subject or provide additional logic. It can also be used to limit or provide access to the underlying subject.



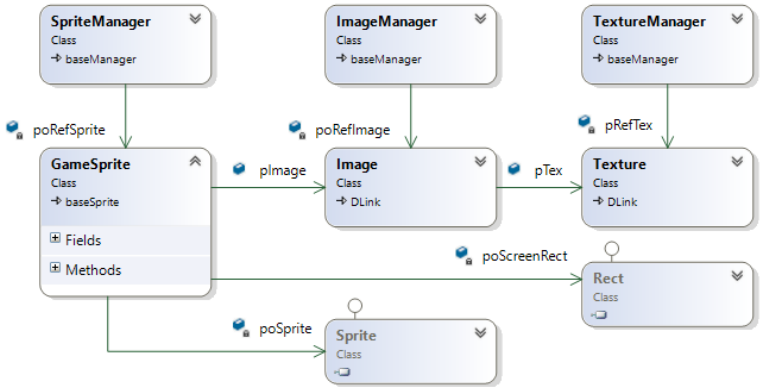
Proxy Sprite acting as a stand in for the game sprite.

In my Space Invaders, I use the Proxy Pattern to create cheap instances of game objects, so I don't have to duplicate data. The ProxySprite inherits float x and float y from the baseSprite and only has a pointer to a GameSprite as additional data. The proxy stands in for a full game sprite and allows me to treat it in the same way as the full game sprite without having to duplicate the data that is shared between all the proxy sprites. Things like Angle, scale, Image, etc. is shared between all instances of a game object, so I simply avoid duplicating that data. During the engines update phase the proxy sprites x and y position is updated and held internal to the proxy sprite. During the engines Render phase as it tries to draw the proxies, the proxy sprite pushes it's x and y data to the underlying GameSprite then is rendered at the correct position, allowing each proxy to be "stamped" in the correct position on the screen. However, there are also instances where I need an entire game object which leads me to the next pattern the flyweight.

{Fly Weight}

The Fly Weight Pattern allows resource preservation for many objects that all can share part of their internal state.

A flyweight pattern is a way to minimize memory use of many similar objects by storing shared data in a single place and having external classes/data structures store the external data pushing the external data to the flyweight temporarily only when necessary.



A Chain of Flyweights, a single texture can be used by multiple images, a single image can be used by multiple game sprites and each level up has the extrinsic data for the object below.

In my Space Invaders, I use it for both Images and Textures internal to GameSprites, for the Font system, and anywhere I can get away with sharing data. In the font system, I use a Glyph class to represent a single character, then a font sprite uses the shared glyphs to write a message. Reusing an entire glyph instead of creating a proxy to it. I just push X and Y data to it when necessary and use it as is.

So, was all this work to setup the proxies' worth it? Consider this the savings just in the 55 aliens in the grid with the ProxySprite being 12 bytes in a 32-bit runtime, and the GameSprite being 40 bytes and its non-shared objects 92 bytes overall in a 32-bit runtime, this is not counting the size of the image and texture. Considerably more still if we were to remove the flyweights for sprites and textures. Taking 55 GameSprites down to 3, is taking the original 5060 bytes down to 276 bytes. After adding the 55 proxy sprites back in totaling 936 bytes using the proxy sprites, the proxy method leaves us with just 18.5% of the original memory usage for the gird. Keeping in mind we aren't considering the cost of the images and textures also if we were to remove the flyweights and duplicate those things would get very costly. Having the proxies and flyweights working together gives us massive memory savings.

So far so good but things are starting to get complex, lots of things that could go wrong, which leads me into my next problem.

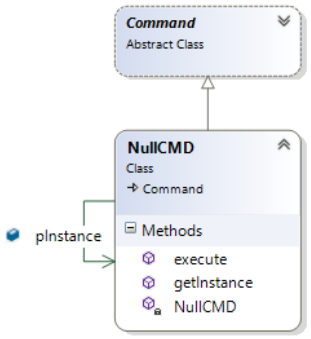
Problem 4 – Special cases

Game Objects have multiple uses, some like walls and bumpers do not have sprites, others like the ships in the life counter have sprites but no collision objects. Code is starting to get messy adding checks for null values. If you miss a check the game could crash. So how do we keep things robust and clean?

Solution – Null Object

Application of the Null Object Pattern is subtle and used throughout Space Invaders, by replacing a null value with a Null object I prevent crashes in the case that something was to not be initialized in time and remove having to check for special behavior of game objects before use. Being able to treat all Game objects as if they were full featured removes a lot of time and worry of checking for the strange special case. If an object were to not have a sprite that is displayed, the Render call for the object just does nothing, rather than needing an "if(obj.pSprite != null) { obj.pSprite.Render();}" you just call Render always, and when an object doesn't have a sprite, nothing happens. This is the simplicity and the beauty of the Null Object Pattern.

{Null Object}



The Null Object pattern improves robustness through the intelligent application of nothing.

The Null Object pattern is subtle but extremely useful to add robustness by providing a default object that implements an expected interface but has no code in the method bodies. This allows a system to treat all objects uniformly even if an object or part of an object would have been a null reference. This prevents crashes due to null references and since the null object's method does nothing you can be sure that you are not adding unexpected side effects into a system. Because it predictably always does nothing.

In my Space invaders, I use it in place of null to prevent crashes due to null references. As you can see In the Diagram the null command object is both a Null Object and a singleton, since there is no point in creating multiple objects to represent the lack of an object. I also have null Sprites, Null Collision Objects, anywhere where something could be null it is helpful to supply a Null Object as a placeholder to add robustness. Even more helpful when you can use a null singleton to prevent duplicating a null object.

Problem 5 – Who collided

Big problem, I needed a very quick and efficient way to discriminate collision decisions, who collided with whom? It needed to be extensible and scalable, no conditionals to worry about and no switch statements to slow things down. There are multiple levels of collisions as well, with the grid of aliens existing in a hierarchy, different collisions need to be processed at different levels. For example, a missile needs to collide with a specific alien, while the grid colliding with a wall doesn't need to dig any deeper to know what to do.

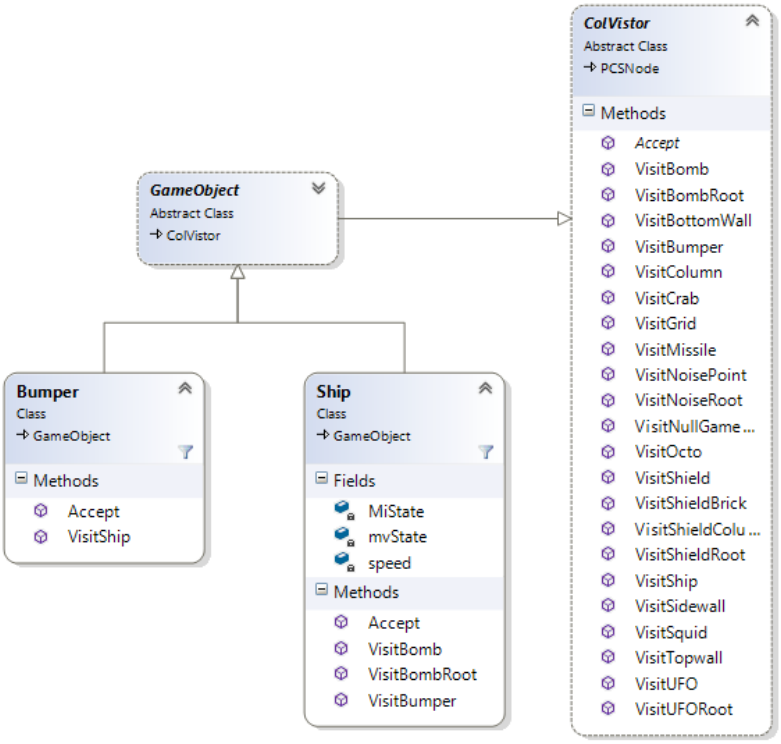
Solution – Modified Visitor pattern

This is one of my favorite tricks, Double dispatch allows the Visitor Pattern to be used in reverse as a very clean solution to a complicated problem. Removing switch statements and efficiently making the decisions for collision reactions. It avoids a common pitfall of handling collisions at a lower level, this system allows all collisions to happen at the game object level. Abstract class for virtual functions prevents unnecessary implementation of an interface for collision logic between objects that should never collide, while providing safety in debug mode if somehow a collision does occur an assert is thrown to alert us of things going awry.

{Modified Visitor}

This Pattern is used eliminate costly switch statements and searches.

By accepting a visitor of an unknown type with the Accept method then immediately calling the appropriate visit method of the accepted visitor. We get the explicit types and pointers of the two interacting objects which allows us to jump directly to the code we need.



In my space invaders I am really only using the Double Dispatch mechanism of the visitor pattern in order to handle events at the game object level requires each GameObject to implement the Accept(ColVisitor other) method. Each implementation is a single line for example if the ship collides with something, the ships Accept(ColVistor other) method is called, in the Accept method the only line necessary is the other.VisitShip(this);. This allows the ship to visit the accepted object. VisitShip(Ship pS) is called in the accepted object in the visit ship method you have the benefit of being in one of the objects code with a pointer and explicit type of the object visiting. I now have the pointers and types of two objects that are colliding with 1 line of code, replacing any previously necessary conditionals, searches, lookups, switch statements with 1 line of code. Then in the VisitShip method of the accepted object I only require 1 more line of code to get to the appropriate reaction. Say a Ship collides with a Bumper while it is moving, in the Bumper.VisitShip(Ship pS) method I have Reactions.Reaction(this, pS); "this" being the bumper and pS being the ship, Using the Reaction alphabetically allows for uniform call in the case that a bumper collides with a ship or a ship collides with a bumper. Reactions being a static class with no data and only public static Reaction() methods. Each reaction method has two parameters, one for each type of object that can interact. Since I also use virtual methods in the abstract class I don't force objects that shouldn't collide to implement a full interface. However, being able to quickly and efficiently decide what objects are colliding and get to the proper Reaction Method is great but once I get there what do I do? Which leads me to my next big problem, Notification.

Problem 6 – Notification

How do you notify the proper objects of collisions so that the proper reaction occurs? If an alien collides with a boundary wall, it needs to change directions not explode. If a missile hit's a shield the shield needs to erode. The systems need to be able to handle their own behavior in these events.

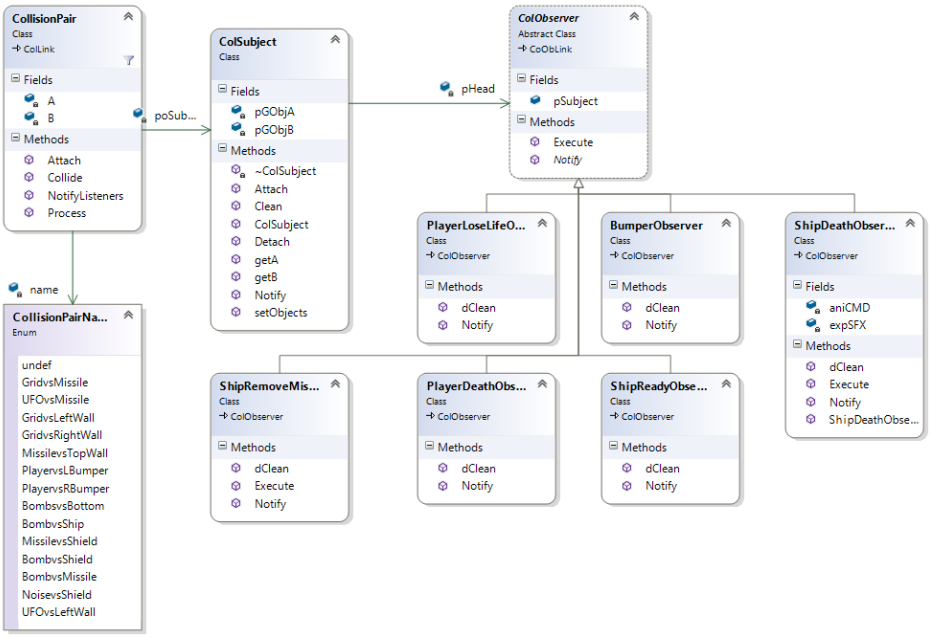
We don't want to provide every class with knowledge of the collision system, and we don't want to handle the collisions in the visitor pattern because things will get large and duplicate code will start to make things messy, and we certainly don't want to deal with flags in each object and poll during the update loop to check for collisions.

Solution – Observer pattern, Command pattern and priority Queue.

Once the proper Reaction method is reached the collision event iterates through a list of observers to let every class that needs to react to a collision know who collided and the behavior is encapsulated inside the observers. So, the collision system and the reactions are decoupled. For example, when the missile hits a UFO several observers get notified, the player's ship needs to be notified that it is ready to fire another missile, the score needs to be modified, animations and sound effects need to occur. Which brings me to the first Pattern used to solve this problem the Observer Pattern.

{Observer}

This pattern defines a one-to-many dependency between objects, allowing objects dependents to be notified and updated automatically.

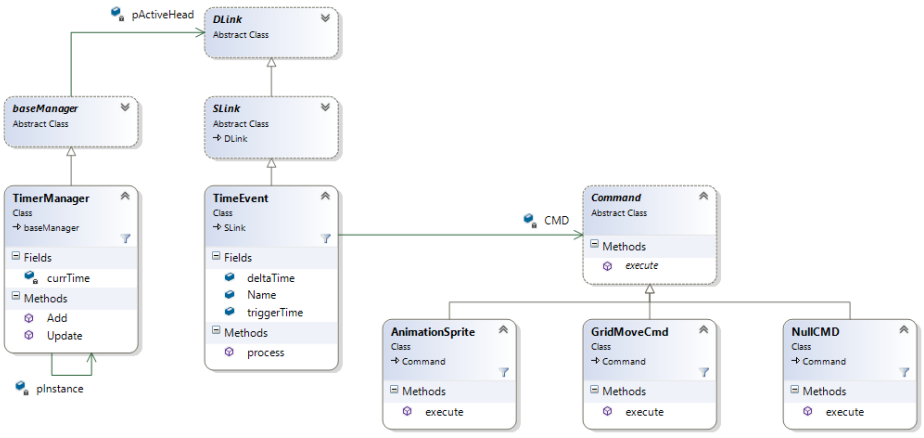


The observer pattern or the publish and subscribe pattern

In my Space Invaders, the observer pattern is used in the collision system, once two objects collide it begins to traverse the Hierarchies of objects inside the visitor, it relies on the CollisionSubject of the collision to notify its observers, being data driven the visitor doesn't require any knowledge of the number of observers, the behavior is delegated to the observers. Allowing for simple behaviors to be defined, multiple simple behaviors can be chained to make more complex behaviors without extra code. Example, having an observer that plays a sound, an observer that removes an object, and an observer that places a sprite to be removed later are used to create the effect of a missile hitting an alien. The sound observer plays the alien death sound, the object removal observer marks the alien for removal from the grid so it can be placed back into the graveyard for reuse later. And two temporary sprite observers are used to create the graphical effects of the collision, one is used to place the explosion effect for the alien and the other for the explosion effect for the missile, each of those observers push a command to the timer manager for the effects to be cleaned up after half a second. Which brings me to the next part of the solution the command pattern.

{Command}

The command Pattern is used for dealing with requests.



IThe Timer Manager has a Priority Queue of Time Events to process, Each time event points to a concrete command to be executed at agiven time.

You can think of it kind of like a TV remote each button on the remote may do one simple thing like add a digit if you are changing the channel or more complex things like launching an app, changing inputs. It doesn't matter you press each button in the same way, the discrete steps to carry out a command are hidden from you.

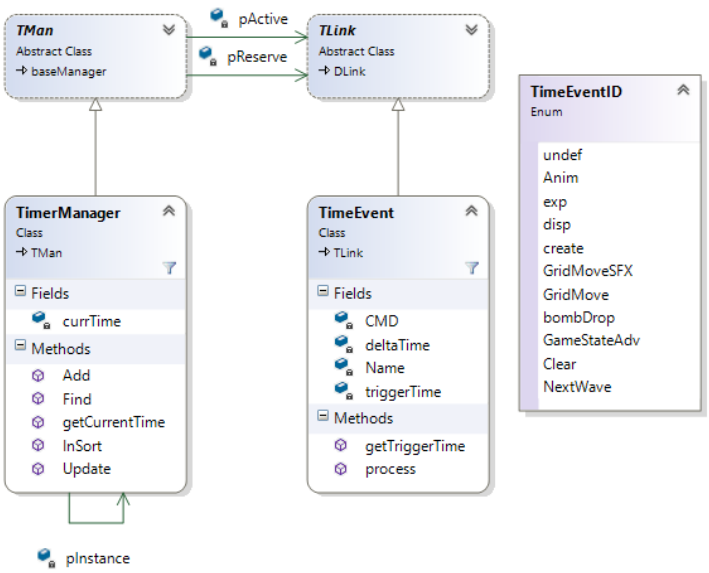
The Command pattern is a way to store information needed to perform an action in an object. One of the primary benefits of the command pattern is the ability to handle a wide variety of concrete commands without needing to know where the commands originated or what the command entails. Simply put that object in a queue to be completed as soon as possible or it is useful to trigger events based on a timer.

In my Space Invaders, I use the Command pattern to handle both the sprite animation events and object movement events for the grid but also handle any cleanup of visual effects, delayed creation of objects like the UFO, animation of the player after death. The commands are encapsulated in a Time event, and the Timer manager maintains a priority queue of time events, and the commands are processed from highest priority to lowest if the trigger time for a given event has past. Neither the timer manager nor the Time Event knows what the Command entails, it just knows to call it's execute function to make it happen. To make sure commands are executed properly at the time they are specified I need to use one more pattern to help me the Priority Queue Pattern.

{Priority Queue}

This pattern allows for tasks to be completed based on priority.

This pattern takes a standard FIFO queue and provides a priority system to allow for items with a higher priority to be processed sooner than those with a lower priority. It is a relatively simple pattern that can be implemented in several ways.



The Timer Manager uses the current game time to see if it is greater than the trigger time before executing the command, it only process the list if the front of the list needs to be executed. Cleaning the list as it goes.

How I used it in Space Invaders, Timer manager handles tasks on a doubly linked list that uses insertion sort to add events based on their trigger time. It is very simple to do and provides me the ability to script events very easily, for example when the missile hits a UFO the Observer pushes a command to clean the explosion sprite in half a second and to place the score sprite in half a second as well as to remove the score in a full second. This also ensure that Grid movement, animation and sound are synchronized. The cyclical nature of grid movement and animation mean that the commands place themselves back on the timer. I can place commands on the priority queue and rely on them getting processed in the correct order. It is such a simple thing but very helpful.

Problem 7 – Getting data

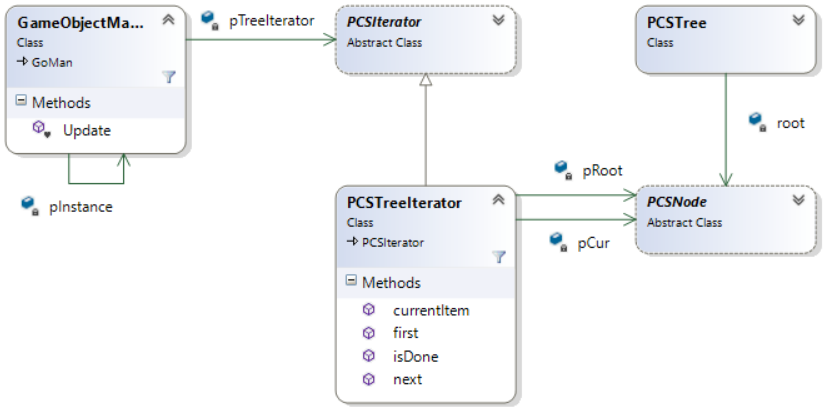
Most of the data is stored in the managers in linked lists but many game objects exist in a hierarchy tree structure. How do I access the data without direct references and no knowledge of the topography of the tree or ideally not knowing anything about the data structure at all?

Solution –Iterators

So, to separate the data structure from the use of the structure I rely on Iterators to handle the details of tree traversal, and to provide a clean and familiar interface for acting upon the objects. Iterators are another simple pattern with serious benefits. Firstly, what it says on the box it allows you to separate the details of a storage mechanism and it's use, your data can be as complex or as simple as you need, you can even change the underlying structure and update the iterator and not have to change code in every class that uses it. Iterators also allow you to write cleaner code when dealing with composite patterns or complex data structures, turning a complex traversal algorithm into while(iterator.hasNext()) { dowork; iterator.next();} making the code easier to develop and maintain.

{Iterator}

The iterator provides a way to access elements of a data structure without exposing the details of its implementation or its structure.



The game object manager uses an iterator to call update() for all objects on its active list.

The iterator accomplishes this by defining the traversal algorithm internally and providing an interface to advance through each item in a structure. Maintaining a pointer to the head of the structure and a pointer to the current item of the structure. Advancing the current pointer on each next () call returning the new current item, and can be polled to see if the structure has any more elements.

How I used it in Space Invaders I used it in the GameObjectManagers update() call to update all of the active game objects. The way that they are stored is each tree or singular object exists on it's on node in the game object managers active list. The nodes are in a linked list and can either contain a tree in the case of the alien grid, or shields, or a singular object in the case of the player ship or the missile. Having the iterator allows me to iterate through tree in each node the same way. Singular objects update and report completion after they are updated, while the hierarchies of objects iterate properly though.

Problem 8 – Special Behaviors

How do I have my system specialize behavior automatically? Behaving differently based on the data it contains, isolating the specialized behavior from the shared behavior?

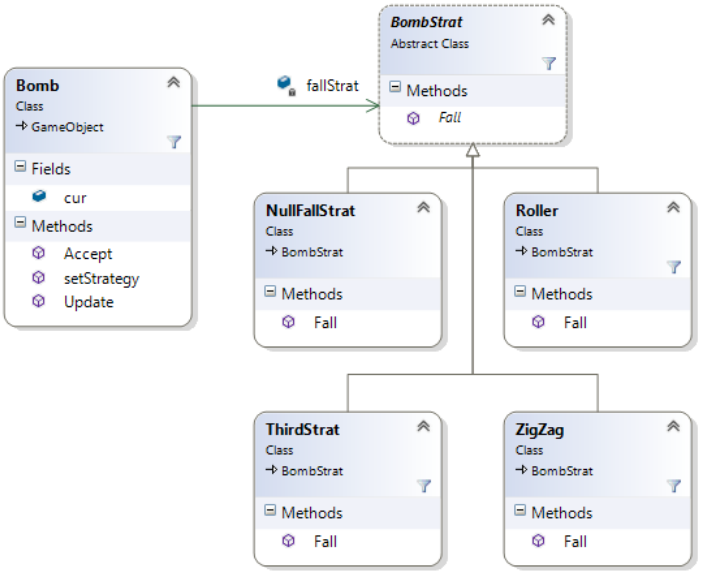
In an organized and easy to extend way that removes checks for special cases?

Solution – Strategy pattern

The strategy pattern here is used to create some graphical variance in the bombs that the aliens are dropping, each bomb upon creation is given a strategy to use for its fall during the update phase. When the bomb is updated it calls fall in its strategy passing a pointer to itself the strategy determines the special behavior and updates the bomb accordingly.

{Strategy}

The Strategy Pattern allows for the creation of a family of interchangeable algorithms.



The strategy pattern works by defining a simple interface sometimes if not usually a single method, then by isolating differing algorithms into separate classes, encapsulating the behavior in interchangeable concrete classes so that behavior can be selected at runtime by supplying an object with a concrete strategy.

In my Space Invaders, I used the strategy pattern for the bombs that the aliens were dropping to allow them to cycle through Images based on a type of bomb also bypassing the animation sprite system to allow each bomb to iterate on its own timer. I integrated the strategy into the bomb factory so there is only a single instance of each strategy that gets created and then based on what type of bomb I want I set a pointer internal to the bomb to that strategy, since the strategies are singular in the fall method the bomb passes a pointer to itself so the strategy can decide what Image it should have for a given frame.

Problem 9 – Hierarchy

How do I efficiently rule out collisions but also find objects that are colliding in an efficient way?

How do I organize my data into a structural relationship that optimizes performance and reduces the work necessary to enforce the rules of the game?

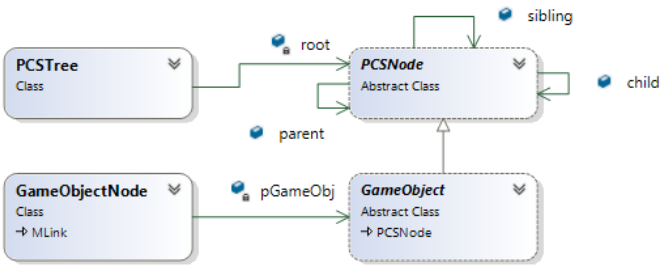
Solution – Composite pattern (PCSTree)

Up to this point I have only ever discussed data structures in terms of the DLink doubly linked list implementation. I have however been hinting at objects existing in hierarchy's, the grid, Shields, Bombs all exist in a tree structure called a PCS Tree. The structure does a lot of the work for me, for example during a collision I check if the missile is colliding with the grid of aliens in its entirety. The grid's collision box is the union of all its children, in other words it encompasses all its children. If the missile is colliding with the grid, the reaction then checks the missile against each existing column in the grid, the column's collision box if the missile is colliding with a column then in the reaction it checks each alien, if it's colliding with any alien in the column we have a collision. During each step, it only does the work that it needs to, if during the grid vs missile check there was no collision it wouldn't check the missile against the columns, also avoiding the missile against each alien in the grid. Providing early outs and allowing collision detection to start at a high level by checking against the entire collection then working its way down until it has the specifics. This requires the data to be organized in a structure in a meaningful way so processing can occur in layers. This is done with the Composite Pattern.

{Composite/PCS Tree}

The Composite pattern provides a means to uniformly manipulate singular objects and groups of objects.

The Pattern or in my case the PCSTree stores pointers to its neighbors in the tree rather than each level containing a complete list of children.



The PCS Node takes up a predictable amount of space and many nodes can make any topography.

In my Space Invaders, I used a variant of this pattern called an SPC (Sibling Parent Child) Tree. Which eliminates the need for the composite objects to have a separate data structure that contains its children. Bypassing the need for slow dynamic storage internal to each node. The game Object manager has a doubly linked list of game object nodes, the game object nodes contain the game objects, the game objects may or may not be the root of a tree of game objects. So, I use iterators obfuscate the details of the data structure from the GameObjectManager. This also allows collision processing to occur based on the depth of the tree, Grid is the root, columns is depth 1, aliens depth 2.

Problem 10 Modes

How do I cleanly handle the flow of the game?

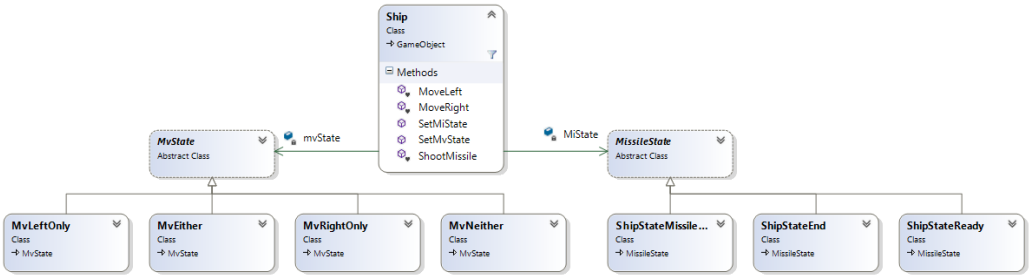
How do I cleanly handle the modes of a game Object?

Solution – States

To handle the flow of the game and to allow an object to alter its behavior based on context I avoid multiple conditionals I use game states to control the flow of the game and the states of objects. This is the object orientated way to implement a state machine, with the State Pattern a full class represents the discrete states. With the behavior, internal to the state, an object's behavior changes when it's active state changes.

{States}

The state pattern implements a state machine using classes.



The State pattern structurally and mechanically is like the strategy pattern but with the added ability for an object to switch its current state based on events or actions internal to the concrete state. This is a clean way to change behavior at runtime while avoiding complex conditional statements.

In my Space Invaders, I use States to control the flow of the game from the Splash screen to the select screen into the game screen, to game over when the player runs out of lives and back to the select screen after the high score has been updated so the game is ready to play again. I also use it to control the player ship behavior. The Player ship has two separate states internally. One to control the firing of the missile and the other to control the player's movement. Since in space invaders only one missile can be fired at a time the ship starts in the ready state and upon firing a missile and while the missile remains in the air the ship cannot fire another missile. Once the missile collides with something the missile is set to state end, once the missile is removed from the screen the state is set to ready so the cycle can begin again. Movement is controlled in a similar fashion when a player collides with the boundary bumpers that limit the players movement the state is changed to disallow further movement in that direction, this prevents the player from purposefully or accidently moving through or getting stuck in the bumpers.