Space Invaders Documentation

SE 456

Eric Fleming

Table of Contents

Problem #1: Standardizing Object Creation

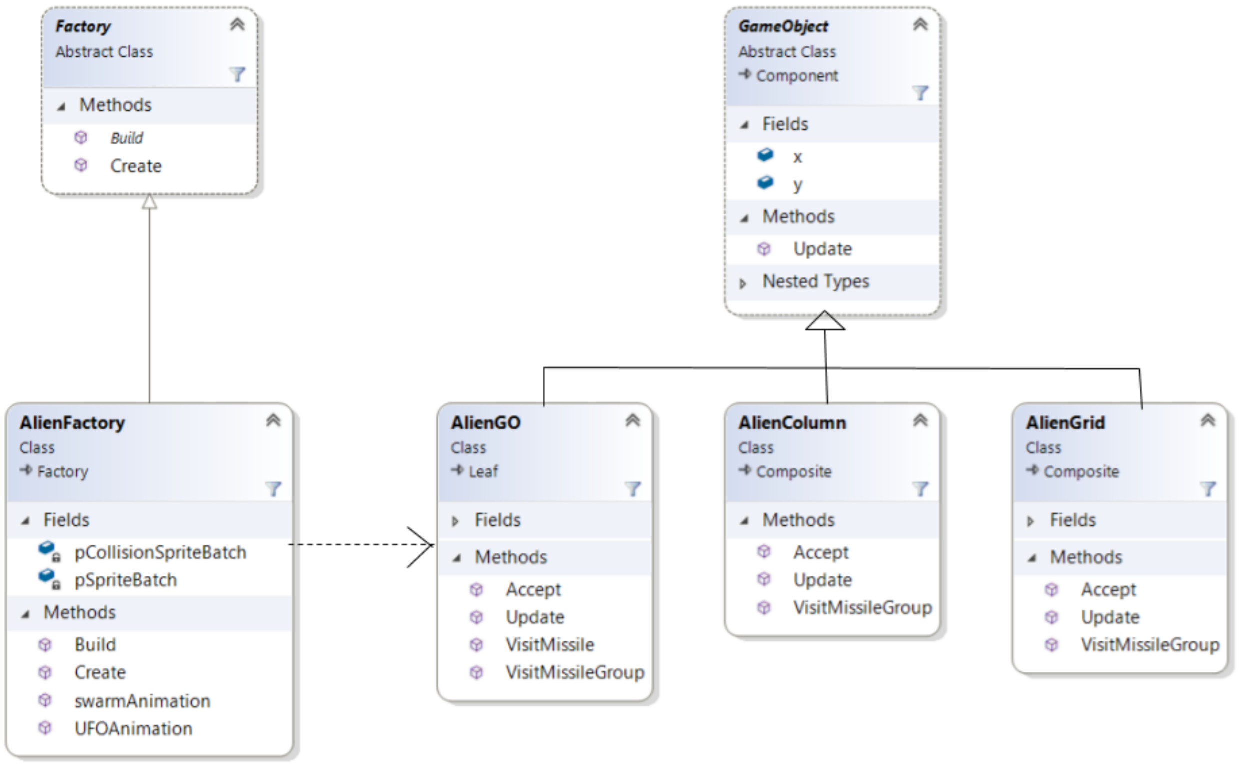
Diving into recreating arcade videogame Space Invaders we can recall that there are many pieces to the overall game. You will encounter 55 aliens per, a nefarious UFO, four giant shields made of smaller pieces, and many missiles and bombs moving across the screen. It would be a real pain if we have to manually configure the data of each of these game elements for two reasons: It would be extremely repetitive and time consuming. There could be many parameters to set in a given class even though the use cases only permit a few combinations of these initial states. This feature leads us to the implementation of the Factory Pattern.

Factory Pattern

Overview

The Factory Pattern defines an abstract class which has a common create method in its interface. When you ask a concrete factory to create an object for you it then searches for the kind of object you want and then calls that object's constructor. Depending on how many parameters you have in your create method, you can supply as many custom parameters to each constructor that you which. It is preferable to limit this. If you are supplying too many parameters it means that you could probably encapsulate some of these changes with adding more cases to match against in your factory. It's okay to call the same constructor more than once and pass it different preset values. When the object is done it will return a reference to you.

UML



Internal Workings of the Factory

This factory is concerned with generating a specific class of Game Objects that are of an Alien Category type. Game Objects are the programmatic game elements which process information and ultimately get pushed to Game Sprites which get rendered on the page. This is why in the diagram above you see that the Game Objects have references to x and y coordinates. While each Game Object has many specific methods, Game Objects of a similar Category possess many common methods which are also highlighted in the diagram. This is how you know it would be a good idea that these Objects should be created in the same factory.

Essentially you supply the Game Object's name which signifies the class that you are targeting for creation and the relevant Game Sprite that you are targeting to delegate to the right constructor. Let's not also forget that you should specify the initial position - (x, y) - of the Game Object. With all the information accounted for we then match the Game Object's name against a switch statement which targets the correct constructor. Before returning the reference to the created object we perform some house-keeping to make sure that other managers have a reference to this object so they can be rendered to the page.

How I use the Factory in the Game

You will notice that in my diagram I exposed another method common to factories call Build. A factory's Build function is essentially the same idea as the Create method but at a macro level. For example, I know that at the start of the game I need to build 55 aliens in 5 by 11 grid. This means that I need 55 calls to the Create method. Well I take care of this all with the main build method and smaller private build methods. To build this alien grid I use two private build methods. The first private method creates the alien grid and many alien columns. The second private method fills each column by calling the create method multiple times. So one you do the hard work once, you could hypothetically create as many fully filled alien grids as you wanted by calling on the factory build method.

Also used the same pattern with the Shield Category of Game Objects. This had an extra layer of depth compared to the alien grid but the overall execution was the same. Custom create methods and custom build methods to generate the 4 shields.

Problem #2: Organizing Common Resources

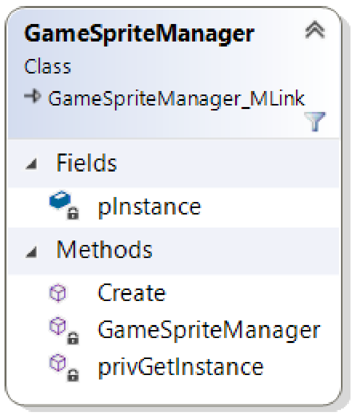
Taking a close look at our design we realize having factories that create a multitude of objects is great, but this creates another problem: How do we best organize all of these Objects and References to their internal state. We begin to notice that many of the fields between our objects share common values which is immutable. This means that each Game Object doesn't need to own/possess all of its internal data but can hold pointers to it. This means that we need to store this information in well-defined location and create methods to search for it. Sure we will pay a small time cost to perform this search for this data but since the number of unique items is relatively small we can spare the extra execution time. This problem leads us to implement two different patterns. We create Singleton-Manager classes that represent instances of homogeneous repositories for object to look up and reference. The specifics of how we keep our memory (space) footprint small is through the Object Pool Pattern.

Singleton Pattern

Overview

In Object-Oriented programming sometimes we want a single unique instance of a class. Generally this is because multiple other classes intend to rely upon it and we want all of these classes to get their stories and states straight. Ensuring that there is only one instance of this class allows for such consistency. Instead of having a public class constructor a singleton class has a private constructor. The class has a private static reference to a single instance of the class and can either be initialized at run-time or its creation can be deferred until the first time it is called upon. In this project we took the first approach more often than not. All of the methods for a singleton class are static as a result and apply changes or pass messages from this static object reference.

UML



Internal Workings of the Singleton

As mentioned in the overview I took the first – lazy – approach to most of the singleton managers. At the beginning of the Game when we are loading the content of the game. Each Manager class invokes its Create method which asks if there is already an initialized private instance of the class: what you see as pInstance. Therefore you must call the Create method at least once or pInstance will be null. Every time a static method is called on the singleton it will first internally call privGetInstance, which is another method which double checks that the singleton has been initialized.

How I use the Singleton in the Game

To reduce the overall memory, aka space, that the game takes up I have these singleton managers preside over immutable objects that will be shared and referenced in multiple classes throughout the Game. Examples of this include the Image Manager which hold the positions of essentially the sub-rectangles in tga texture files which correspond to graphics in the game. In the diagram above you also see a Game Sprite Manager which performs similar data management. The Game Sprite Manager holds the position, size, and orientation of the Images to be processed and moved about the screen by the Azul Game Engine: the backbone on which all of this logic rests.

These Managers are singletons I can Invoke references to these in any file and be sure that I am referring to the same location and don't have to worry about passing the right object as an argument in some function.

Managers

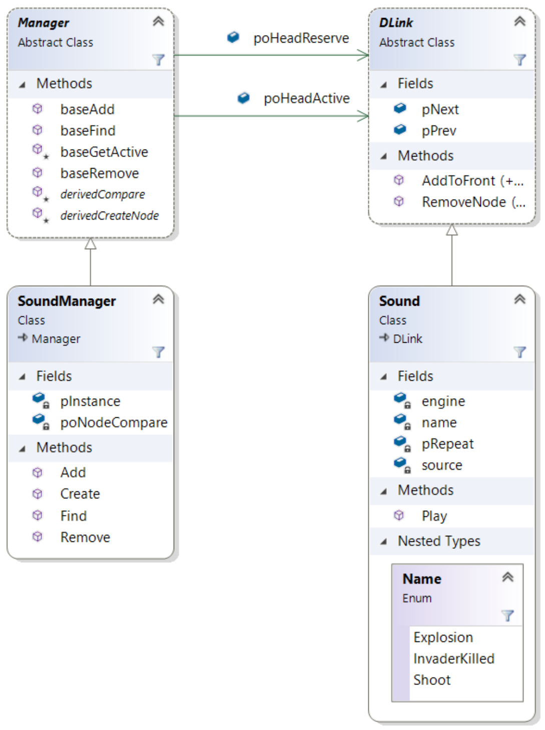
Overview

The Manager is an abstract class that defines a common interface for storing and retrieving data of the same type/class. The Manger Pattern is generally combined with the Object-Pool Pattern which I will describe in the next section who is responsible for the actual storing of the data.

When you want to add something to a concrete manager it calls upon the object-pool to provide it with a blank copy of the type of object you want and you overwrite it and store it in the active D-Link. When you call remove it moves the node from active to reserve.

To keep the code clean we have base methods that handle working with the D-Link, which call derived abstract methods that each manager implements and substitutes in class specific logic.

UML



How I use the Managers in the Game

I have many managers: Texture, Image, Sound, GameSprite, GameObject, ProxySprite, … , etc. The goal of each manager is to facilitate ownership which allows for the possibility for reuse instead of recreating another instance of the class with the exact same properties. This also makes Communication and notification between objects easies because you essentially have a phone book to look up a reference to a given object. One case in particular was how I got the alien grid to refill itself when you kill all 55 of the aliens. The problem was that when the last Alien GameObject was removed I would have lost the reference to its parent, but that reference still existed in the Game Object Manager so I was able to use its find method and then ask the Alien Grid to repopulate itself with more Alien Game Objects.

This was also similar to I implemented sounds. Instead of every object that plays a sound carrying a copy of it the Sound Manager has its own copy. If an object needs to play sound for movement or destruction its as simple as writing two lines of code. Find a reference to the Sound using its name from the sound Manager, then ask the sound to play itself since I also gave the sound class a static reference to a sound engine. Again every sound doesn't need its own engine. None of them needed to own it. We just needed a common location to search for what we need. That is the purpose of the Manager class.

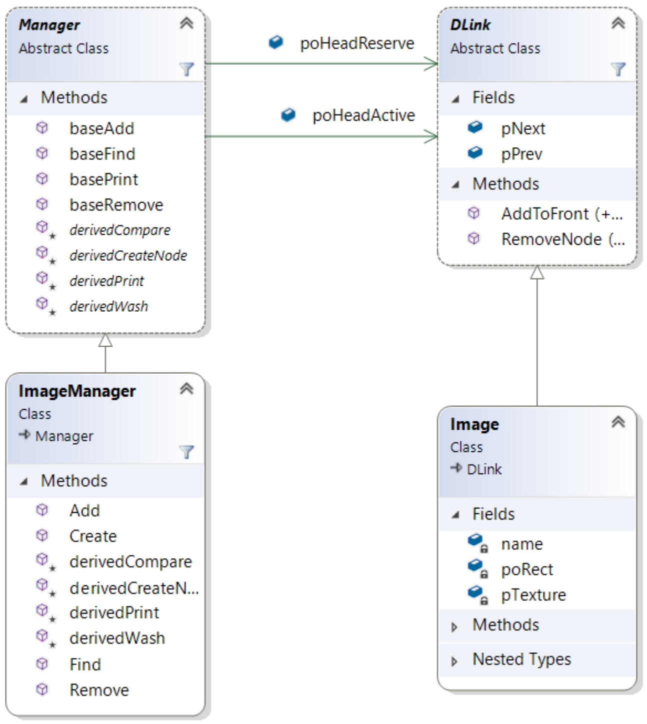
Object-Poll Pattern

Overview

This pattern is designed to keep our programs lean. Its main purpose is to cut down on the number of times we call upon a constructor which takes more time than it would to write over an existing object that we no longer needed. This is because we don't need to go bother asking the OS to provision new memory for use as frequently.

When you want to add a "new" object the pool first checks to see if it has any premade objects on the reserve list. If it does instead of passing the parameter to the class' constructor it instead passes it to the classes Set method. We then append this object to the active D-Link.

UML



Internal Workings of the Object-Pool

When you are adding an object to the pool there are two things you need to consider. What if there are no nodes on the reserve list and to be careful of your pointers. This means at the beginning of every add method we need to check to make sure we can pull a blank object from the reserve. Each Pool has a private method which creates a default number of blank nodes to the reserve: you set this number in the constructor, default is 1. From here you have a blank object but you need to make sure it doesn't point to any of its previous partners in crime because all object held by managers extend D-Links. We call a clear on this so now the object is pristine. Now we can set the object with its parameters and let the Manager add this fully formed object to the active list.

How I use the Object-Pool in the Game

The benefit of the Object pool is that it grows dynamically. It is a good practice to understand the initial state of the system so that you can set the reserve list to meet that goal upon loading of the game. For instance with the Font and Image managers we should expect to see at least one copy of each of these so do a quick count and set it to that. However when it comes to the Game Object Manager things can get a little more complicated because we will be adding new missiles and bombs to the screen.

Problem #3: Separating Ownership from Use

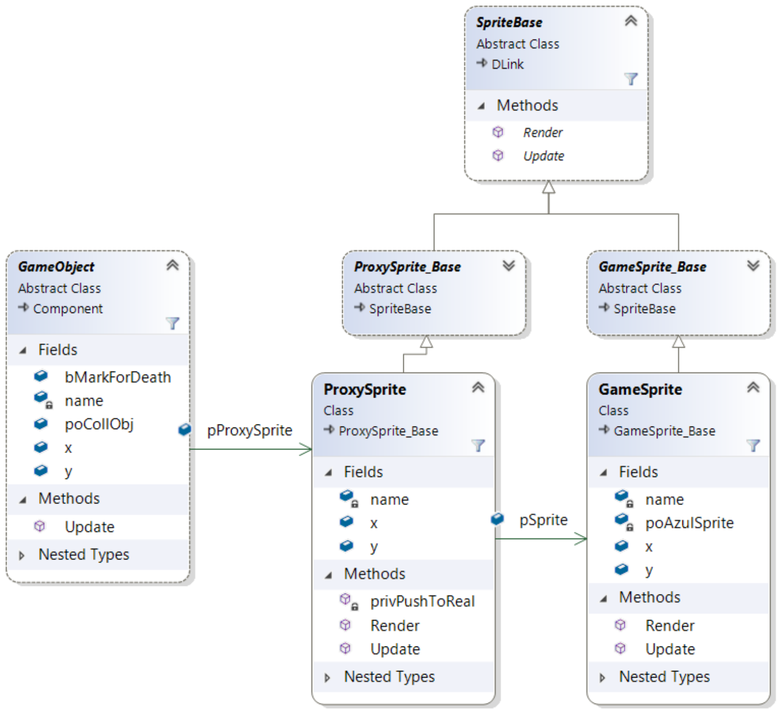
While the Texture, Image, Sound, and Game Sprite Manager Object Polls provide us with a common interface to reference commonly used information without owning it, we run into this same issue again when we look at the Game Object class and all of its derived sub-classes. Thinking back to our alien grid, sure we have 55 Game Objects but is really more like 3 Game Objects, an alien, and octopus, and a squid that all have unique x and y coordinates. This means we need to create an intermediary class which extracts the position and size information from the Game Sprites. Similarly when we need text to be displayed on the screen we get a similar issue. When we need to write a words to the screen we will invariably end up reusing the same letters. We can store references to these and write over them with new positions on the page when they are needed. These cases are slightly different in how they marshal data back and forth. Each situation represents a design pattern. The first is the Proxy Pattern and the Second is the Fly-Weight Pattern.

Proxy Pattern

Overview

Instead of everyone owning their own copies of real objects, which are mostly the same, we introduce a proxy objects to encapsulate the fields which constantly change from the fields that seldom change in the real object. When we need to access the real object we delegate our request to the proxy object who updates (sets) the fields of the real object. This pattern hinges on there not being any race conditions on the real object in question otherwise someone else might call an update after we update but before we execute.

UML



How I use the Proxy in the Game

Each Game Object instead of holding its own unique copy of a Game Sprite instead holds its own Proxy Sprite which can be thought of as a loose wrapper around the Game Sprite. The idea is that the Proxy Sprite should only need to carry the information about itself that changes. Namely these would be the x and y coordinates as I did not have anything in the game that required me to change its size or alter its angle although I did enable to proxy to handle these situations as well.

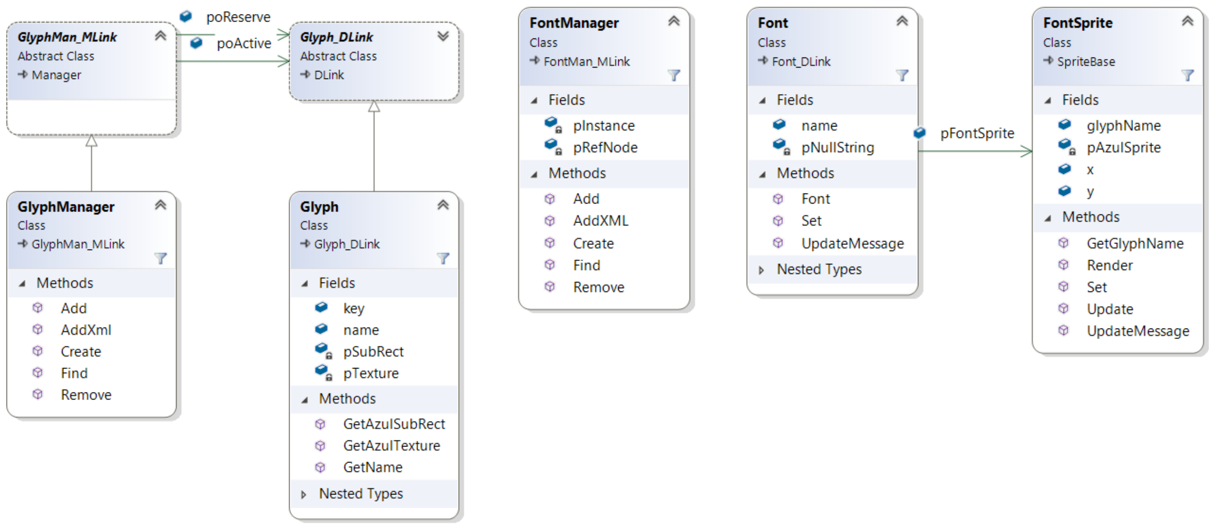
When the Game Object needs to access information about its Game Sprite it delegates its update to the Proxy who calls its update method which makes sure to copy the data that has changed over to the Game Sprite (privPushToReal method) before it moves forward. This is important because this is what enables us to draw 55 space invaders to the screen with essentially only 6 Game Sprites. When the Render method is called it first calls update making sure that all the data is set in the Game sprite, and then it gets stamped to the page. When the Next Proxy goes through this process it just over writes what it needs and you see a "new" Game Sprite in a different location.

Fly-Weight Pattern

Overview

This pattern feels similar to the proxy pattern but its main difference is the direction which the data is moving. For the flyweight we have a collection of Fly-Weight objects which are shared. The sharing mechanism I am using is the Manager class. When we want to create a new instance of a Fly-Weight we first look for it. If it is not there we create it and return it.

UML



How I use the Fly-Weight in the Game

I used this pattern initially for the Font system. The Glyph Manager first mines all of the characters in the file, parsing it correctly with the AddXML function. Then it attaches them to the Manager as Glyph Objects. Later when trying to construct a specific font, or a string of Glyphs we consult the Glyph Manager to see if it has what we need.

Similarly when I was thinking about the best way to get explosions to appear on the screen I returned back to the Fly-Weight Pattern. Here I made an Explosion Sprite Factor which contained a simple D-Link which acted as its internal manager. When you create the Explosion Sprite factory for the first time it load the Factory with one copy of all of the types of explosions. Later if someone else asks for one you search through the manager to find it and return a copy. There is a fall back case where if the Explosion Sprite is currently not in the manager it will make a new one for you and reattempt at grabbing it for you.

Problem #4: Knowing When Objects Collide

In Space Invaders we can be sure that we will run into a fleet of aliens. The grid of space invaders marches across the screen raining bombs while you fire missile up at them. The shields can only protect you for so long as they begin to whittle away: collision after collision after collision.

I think it would be appropriate to say that handling how objects collide is the heart of this game. You can trace all important points to objects position and if they overlap (aka collision).

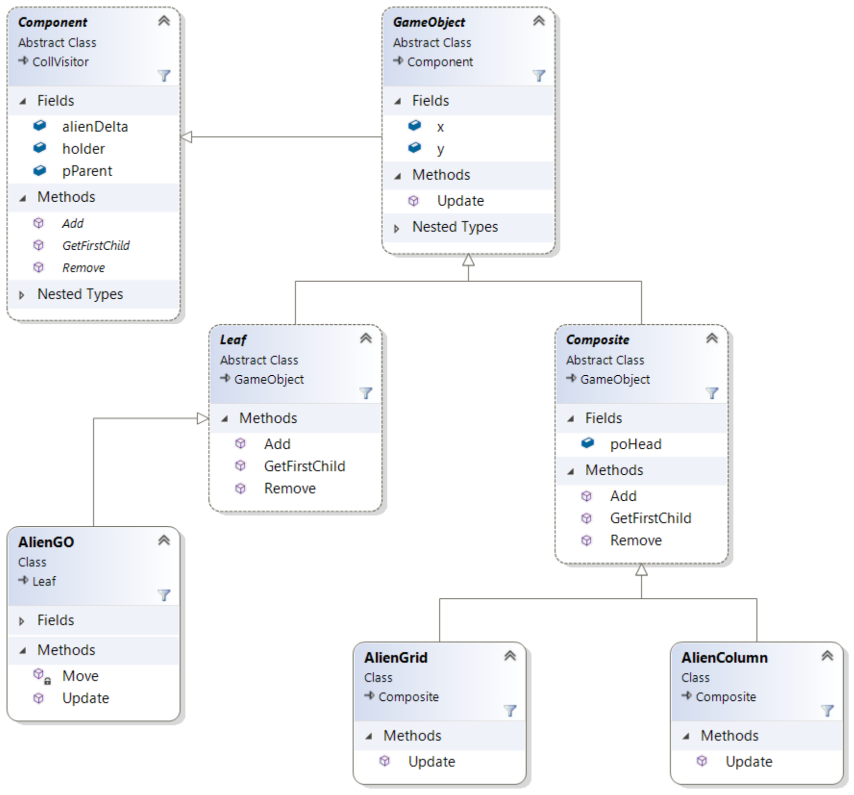
Composite Pattern

Overview

This pattern is a way to aggregate objects of the same type. Typically we can think of objects in a collection. This pattern takes the idea one step further and allows you to aggregate aggregates of those objects as well. The terms to keep in mind are *components*, *composites*, and *leaves*.

Leaves are the objects; *Composites* are the aggregates of the leaves; *Components* are an abstract class which both leaves and composites are derived. The idea is you create a composite type then add leaves and/or other composites which could potentially hold other leaves and other composites. This represents a tree structure of object composition.

UML



Internal Workings of the Composite

The Component class defines an abstract interface to be implemented by the Composite class and also serves as a parental abstract class to make implementation details easier. The Component class holds the type of component it is in holder: either leaf or composite and it also holds a reference to its parent. If a particular component is the root component, its parent should point to null.

The Leaves are not concerned with other leaves. In my diagram the Add, Get-First-Child, and Remove methods all assert null. This is because these methods exist in the abstract component interface. If you look in the UML diagram the concrete class of the leaf does not implement these.

The Composite class is where all of the magic happens. The composite class maintains a doubly-Linked list called poHead. The Get-First-Child method returns a reference to the head of this linked list. When the Add method is called we grab the head of the linked list and traverse down until we find the last node, then we add the new Component to the linked list. Although in the code this action is delegated to the D-Link class which has a static method Add-To-Back. The input is a component meaning it could either be a leaf or another composite. The remove method delegates its responsibility to the D-Link class which has its own method for removing D-Links called Remove-By-Address.

How I use the Composite in the Game

As mentioned early this pattern is used to organize the space invaders physically in the game. This pattern is also used to facilitate communication for how they should move. In the Loading Content phase I created an Alien Grid and 11 Alien Columns where each column had 5 aliens. I used a factory and a loop to generate Alien Game Objects (AlienGOs). After each object was created it was added to a column. When each column was filled with all 5 aliens I could then add the column to the grid.

In the update phase originally I had Move method as an abstract method which could be called from a composite which would loop through its children and call Move on each. This was later absorbed by the update method. Each leaf has its own move method which I made private. To have the aliens move left and right that is where the static field Alien Delta came into play. If the x-coordinate of an alien dipped below zero then I multiplied the Alien Delta by -1 and it incremented in the opposite direction. If the x-coordinate exceeded 800, my game width, then we multiplied by -1 moving it back to the left.

The move method is called from within the Update method. All of the Game Objects have a manager and Update is called on all of them each time the game refreshes. Since the composite Game Objects were just shells for the leaves, there was no private Move method being called in their Update().

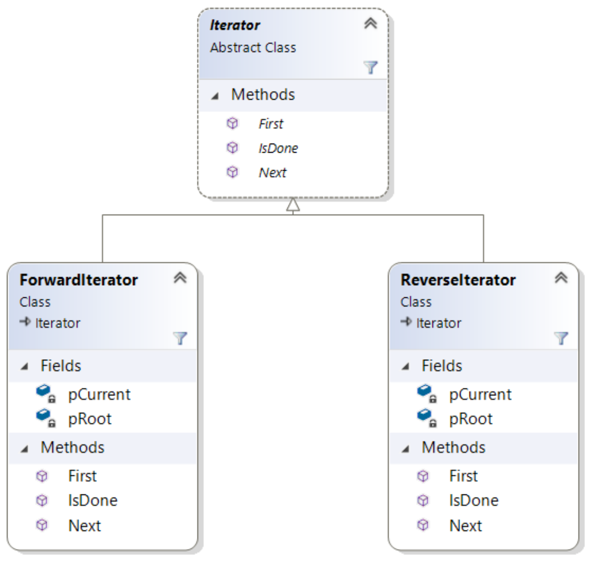
Iterator Pattern

Overview

We often want to perform the same task on a collection of objects but don't want to concern ourselves with the traversing of the collection for every single action. The iterator pattern encapsulates this behavior through a common interface that you can adapt to your specific collection and how it is organized.

Using the iterator we generically use a while loop. We start with the first element, perform the action, and move onto the next object. Each time around the loop we ask the iterator if it is finished and then the process stops.

UML



Internal Workings of the Iterator

These iterators were designed to traverse the composite pattern that was mentioned previously. This means that our iterators had to take on a tree-traversal behavior. For the sake of this program I implemented a depth-first approach which was okay because I had guarantees that the tree depth was small, 1 to 3 depending on the type of root element we were dealing with so I didn't have to worry about blowing out the call stack. So essentially you ask to the first child. Then you ask the first child if it has a child and dig deeper. When there are no more children look for a sibling. Recurse on all of the siblings. When there are no more siblings go back up the parent. When you find that you are back at the root element then you are done.

How I use the Iterator in the Game

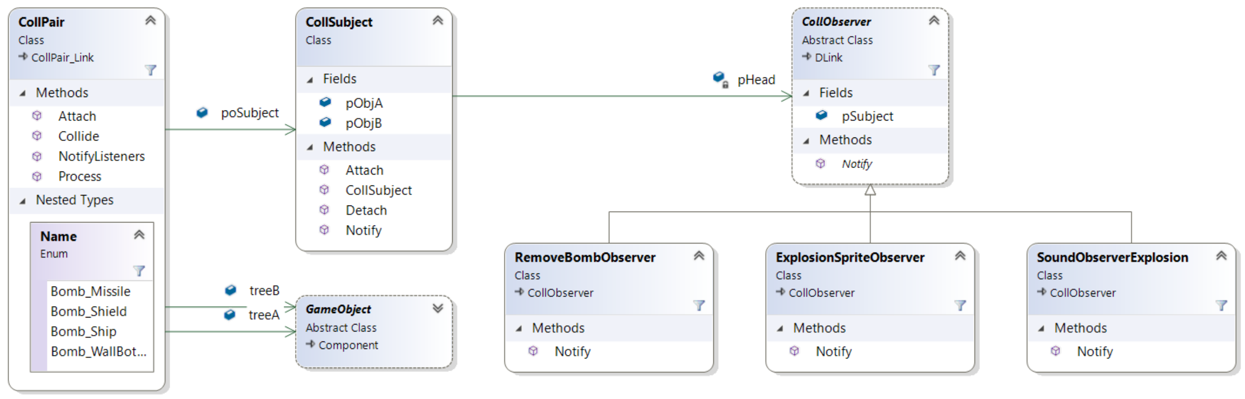
The reverse iterator was instrumental in making sure that Objects were removed correctly from their parents. It also managed the complexity of making sure that the bounding boxes of parent elements updated their size to reflect the current number of children that they had.

Observer Pattern

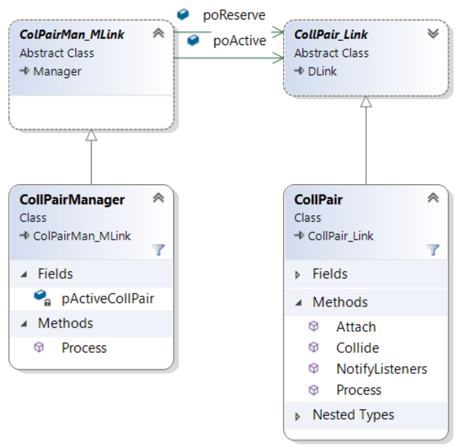
Overview

This pattern is a way to organize notifying object of specific events in the game and allows them to carry out any pre-defined set of actions you wish them to perform. In standard terms there are subjects and observers. Observers register themselves to subjects via the attach method. At some time in the future the subject deems that something important has happened and it calls notify which in turn calls notify on all of the observes who have been attached. Each Observer implements its own custom notify method. To properly do the work the Observers were assigned to do they have back pointer references to the subject so they can know its internal state and or how its state has changed.

UML



Collision Pair Manager



How I use the Observer in the Game

There is a Collision Pair manager that Processes every Collsion Pair that has been set up. When a Collision Pair is active it also notifies all of its listeners (aka Observers). These Notifications are set it motion by the visitor pattern which will be discussed next. On a given collision there are various flavors of observers that could be notified. We could remove and object from the field of play. We could play a sound. We can have objects move in predefined ways. We could have objects appear or disappear. I have Commands that get added to the Timer Manager for all of these observers and these types of interactions.

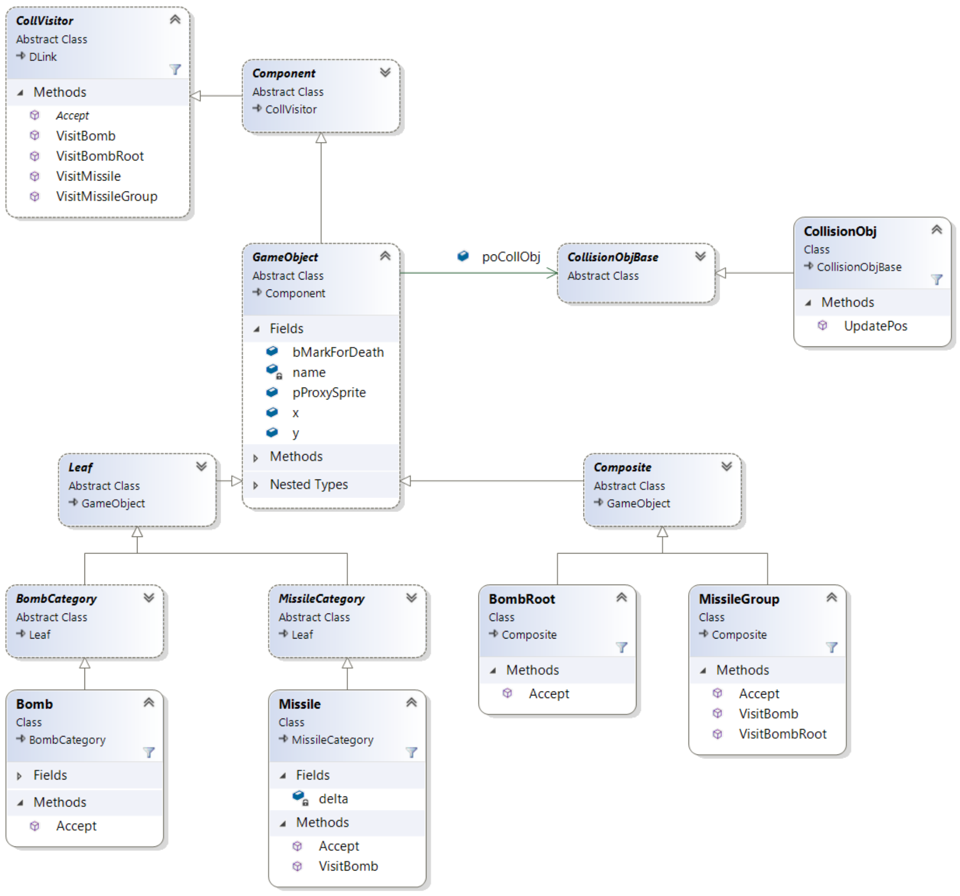
The advantage here is that I don't need to think of every collision as a unique interaction. I can separate the action that needs to be performed into broad categories, attach the kind of interaction that I need and configure some details in the constructor. For example with adding explosions to the screen I had one Observer and passed in the name of the Game Sprite that I wanted it to print to the page. The only thing the observer had to do was to pick the correct party (pObjA or pObjB) in the collision and print the Explosion Sprite to the same x and y coordinates. Originally I had multiple Sound Observers, but then I realized that I could just have a single sound observer and pass in the Name of the sound file I wanted the manager to find and play.

Visitor Pattern

Overview

The Visitor Pattern defines a abstract interface for communication between objects by passing references to each other through the signatures of the functions you choose to implement. There are two broad categories of methods: Accept methods and visit methods. The visit method accepts as its parameter the calling object type. This means that you have access to its scope. Also each class provides its own accept method which passes itself as a parameter to the abstract class for Visitors. What this means it that to establish communication you need only implement the visit methods of who you wish to interact with since you open yourself up to everyone else via your own accept method.

UML



Internal Workings of the Visitor

This system is synonymous with the Collision System. In order to reduce the number of interactions in the system there is a system wide organization principle or organizing collisions alphabetically. Along with the Collision Visitor being composed of virtual methods cuts down on the number of methods a given class needs to implement. Every Game Object has its own Accept method because it could be involved in a collision and points to the scope of the other partner in the collision.

The Game Objects are organized into many composite patterns of Game Object trees. The collision Pair Manager calls process on each Collision pair and figures out if any of the root Game Objects have intersected. If they have intersected their internal collide methods are called on the child of the second root element and you traverse calling collide down its children until you reach the bottom of that Game Object Tree. When this happens we go back to the first Game Object Tree and call collide on its children similarly until you reach the bottom of the first Game Object tree. When you bottom out you end up in the Visit Method of the second Game Object tree and this is what sets the collision pair to active and notifies the Collision Observers.

To make sure that the visiting methods line up it is important to understand how the visit methods related to where they fall in the composite pattern. The Visit methods of the alphabetically second Game Object are where all of the work is done. Each composite needs a visitor of the first trees composite class and the leaf class while the leaf in the second tree only needs a visit method for the first tree's leaf class.

How I use the Visitor in the Game

As touched on throughout the project this pattern is used exclusively by the collision system. It was quite confusing to get a handle on who needed which visit method until I looked from regularities along composite vs leaf and the requirements of the first tree vs. the second tree. The advantage of this pattern is that it disambiguates who is colliding with whom. This pattern along with the composite structure of Game Objects makes for a much more efficient search since we are asking parents first if they were involved and not involving any of their children if they are not.

Problem #5: Processing Game Events

Every 60Hz or roughly 16.6ms the Azul Game Engine runs through an update of the state of the game. Some of these processes (which we will call events) we want to process instantly while others we want to defer to a later execution time. These events include updating the positions of Game Objects, registering the Collisions, playing sounds, "placing" and "removing" objects on the screen, etc.

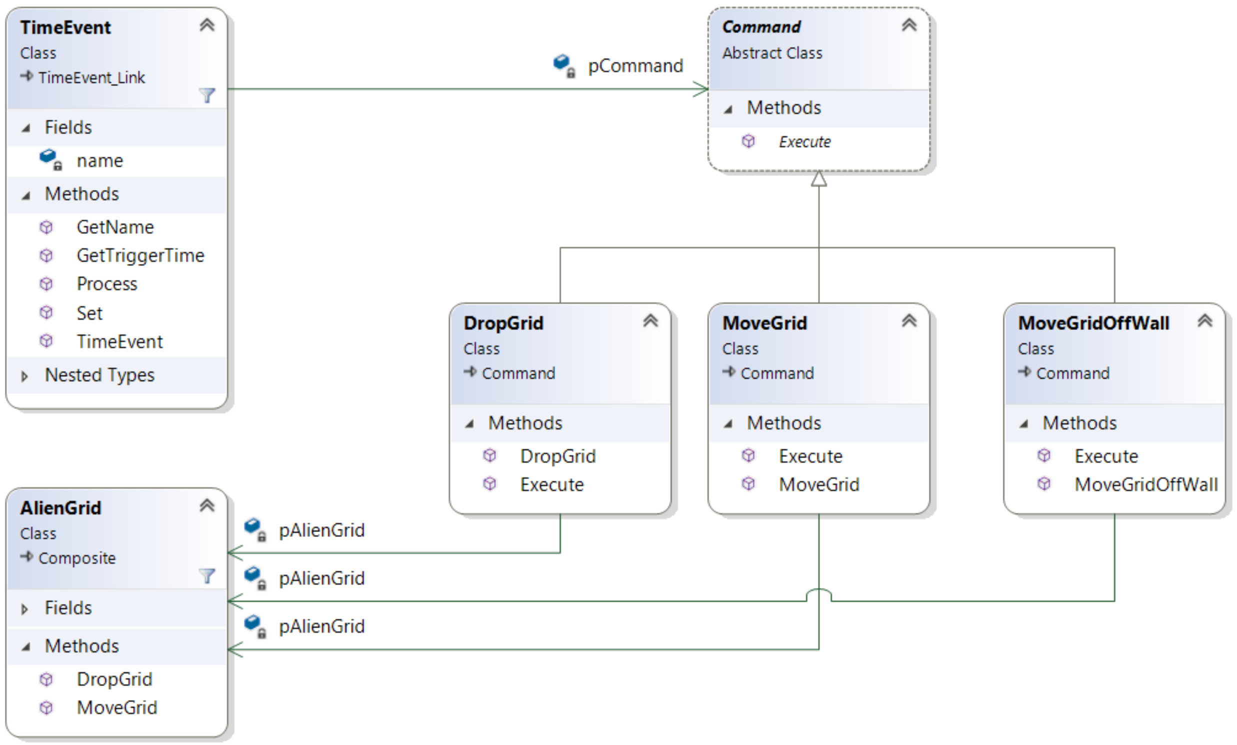
Unlike the managers that we talked about before there seems to be a much greater diversity of events that should take place. Events seem to be fairly heterogeneous when we think about what they affect but in fact they are homogeneous in terms of how they have should be processed in ordered time intervals. To get this "one size fits all" behavior we create what we call a Time Event. We encapsulate the diversity of behavior between Time Events by composing them with the Command Pattern.

Command Pattern

Overview

We define a common abstract class called command with an abstract interface method called Execute. When the program recognizes its time to perform a task, one client will invoke a new concrete command. When creating this new command we often need to supply it with some information. Perhaps some arguments for the initial conditions of the process and perhaps the intended client/receiver where the work will be done on. Depending on how many concrete commands you make you might not need to supply that much information.

UML



Internal Workings of the Command

There really is not much to say about this other than what was provided in the overview. The command pattern is really just a wrapper to package some preordained logic to be run and output to a specific client/receiver. The meat of what is done depends entirely on the application. You just create a new instance and call its Execute method.

How I use the Command in the Game

Now this is the more interesting part. In Space Invaders the command pattern is used heavily to process Time Events in the game. My UML diagram focuses on how I get the alien grid to move across the screen. I have it cycle through 3 different commands which either call themselves or call one other based on a Boolean in the alien grid set by the collision system. The grid starts off with a move command which keeps producing other move commands until the grid has registered that it hit a wall.

When this happens instead of issuing another move command I issue a move off wall command, flip the Boolean, and run a drop grid command. Then the drop grid command runs it then invokes another move command which starts the process all over again.

The command pattern is also used for having the UFO appear, move, and disappear from the game in a like manner that I just described: cycling through multiple commands. I also initially used this to have it play sounds at the same time that the grid moved but when I tried to make the grid speed up I realized I would need to engineer something slightly different to keep the move commands and sound commands in parallel.

Problem #6: Reacting to User Input

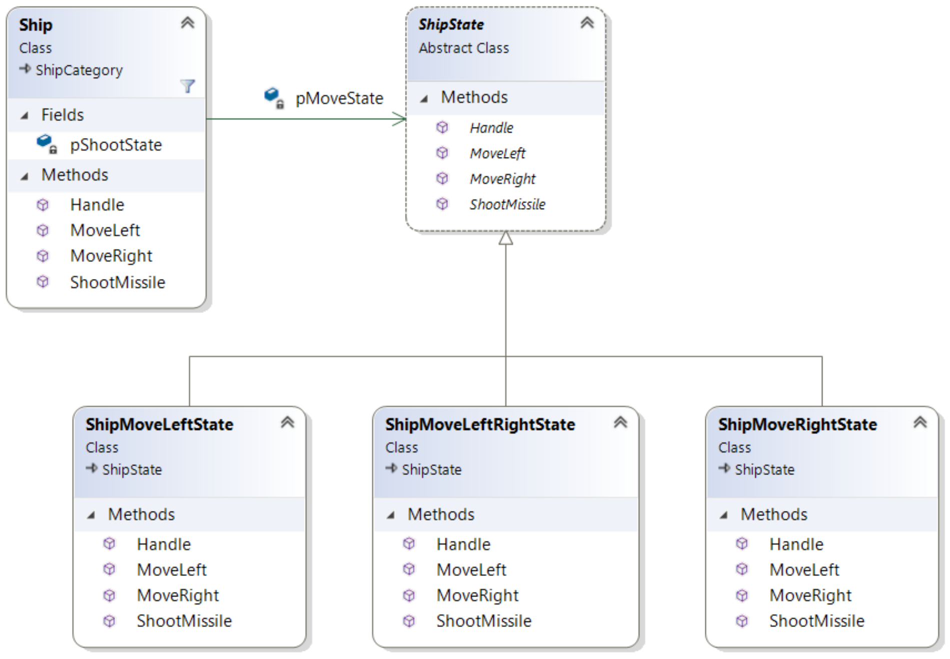
When we are talking about videogames this is one of the first problems that you think of. While the Azul Game Engine thank fully takes care of many of the fine-grained details of registering signals from the hardware, we need the system overall to responsive. This means that it needs to involve as little computation as possible. Essentially every user action should fire off a short task. The State Pattern gives us this responsiveness by associating the key inputs to move functions. We always want the keys to be live, hence the responsiveness, but depending on our state perhaps they are secretly disabled.

State Pattern

Overview

The state pattern provides an abstract interface with common methods at may or may not be unique depending on how you define you states. This allows us to focus on the behavior that we want to have occur and to eliminate the anti-pattern in programming of “check-then-act”. The state pattern will always perform actions uniformly but what they do will be subject to its state. The state pattern also has a method for migrating between states which is commonly called handle.

UML



How I use the State in the Game

In Space Invaders we need a way to control the ship in a fairly responsive manner. The ship moves but it cannot move off-screen. Also the ship shoots but it cannot fire another missile while there is still an active missile on the field. So I broke these features of the ship into two states: a moving state and a shooting state. Both of these states are indirectly handled by the collision system who have references back to the Ship. When the ships moves on the bottom of the screen there are two Game Object Bumpers which it can collide with. The ship starts out in the move left and right state. If it collides with the left wall the state is handled and set to the ship move right state whose move left function is void. That means if the player tries to move left nothing effectively happens.

However this presented an interesting issue because then I was trapped only moving right from that point forward. There wasn't another collision that would have sent me back to the move left and right state. To fix this in the ship's update method I had the ship always set its state to the left and write state. This worked since the processing of collision pairs happened after the Game Object Manager called its update. This means every time the game refreshed I had the option to move both ways but If I chose to move left again I could still be colliding with the wall and be blocked from moving in that direction.

#7: Miscellaneous

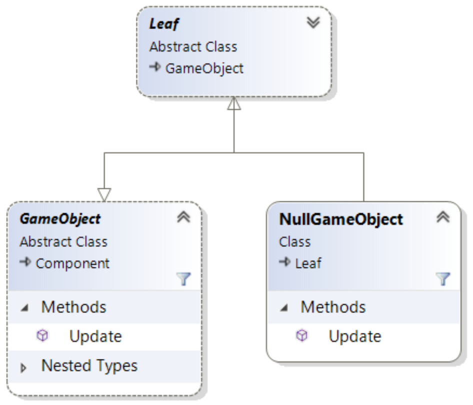
Some other patterns that were not at the heart of a component but used by them to function properly.

Null Object Pattern

Overview

The Null Object is an object of the class with no information and exposes no interesting behavior.

UML



How I use the Null Object in the Game

When it came to the composite pattern, all elements were game objects but they didn't all function like full meaty Game Objects. The Alien Columns and Alien Grids didn't have any Game Sprite associated with their rendering so it became clear that they needed a null type, A Null Game Sprite. In the diagram above we see a Null Game Object which functions similarly. If its update method is called noting happens, the method is blank. Since none of this fields are set this is perfect. It does nothing and does not break any pre-existing contracts.

Areas for Future Development

This project was challenging and rewarding at the same time. When I think about areas for future development my mind first goes to the features which were not fully implemented. The first being the recycling of Game Objects. Essentially every time a missile is fired, bomb dropped, or alien invader swarm repopulated these are new objects. The struggle for me involved untangling all of the managers that Game Objects are associate with. I did however achieve Game Sprite recycling with the Explosion Sprite class. There I only had to make sure to remove them from the Sprite Batch Manager so fewer references to untangle. The Explosion Sprite Factor has its own internal manager. When the Explosion event occurs you search for the explosion on the manager and print it to the screen at the location of one of the collision objects. When about a half a second goes by the explosion sprite is removed from the batch. Each explosion Sprite has an internal proxy sprite so that is how there can be more than one on the screen at a time.

Also I did not implement a scene manager and that is the second biggest area of improvement for the project. I essentially made the scene context a global variable and can then call a set scene state function from anywhere at any time. Very dirty. Because of this oversight the user can jump to any scene by mashing on the numeric keys.

The last large area of development which I will consider to be a non-bug, is getting the Timer Manager to transition between scenes smoothly. There is this problem where upon losing a game and entering a new game, bombs seem to drop out of thin air. I would guess from the locations of the previous alien grid from the previous game. I did some debugging and saw that this is the case. That even when the new game loads the hash code of the old drop bomb command creeps into the new scene.

I also have a bug with updating the high score. When you lose a game the high score does not appear on the select page, however when you enter a new level, the high score is correctly displayed in the middle of the screen. Somehow I thought that using the same font name on both screens would mean they update together but this is clearly not the case. I probably need to give it a new font name and update the new font in the entering or leaving method of one of the scenes.