2020

Space Invaders

C#

NICK SCHORER

Introduction

Our goal is to recreate Space Invaders from scratch. Other than using the Azul engine, which provides us an API for drawing on the screen, we are coding the entire game. In this document, we are going to outline a series of the problems we expect to encounter while developing this application and our approach to solving them. Ideally, we want to write code that:

- Avoids memory spikes and has smooth performance. There should be no stuttering or noticeable load times. New and delete are our enemies so we want to limit how much we call those.
- Is modular and reusable as much as possible. We want to reduce coupling so that we can move code around as much as possible without causing unexpected behaviors.
- Is easy to change as we update our requirements. When we are several weeks into development, we shouldn't need to go back and revise code from our first week. Every time we tackle new problems, we should build on top of what we have previously done, because changing old code is more likely to introduce new bugs.
- Is as easy to understand as possible from a high level. There are going to be many classes in this project. We should at least be able to group them into categories such as Colllision, GameObjects, Input, Sprites… so that it is clear how to update code later on. Each object should have a clear function and place within the overall game hierarchy.

Problem

Our game is a real time application, which means we want as little stuttering as possible. We shouldn't have any noticeable 'loading' times and the whole experience should feel buttery smooth to the user. One of our main objectives is to cut down on the number of dynamic allocations and deletions that happen during the course of the game, since these are a huge performance hit. This is a tricky goal because we are constantly deleting and respawning objects in our game. Ideally, when we remove our objects from the game, we would like to

keep holding onto that memory so we could turn around and use it again later when we need to respawn those objects.

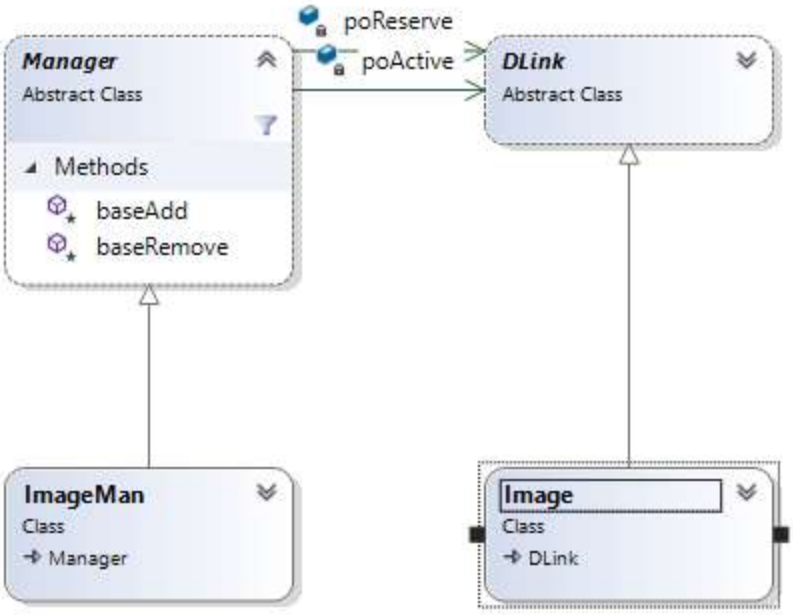
Solution

We will use define Manager objects that will be used to track which objects are active (and therefore should have a visible impact on the game) and which ones are on reserve (pre-initialized chunks of memory). Whenever we want a new object, we will first check if we have a pre-initialized object in our reserve. If we do, then we will simply sterilize it and then reuse it without having to dynamically allocate any new memory.

This approach is useful because we are going to be managing a lot of objects, which means potentially a lot of allocations and deletions if we aren't careful. Further, while the number of objects we are managing is fairly large, it is also predictable – there is a defined max number of concurrent active objects. Once our pool reaches the size of that threshold, we won't need to do any more dynamic allocations.

Object Pools Applied to Space Invaders

We will use these Object Pool managers as a primary data structure across the entire application. We will use them to store all of our assets (images, textures) as well as our game objects (aliens, the ship, missiles/bombs). In the case of the enemies, we know there will only ever be 55 at a time, so that means we are saving 55 allocations every time we start a new level. Once the player makes it through the first level, memory usage should be pretty much static.



Our abstract manager class will be used to store a list of active and reserve nodes. Our concrete managers, such as ImageMan, will leave all of the data structure implementation to the Manager while they focus on their class specific elements. When the client wants to load a new image, it will ask the ImageMan to store it, and the ImageMan in turn will rely on the Manager to allocate active and reserve nodes as needed.

Problem

We will have several managers that are used for accessing and manipulating data. These managers need to be accessible across the program. However, we don't want to just pass these managers all over the place through different constructors and functions since that will be a lot of overhead and is likely to get in the way of the code that we want to write. Plus, if we, for example, have a function that is used heavily across our application, and then later we decide we need to pass in a new manager, then that would mean updating all of those function calls, which is a real pain. Our goal is to make our managers globally available, but we still want to

have some encapsulation so that we can control how our calling code interacts with those managers.

Solution

We will have our managers use the Singleton pattern. Essentially, in a singleton pattern we make a bunch of methods public and static, which the consuming code uses to interact with the singleton globally. But within the singleton class, we have a pointer to a single instance of the class, which allows us to keep some encapsulation. So users can interact with the manager across the program, without a specific reference to the class, but we can still control how the user is allowed to interact with the class.

Singleton Applied to Space Invaders

Pretty much all of our manager classes (the ones using objects pools) will incorporate the singleton pattern. We want them to be globally accessible across the program, and for each type of class, we only want one manager so that we know all the data will be in that one structure.

In the above example, you can see how the GameObjectMan is a non-static class but it has a static reference to the active instance of itself.

Problem

Although our game will have many objects, a large percentage of these objects will be duplicates. If we are talking about the enemy grid, we will have 22 octopi, 22 crabs, and 11 squids. The only difference between each of these individual enemy units is their x and y coordinates. That means each object is carrying a lot of redundant information, namely the Azul.Rectangle that describes its location and the Image object that is associated with the game sprite. We want to be able to encode the constant stuff in one class and the changing stuff in another class, so that we aren't duplicating anything.

Solution

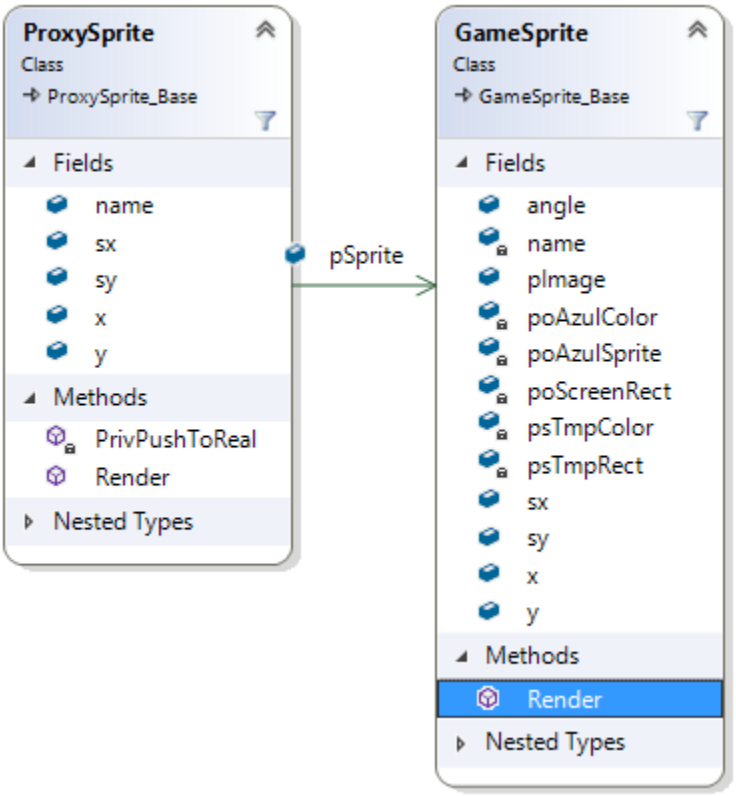
We will use Proxies in place of our game sprites. A proxy is basically a wrapper that is used to substitute for the real thing. Proxies are lighter weight than what they are substituting for, only containing the information that is changing and a pointer to the original class. When it is time to use the proxy, we push the data from the proxy to the real object and then perform whatever functions are needed. We repeat the process for each proxy.

This pattern is useful to us because GameSprites are fairly bigger than our proposed proxy sprites, which only need to store two floats and a pointer. It will allow us to shrink the footprint of our program and get more cache hits.

Proxy Applied to Space Invaders

We will create ProxySprites as a substitute for GameSprites. After loading one of each GameSprite at the start of the application, we will no longer interact directly with the GameSprites. Instead we will create and interact with the proxies. After establishing a link between the proxy and the GameSprite, the only thing we need to change on the proxy is its x and y coordinates. We will create a proxy for each instance of an object in the game, while

having only one of the original GameSprite. So, instead of having 22 GameSprites of the Crab, we will have 1 GameSprite of the Crab and 22 ProxySprites. This is good because ProxySprites take up less space. In the background, when we tell the ProxySprite to render, it will push its data onto the corresponding GameSprite and then use the GameSprite to draw to the screen.



When the ProxySprite goes to render, it will push its data onto the GameSprite, which will in turn call its own render function.

Problem

In our main ScenePlay class, where the high-level flow of the game is happening, we already have a lot to deal with. We don't want to get bogged down with the specific details in our scene class; rather, we would like to have another class abstract out those parts for us. Specifically, whenever we create a single Shield object, that actually involves creating around 50 smaller

brick objects in a carefully organized grid. The scene shouldn't be concerned with how those smaller bricks are organized. Plus, if down the line we decide we want to change how those shields are laid out, we shouldn't have to change the scene class in order to do that. We should have a separate class in charge of managing how we produce shields in our scene.

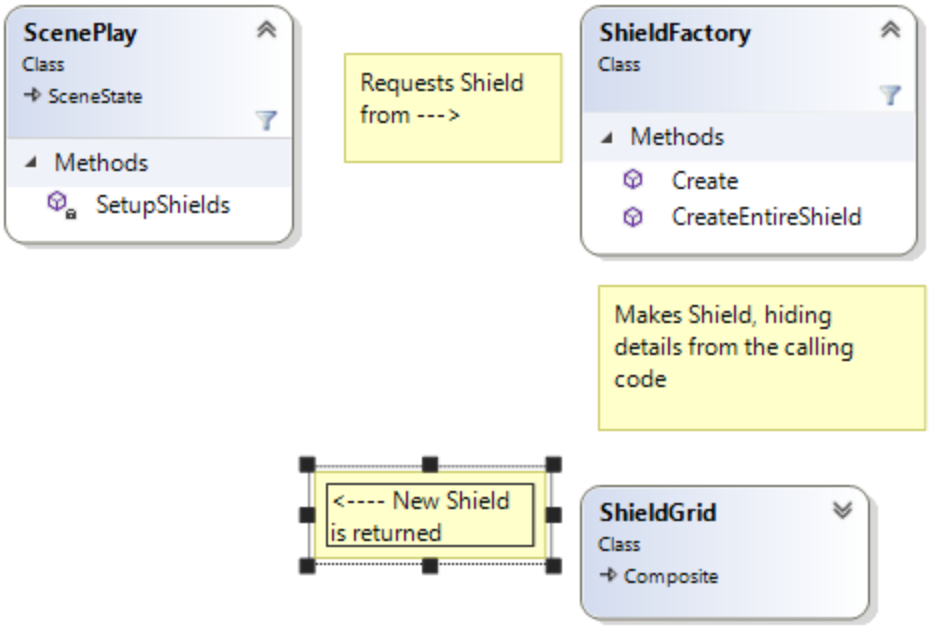
Solution

We create a Shield Factory that is in charge of generating Shields for our scene to use. A Factory is a design pattern that creates a new instance of an object while hiding the creation logic from the calling code. The client simply asks the factory to give it *something*, usually providing minimal parameters, and the factory decides on how to specifically create it.

This pattern is useful because creating Shields tends to be cumbersome – it's not that it's hard, it just takes a lot of careful ordering of code that the scene code really shouldn't have to manage. The scene just wants a shield.

Factory Applied to Space Invaders

We will create a ShieldFactory class that will act as the middleman between the scene class and each of the shield brick objects. The scene class will be able to ask the ShieldFactory for a full Shield, and the factory will get to work stack all of the bricks in the right order. If later in the project we decide that we want a different structure to our shield, then we only need to modify the ShieldFactory and not the scene class.



Problem

Our game is going to have a lot of objects and that means a lot of potential collisions that we need to check each frame. When we fire a missile, it will be too inefficient to individually check every enemy to see if the missile collided with it. We want to be able to accurately register when objects collide with each other while keeping performance strong. In the case of our game, we know our aliens are going to be rigidly structured in rows and columns. Rather than checking each enemy individually, we want to be able to check an entire column at a time. If and only if the missile collides with the column, then we check each of the aliens that make up that column. This way, we can cut down on the number of collision checks we need to do each frame.

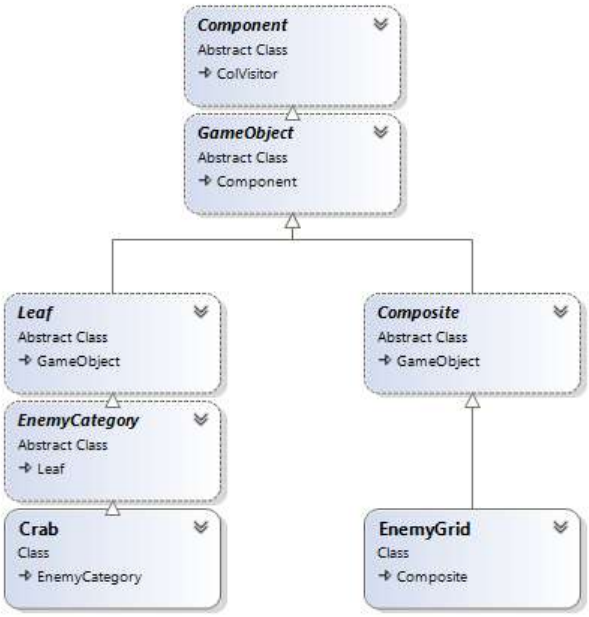
Solution

We will structure our EnemyGrid using a Composite pattern. The Composite pattern organizes objects into a tree structure consisting of leaves and containers. Further, it allows us to treat each node (leaf or container) uniformly.

This is useful to us because we want to organize our enemy grid into leaves and containers (individual enemies are leaves, columns of enemies are containers) so that we can check collisions. But we don't want to need to add in special logic for handling leaves and containers separately.

Composite Applied to Space Invaders

We will organize our game objects into composite collections. For the enemy grid, we will have one grid that contains multiple columns that contains multiple enemies. The grid, the column, and the enemy will all be considered a *Component*, so we can treat them uniformly. Each frame, we will check if a missile collided with the grid. If there was no collision then we are done – no need to check the remaining 55 enemies if we didn't even collide with the big box. If there was a collision, then we go to the next level and check each of the columns, and so on. Although I have primarily been using the enemy grid as the example, we will apply the composite structure to all of our game objects.



Problem

The Composite pattern is useful for reducing collision checks, but it's lot less straightforward to traverse than a linked list. The client code that uses the composite code is going to need to know the exact order on how to get through the list. Which node is first? Which node is last? Are we done traversing? Ideally, the client code shouldn't need to know the implementation of the underlying data structure. From the client's point of view, it just wants to get through the group of objects, whether that is an array, or a tree, or whatever. Plus, if we change our data structure later on, that means we would need to change the client too, which would be problematic. The client should be able to traverse the composite without knowing its implementation.

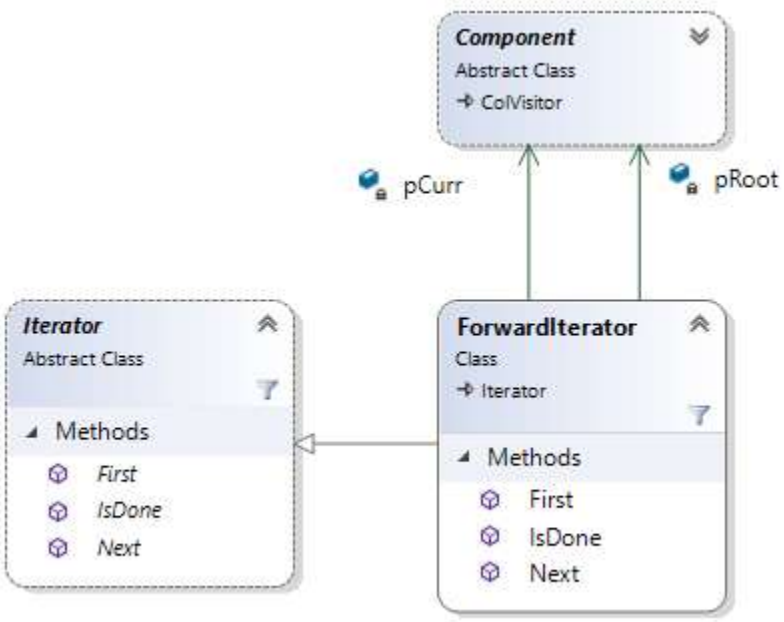
Solution

We will use the Iterator pattern to traverse the Composite structure. In the Iterator pattern, all of your data structures inherit from an abstract class called an Iterator. An Iterator has functions that define where the start of the structure is, how to get the next element in the structure, and whether we have reached the end of the structure. You could have several different data structures in your program, but as long as they all inherit from Iterator, your client code can iterate on them uniformly, using their inherited methods.

This is useful to us because the Composite structure is not obvious. The class that knows the Composite structure the best is the Composite itself. So, let the Composite define how to traverse itself, and then have the client code use the Composite's iterator to traverse it.

Iterator Applied to Space Invaders

We will define an Iterator class with First(), Next(), and IsDone() methods. That way, throughout our program whenever a class needs to traverse the composite, it just needs to call those three methods. Furthermore, if we want to traverse the composite in a different way, instead of redefining our existing iterator, we can simply create a new derived Iterator class. One place in code where we will need to traverse our composite is when we are calculating the collision boxes for each of the components.



Problem

So now we can use composites to detect when collision happened. How do we react to these collisions? One approach is to have each object check whether it is colliding with another particular object on every frame… but this is obviously inefficient. It would be nice if we could instead have something tell us when a collision happened. That way, instead of constantly checking to see if something is colliding, we simply respond when we are alerted to a collision.

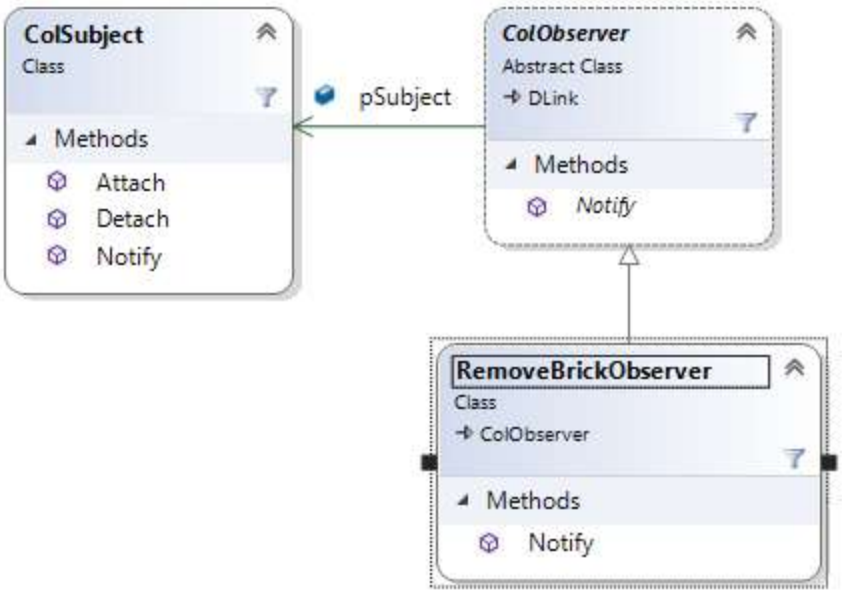
Solution

We will respond to collisions using the Observer pattern. With the Observer pattern, we 'subscribe' to events happening the same way we subscribe to a magazine. The observer object tells the subject that it wants to be alerted when something happens. The subject's sole job is to look out for this something happens, and the moment it does it notifies the observer object. Once the observer is notified, it can respond however it wants.

This is useful to us because we have many events that we want to trigger after certain collisions happen. Rather than having each of those events track whether the collision has happened, they will simply listen to the subject whose sole job is to watch out for that collision.

Observer Applied to Space Invaders

We are going to have a lot of collision pairs. Just to name a few: missile vs alien, missile vs shield, missile vs wall, bomb vs shield, bomb vs player, player vs wall, enemy vs wall… and so on. Each of these collisions will spawn one or more reaction events. Each of these reaction events is going to be an observer in our game. Some of these observers will react to multiple types of collisions. For example, one of our observers will delete a shield brick. This shield deletion will happen when a missile collides with the shield OR when a bomb collides with the shield OR when an enemy collides with a shield. By having our collision subjects separated from our reaction observers, we have a lot more flexibility in how we can structure our code.



Problem

Suppose two of our game objects, a missile and a squid, collide. Did the missile collide with the squid, or did the squid collide with the missile? Our game needs a way to definitive way to decide in what order the collision happened in order to process it. We need a way to specifically define how these collisions happen. We already have a Composite structure for our enemies and missiles, and we would like to be able to build on top of that instead of having to rewrite each of our composite classes to individually handle collision.

Solution

We will use the Visitor pattern to redefine our collidable GameObjects as objects who can *visit* and can be *visited*. The Visitor pattern uses a centralized abstract Visitor class to define which concrete classes can visit other classes by using an inherited Visit() method. Specifically, Visitors

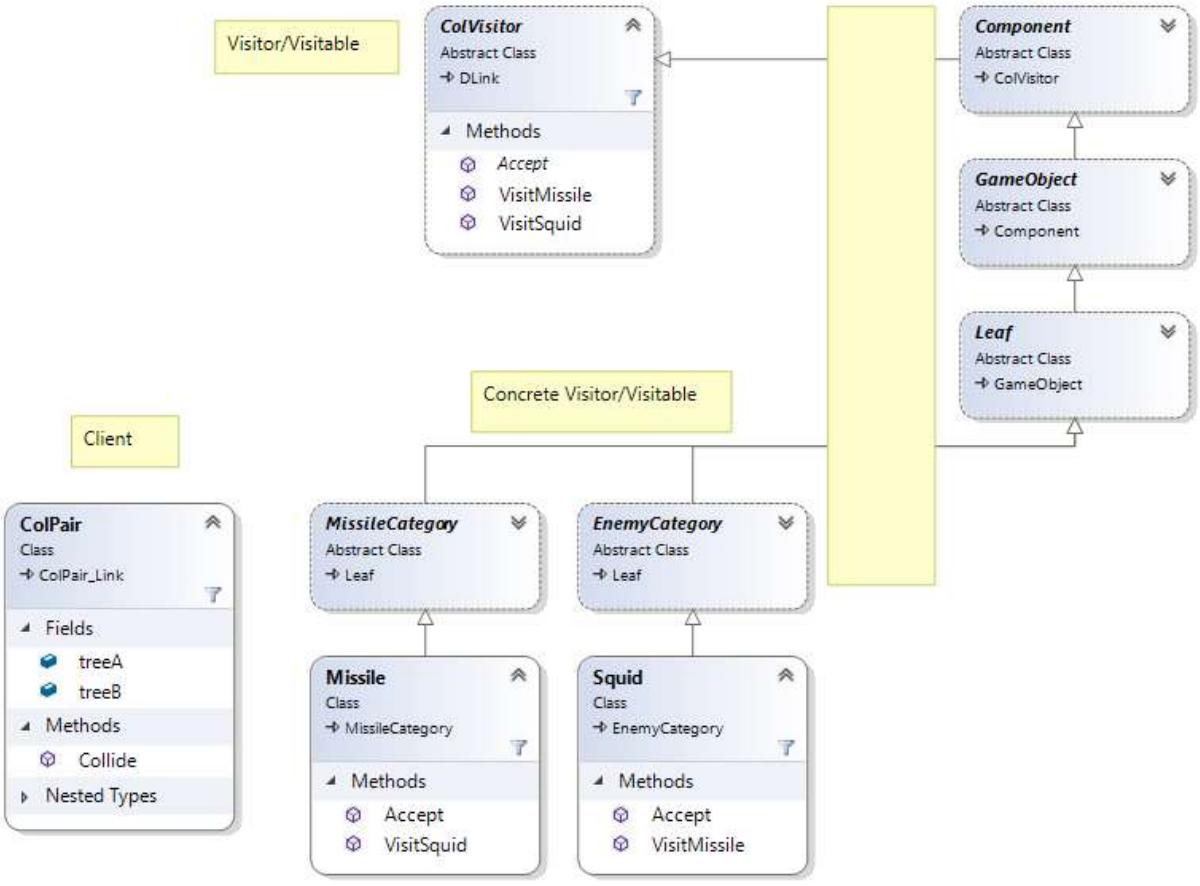
can visit classes that inherit from an abstract Visitable class which as an Accept() method. Any new object that wants to be visitable must add a new abstract method to Visitor to let other classes know it can be visited. Finally, there is a client object that knows how to traverse the composite structure and helps 'guide' the visitor to each of the visitable classes.

This is useful to us for a few reasons. By clearly defining a visitor/visitable relationship, it is easy for us to decide which object 'visits' the other when a collision happens. This prevents us from having to write double the amount of collision code and from worrying about what happens if objects try to collide twice (A collided with B *and* B collided with A). The Visitor pattern can be written on top of our composite pattern easily, so we don't need to go back and modify our composite code to work with the visitor pattern. Mid-development, we won't necessarily have all of the collisions figured out in advance and will need to add some of them later on as we discover them. With the visitor pattern, it's easy to add new collision cases because all of our code is centralized in the abstract visitor class.

Visitor Applied to Space Invaders

We will combine our Visitor/Visitable interfaces into one abstract class called *ColVisitor*. We want the flexibility to say that objects can either be the visitor or visited depending on what is happening in the program. All of our collidable GameObjects will inherit from this class and implement the Accept() method to let other GameObjects know that they can collide. Each GameObject is responsible for defining a Visit____() method for each object it may collide with. It *could* define it for every other Visitor object, but that would be a lot of unnecessary work since there are many object pairs that will never collide (e.g., the ship will never collide with the ceiling). If we forget to define a Visit() case for a particular object collision, that case will fall through to the abstract Visitor class where it will throw an error in Debug mode. That way, we can simply catch the missed cases during testing. In order to trigger the collision, we will have a collision pair object (*ColPair*) which will detect when the two objects are overlapping and it will call Collide(treeA, treeB), which will subsequently call treeA.Accept(treeB) to trigger the collision event.

Let's apply this pattern to our specific example from earlier: a missile and squid collide. We will have previously defined a collision pair for the two objects such that one corresponds to treeA and the other to treeB. Let's say missile is treeA. This will lead to the call treeA.accept(treeB), which will then lead to the call for treeB.VisitMissile(). The treeB member stands for a generic GameObject, but during code execution it will check its derived classes to see if VisitMissile() is defined. Since our Squid class defines VisitMissile(), that method will be called: the squid visits the missile and we can now define our collision.



Problem

Sometimes in our game we will need to cue actions that need to be delayed or actions that need to happen at a steady interval. Here are some examples:

- The enemy grid needs to march across the arena, not in a smooth continuous manner, but with discrete steps.

- When an alien gets blasted by a laser, we want to briefly display an explosion image before deleting that image.
- The same thing for the ship getting hit by a bomb, but we also want to set it up to respawn a few seconds later (if there are available lives)

These actions have different inputs and different outputs, yet we want to try them all uniformly as actions.

Solution

We will schedule timed events using the Command pattern. The command pattern has an abstract command class with an abstract method Execute. If we want to create new commands, we create a new class that inherits from the Command class. In the class's Execute method, we define whatever action we want to be taken. While Execute() has to have the same signature as the base class, you can pass in different parameters via the constructor, allowing you to have very different commands that can all be treated uniformly.

This is useful to us because we want to have a collection of timed events, where each event can be quite different.

Command Applied to Space Invaders

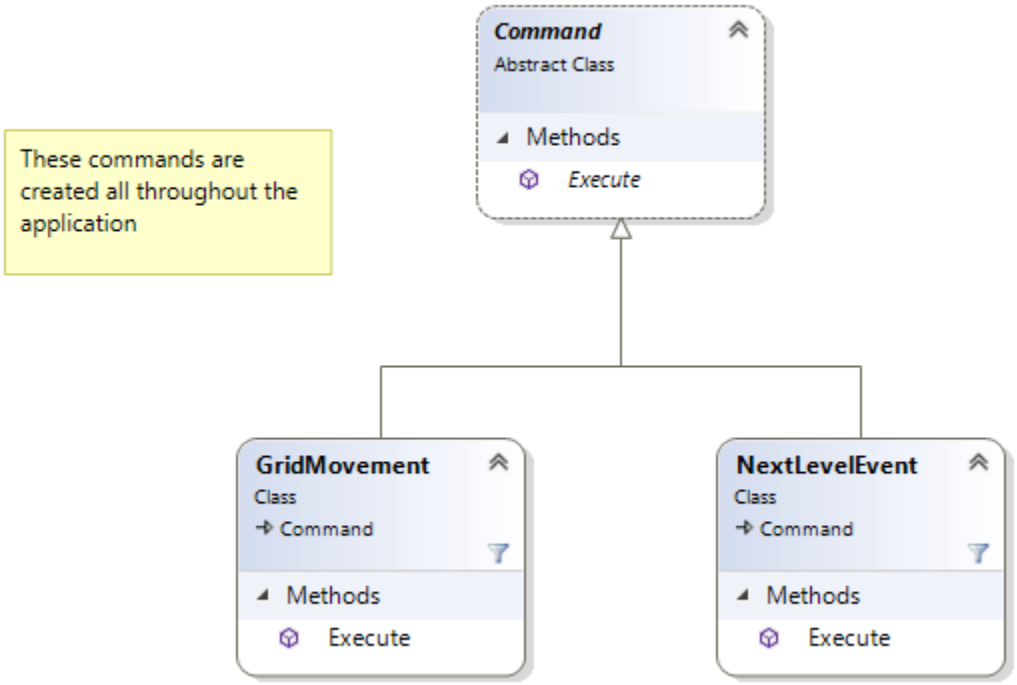
We will create a TimerManager which will handle dealing with elapsed program time. TimerManager will have an Add() method which will have parameters to

- Describe what action will be taken
- How many seconds later it will be executed

The action to be taken will inherit from a base Command class. Some commands will pass in the EnemyGrid, some will pass in the player – it doesn't matter to the TimerManager because they are all commands. There will be many time-based commands:

- Spawn a bomb
- Start the next level

- Switch to player 2
- Go back to the select screen
- Move the enemy grid



Problem

Our spaceship will have the ability to shoot missiles. Missiles are triggered by the player pressing the spacebar. But we don't always want to shoot missiles. If there is already a live missile that hasn't blown up yet, the user shouldn't be allowed to fire a new missile. If the user is dead, they shouldn't be allowed to fire a new missile. This means that somewhere between the code where we detect the user pressing the space key and the code where we create a new missile, we need to check those conditions before 'allowing' the user to fire a missile. But it isn't obvious where these checks would go. And if we start doing a bunch of 'if' checks, we are probably going to have an edge case bite us. We want to remove edge cases by removing 'if' checks.

Solution

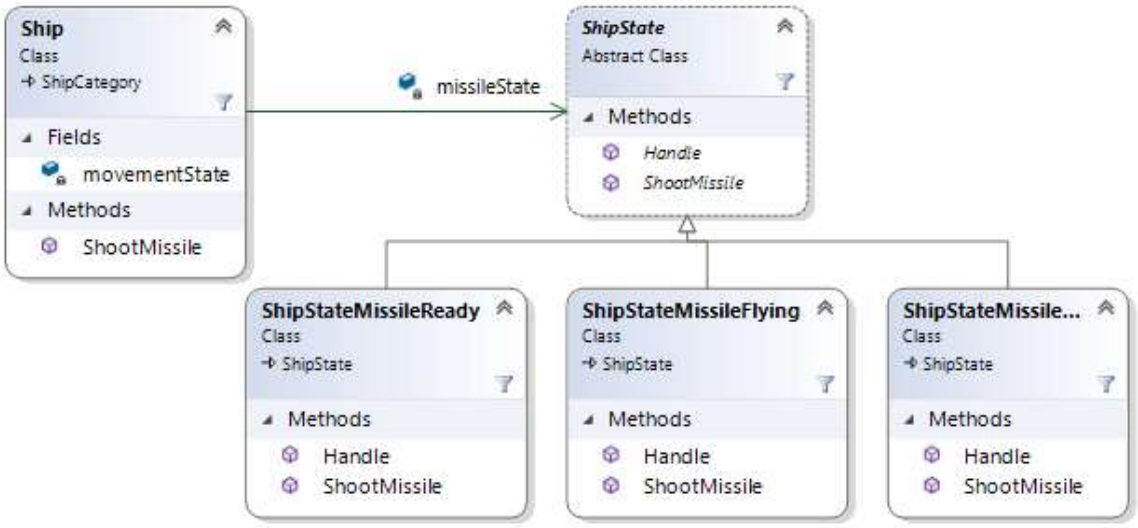
We will implement shooting using the State pattern. In the State pattern, the object maintains its own internal state. An object may have many different states, each one responding differently to different actions. These states are implemented as different derived classes inheriting from an abstract base class which defines which methods the state must be ready to handle. The implementation of the methods may be trivial.

This is useful to us because we don't have to worry about 'if' checks and edge conditions. When the user clicks spacebar, it will always trigger the ship's Shoot() method. But what Shoot() does depends on the current state of the ship. So, depending on the state of the ship, Shoot() may not do anything.

State Applied to Space Invaders

The Ship will have a pointer to a MissileState class. There will be three classes that derive from MissileState:

- MissileStateReady
    - Shoot() will create a new missile object.
- MissileStateFlying
    - Shoot() will not create a new missile object. If Handle() is called while in this state (when the missile collides with an object), then it will switch states to MissileStateReady.
- MissileStateNone
    - Shoot() will not create a new missile object. Handle() will not do anything either. The only way to escape this state is for an outside class to reset its state. This is useful for when the player ship is dead and you want to make sure it can't shoot missiles.



Problem

There are instances when we want to use data that is partially undefined. Namely, if we want to find a GameObject by name, we need to compare it against another dummy GameObject. This dummy GameObject wouldn't have any data – no proxy sprite, no collision box – just a name. Since our dummy game object is missing a lot of data, we would have to put in special 'if' checks to make sure it can get through our lower-level null checks. It would be nice if we didn't have to modify our null checks just to accept an object that wasn't supposed to do anything anyway.

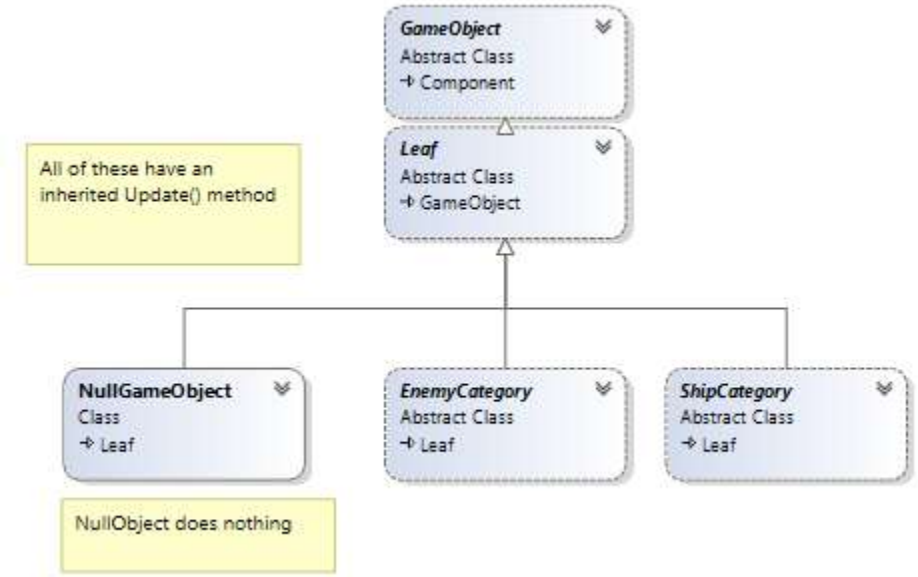
Solution

We will incorporate Null Objects into our GameObject code. Null Objects can be used as a substitute for real objects when the null object isn't expected to do anything. Calling code will treat Null Objects the same as real objects, keeping the code uniform.

This is useful to us because we really just need a dummy object so we can check if the names of two objects are the same.

Null Object Applied to Space Invaders

Not much to say here. Sometimes we need to find GameObjects. We search for them by name. In order to find an object with that name, we need to compare it to another dummy game object that has the same name. Using a NullObject makes it so we don't have to rewrite our low-level comparison code.



Problem

In our game, enemies can drop multiple types of bombs. These bombs share a lot of the same information: dimensions, speed, damage. What makes them different is their respective falling animations: one flips vertically, one flips horizontally, and one doesn't animate at all. The implementation for these is complicated enough that we can't just bake them all into the same class. But it also seems like a waste to create three separate classes for objects that already share much of the same behavior.

Solution

We will implement our bombs with the Strategy pattern. In the Strategy pattern, we define a behavior class with a default implementation. Derived classes can choose to override some or all of the base implementation.

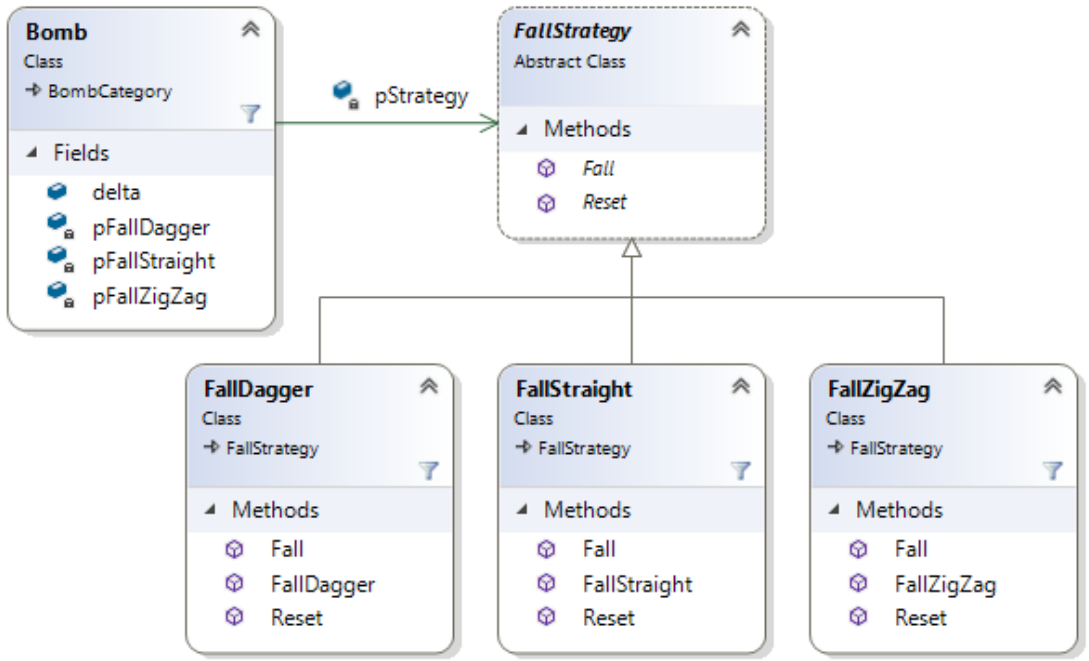
This is useful to us because we want the three bombs to be mostly the same, except they should each have their own falling behavior.

Strategy Applied to Space Invaders

There will be one Bomb class. The Bomb class will have a pointer to a FallingBehavior. FallingBehavior will define a Fall() function which specifies how the Bomb should fall. We will create three classes that derive from FallingBehavior:

- FallStraight
- FallDagger (vertical flip)
- FallZigzag (horizontal flip)

Each class will override Fall() to get the desired behavior.



Problem

We will have text being displayed on our game screen:

- Player Scores
- High Score
- Attract Mode instructions
- Game Over

These texts will be displayed using specific fonts designed for the game. In order to catalogue these fonts, we store them as Glyphs. We read in a texture file that contains every character for that font, and then we look up the letter we need. Once we look up a letter, we shouldn't have to look it up again or else we are doing extra work. If we have the text "Game Over", we have two e's. Rather than create two separate 'full' versions of the text for 'e'. We should create one 'full' version, and then two lightweight versions which just reference the full version.

Solution

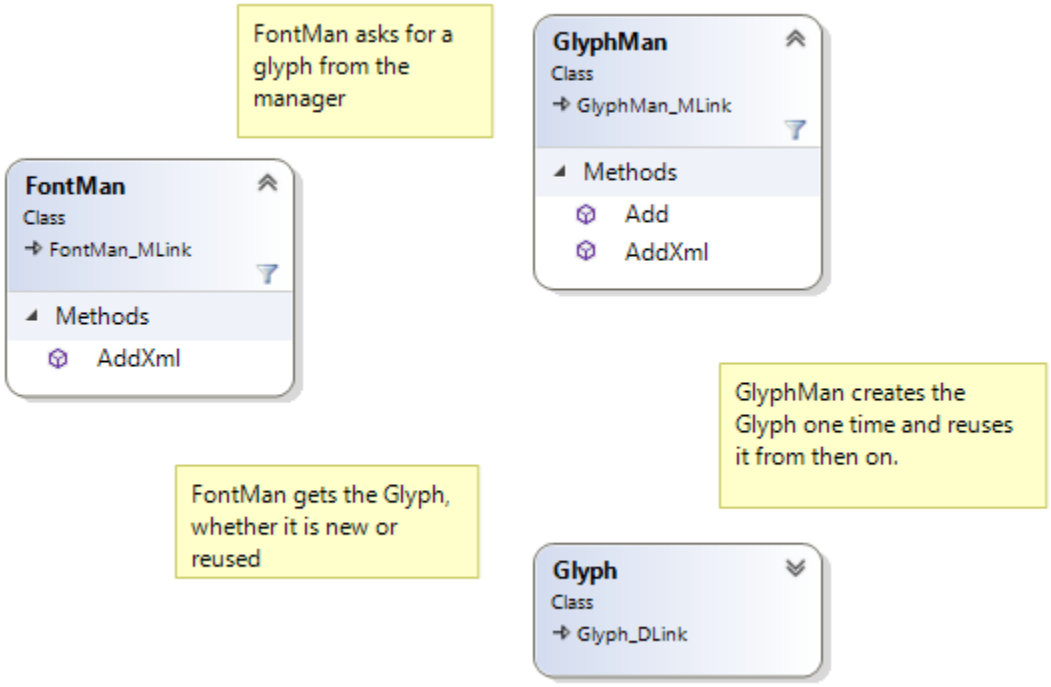
We will implement our FontManager using the Flyweight pattern. The Flyweight pattern is similar to the proxy pattern in that involves substituting many heavyweight copies of data with one heavyweight copy and many lightweight copies. The difference is that Proxies push their data onto the heavyweight copy, whereas Flyweights pull their data from the heavyweight copy.

Flyweight Applied to Space Invaders

Every time we look up a letter to add to the screen, we will first check to see if we already looked that letter up before. If we did, we'll just use what we looked up before. This will save us from having to do the work every time.

Additionally, when we remove GameObjects from our scene (enemy or player dies, UFO goes off the side, missile hits a wall, etc), instead of freeing that memory, we will store inactive objects in a RecycledObjectManager. This manager will keep references to old objects that we know will be used again. For example, when a wave respawns and we create 55 new enemies, instead of new-ing all 55 of

those, we will check our RecycledObjectManager to see if we already have them. If we do, we will simply use those objects again with the new appropriate x and y values.



Closing Thoughts

By using these design patterns to tackle our problems, we are introducing more complexity at the beginning of the design process in exchange for less complexity later on. At the beginning, it is much easier to change our approach or refactor code that only kind of works. By dealing with tough problems earlier on, our code will be more stable later when we have a better idea of what our requirements are and may need to change things around. When we can separate the parts of the design that are changing from the parts that are settled, not only are we writing safer code, but we are also writing less code, which is easier to maintain. With all that said, sometimes it's easier to just hack in some temporary variables with lots of 'if' cases. But once you've prototyped your idea and you know its limitations, reimplementing that code with an appropriate design pattern can save you a lot of grief in the future. Design patterns exist

because people have run into these same problems over and over again – why not take their advice?