Nicholas English

CIS-622-01

December 5, 2023

Term Project Writeup

The term project that I selected to do was to refactor and extend the 2D zombie game I made for CIS-611, which was completed in the fall of 2022. Each of the three phases will share the individual pull request for their respective work. Additionally, to help grasp what that went into this half semester project, here is the final Pull Request that showcases everything completed.

Refactoring

When I approached this project, I found that there were three main areas that needed refactoring, all of which involved how similar classes were and if they were being extended properly. These three areas revolved around the Store objects, Weapon objects, and Player objects.

For better visualization of what went into refactoring, view this Pull Request.

Store Objects

The problem here could be encapsulated with the "Shotgun Surgery" code smell. Simply put, each Store object had its own class with its own unique, yet nearly identical, logic. I previously tried to remedy that by putting some identical logic into a helper class, which the store classes would expose to outside elements. In this case, that could also be described as a "Middle-Man" code smell.

To resolve all the above, I implemented an abstract base class called BaseStore, which the individual Store objects overrode. Additionally, this base class also houses the logic that was in the helper class, which removes the awkward middle-man scenario from before. Outside of code smells resolved, I also structured the BuyAmmo and PurchaseWeapon methods to have a single return statement as I felt this helped to clarify what was being returned by these methods. Additionally, removing the helper class caused tests to fail to compile and I could no longer easily mock the AudioSource class. This was resolved by building an extension on top of Unity's AudioSource class and replacing all calls to Play with a new TryPlay method call.

Overall, this solution solves the two code smells, but it also potentially introduces the "Large Class" code smell since the BaseStore is now doing a good amount of work. However, if you look at LaserStore, PistolStore, or RifleStore, the benefit of how small and easily maintainable those classes have become easily outweighs the potential for that new smell.

Weapon Objects

Like the Store objects, the weapon objects faced a similar issue with having “Shotgun Surgery.” When I originally wrote this code, there was a better attempt at centralizing the logic, but perhaps I got a bit too overzealous with implementing abstracts in the base class. Almost everything was marked abstract and at least half of it really should not have been. Furthermore, the base class wasn’t named properly as it was just called “Weapon,” which doesn’t tell us much.

The first item resolved was to rename Weapon to BaseWeapon. I then removed the abstract modifier from all the methods in the base and instead pulled up the logic (with some tweaking) from the subclasses. With these changes, I was also able to improve my unit tests because I could remove a TestWeapon class I made just for testing the original weapon logic. This removal meant I could test more directly on the real codebase, rather than with a studded class. Additionally, in the spirit of clarifying names, I renamed AmmoClip to RemaininClipAmmo and RemainingAmmo to RemainingTotalAmmo. My reasoning for this change stemmed from the fact that if I, the guy who wrote this, was getting confused which variable did what, then I’m sure it is just as confusing for someone else who may look at my code later.

Just like the Store objects, this suffers a similar fate that the “Large Class” code smell is popping back up. But again, the benefits are clear when you look at what happened to the LaserWeapon, PistolWeapon, and RifleWeapon classes.

Player Objects

A couple of the Player objects have the “Large Class” smell, and the others have the exact same issue of “Shotgun Surgery,” but coming to the solution for this round of “Shotgun Surgery” took considerably longer and with much more trial and error.

To begin with, I’ll start with the Player objects that saw the least changes. The PlayerLogic and PlayerStatus classes were already a result of splitting a previous goliath Player class, and while these two classes have that “Large Class” code smell, the cohesion was high and I couldn’t find any logical points to break into further, smaller classes.

Very similar to the above Store and Weapon objects, there was a lot of repeat logic between the PlayerLaser, PlayerPistol, and PlayerRifle classes. At this point, I’ll note the reason for this repetitive issue likely was a result of the fact I created and implemented one weapon at a time, and this was likely done with little to no forethought on how to make it easier to continue adding new weapons. And as mentioned earlier, the solution here wasn’t as easy to come to and this was a result of these classes being tightly coupled with some Unity logic. Referencing the GameObject from the MonoBehavior class, which was done on nearly all lines in these classes, cannot happen willy-nilly in subclasses. Unity has rules of where this object is allowed to be referenced, which unfortunately doesn’t allow me to use in read-only fields like how I created my previous solutions. I eventually came to the realization that all GameObject related logic was going to have to live in the newly created BasePlayer object, and that the subclasses where simply going to just override the name for the bullet type and the name for the Player object. Additionally, I created a method to lazily create the expected weapon object. Doing this lazily ensures only a single instance of the objects is created and is only created if it's ever needed.

Lazy Singleton Logger

One of the design patterns covered in the course was the Singleton pattern. Because I was continuing with an already existing project and hence didn't want to rewrite any of this "legacy" code with a pattern, I felt it was appropriate to use the Singleton design pattern since it could easily be used with any new features. Specifically, I wanted to add a logger, which I believed to be a perfect candidate for the design. And, to add a hint of complexity to this design pattern, the Lazy Singleton design pattern was chosen. It is worth noting, however, that there is zero benefit to it being lazy in this project since a value will be assigned every time the game is ran. Realistically, this likely only adds possible confusion for future developers who work on this project as they may question what the original design intent was.

This feature was quite simple to implement. The Logger class was created with a static Logger instance (initialized to null), a private constructor, and a static GetLogger method that initializes the Logger instance when it's first called. There exist two additional methods that are standard log methods and have no actual bearing on the design pattern itself. From there, the implementation of this class is spread throughout the project wherever I felt I needed to add logging.

For help seeing what went into creating and using this singleton, view this Pull Request.

New Feature (Support AI)

The end goal of this phase of the project was to introduce a new support entity that would act as an ally to the player. Similar to how the player may purchase weapons and ammo, the player can also purchase the support of up to five of these entities, which follow the player around and shoot at nearby zombies. Below will discuss more of the design with regards to the shared store logic, NPC logic, and human logic.

To help illustrate what went into this, here is the Pull Request. A lot did go into implementing this new feature, and so a lot of files were both moved and manipulated for one reason or another. As a helpful suggestion to get through this PR, and in order of appearance in the PR, I would suggest focusing on: Human folder, NPC folder, PlayerLogic class, PlayerStatus class, BaseStore class, SupportStores folder, BaseWeaponStore class, and BaseWeapon class.

Shared Store Logic

Previously, the only store logic that existed was for an abstract BaseStore, which then had weapon specific stores that overrode it. To account for the new store type, sharing some of that logic would make sense. To do this, the BaseStore was made more simplistic and made to focus on core functionality of what makes a store a store. From there, we now have two additional abstract base stores, one for weapons and one for support. Both extended bases have more specialized logic for what differentiates these two types of stores. For example, the BaseWeaponStore has logic for delivering weapons and ammo to the player, while the BaseSupportStore has logic for spawning specific support entities into the world. Focusing on the support side of the stores (since that's for this new feature), implementation of new support store types is simple, straightforward, and it takes about ten lines of code. Despite it being simple, I did opt to create only one of these stores so that I could deliver the minimum viable product that showed both the design and the new feature worked as expected.

Shared NPC Logic

EnemyLogic and FriendlyLogic are the two classes that correspond respectively with zombies and support. Both entities could be described as non-playable characters (NPCs) since they both interact with the player in some way. Given that, it would make sense that they would share substantial logic. However, while I attempted to move some of the EnemyLogic code into a new abstract BaseNpcLogic class, I realized that both classes were going to have some tightly coupled and specialized logic that just wouldn't benefit the other. This could be attributed to the zombies being programmed to get harder as the waves progress, and it could also be attributed to zombies having to run at their enemies, while the support entities require line of sight, aiming, and shooting at their enemies. Given this, the only shared logic that could move to the base revolved around initializing new entities, targeting nearby opponents, and checking if the entity itself has died.

Shared Human Logic

Similar to the above, we would expect that the player and support entity would share some logic since they are human types in this post-apocalyptic world. To account for this, it was decided that using an interface made the most sense because the FriendlyLogic class was already implementing one abstract class and wouldn't be able to handle another. What came out of this was just a single method that both the player and support entity shared, which was how they handled being hit by the zombies. While I was surprised at the sparce amount of shared logic, I viewed this as a good thing since the method, Hit(), is only called from the EnemyLogic class, which doesn't need to understand how the player or support handle being hit, they only need to notify them that it occurred. If we then viewed zombies as one domain and the player/support as another domain, this would be a nice example of dependency inversion since the EnemyLogic class no longer calls directly to the method, but rather through the interface.

Reflections

This section will focus on my personal thoughts on whether the refactoring and singleton logger were overall beneficial and helpful for this project. The only discussion related to the new feature will revolve around if refactoring and the singleton logger were helpful for its implementation, which will be covered in their respective sections and so no single section below has been given for the new feature.

Refactoring

Part of the refactoring performed was centralizing shared logic and creating abstract bases. Because of this, adding calls to the logger was a fraction of what it could have been since I only, for example, had to only place code in the BaseStore class rather than the individual store classes. One way to look at the net benefit here is that by removing copied and pasted code, we further cut down on future copied and pasted code.

Refactoring also benefited the new AI feature in multiple ways. The first being that giving the AI a weapon was unbelievably easy. By calling the new RifleWeapon constructor with a few parameters, 90% of the work was easily done and all that remained was a few lines of code to handle ammo, and then clicking and dragging a few items in the Unity engine's UI. Outside of the weapons, the refactored store design also gave me direction on the best way to implement the support stores, which is that at the lowest level it should look more like a config file than a "real" class. Having this design in place now means that implementing additional support stores in the future should be as simple as copying and pasting the current RifleSupport store into a new class and just changing a few values in code.

Lazy Singleton Logger

The creation of a singleton logger, in general, makes sense so that multiple loggers don't have to be configured throughout a project. While one would think this project would be no exception, it is one of the exceptions that didn't require it. Using the Unity engine meant I already had access to a static logger, and my implementation could easily be described as a middleman approach as the singleton logger uses Unity's static logger.

When it came to implementing the new support AI, the logger was only helpful a few times. Specifically, the logs helped verify what was occurring when purchasing support through the shop and help catch a couple issues when playing audio. As the project progresses and if I remember to continue adding log messages, then having this singleton logger implemented now will have been more beneficial than attempting to implement it in the future. For now, it didn't add much since I could easily of attached the debugger to Unity and saw in real time what was going on.

References

Refactoring: Improving the Design of Existing Code (2nd Edition)