Object Orientated Software Engineering

Dungeons and Design Patterns - Report

Anurag Singh
18944183

May 31, 2020

Contents

1 Design

1.1 Model

The model classes of Player and Enemy, both share some details, to reduce code duplication and to allow more expandability, a Unit class was created to share the common code and to setup how the two Units can behave, including their attacks and defends. This instance of inheritance wasn't really used to be polymorphic, but rather to allow code reusability and expandability for future other units such as NPCs (Non Playable Characters).

Extending from the Enemy superclass were the specific Enemy types, Slime, Goblin, Ogre, and Dragon, these subclasses did not do much rather than provide basic default values for the enemy super class to use.

```
...
    public const string NAME = "Dragon";
    public const int MAX_HEALTH = 100;
    public const int MIN_DAMAGE = 15;
    public const int MAX_DAMAGE = 30;
    public const int MIN_DEFENCE = 15;
    public const int MAX_DEFENCE = 20;
    public const int GOLD_REWARD = 100;

    public Dragon(Ability ability)
        : base(NAME, MAX_HEALTH,
                new MinMaxRandom(MIN_DAMAGE, MAX_DAMAGE),
                new MinMaxRandom(MIN_DEFENCE, MAX_DEFENCE),
                GOLD_REWARD, ability)
    {}
...
```

Along with a overridden `ToString()` which contains the ASCII art for that enemy type. The enemy subtypes effectively provided contained constant values that the enemy could be made from, as throughout the program enemies are treated as 'Enemy' rather than their specific subtype, because in general throughout, the actual subtype of the enemy doesn't matter.

This reduced code duplication, and adds more reusability and expandability to the code as adding another enemy is as simple as extending the enemy class and calling the base constructor with all of your information, the dependency injection of Ability also allows enemies to potentially use abilities of other enemies, since the abilities are treated as the superclass type Ability, so any enemy can have any ability, it simply needs to be constructed accordingly, this allows for expandability.

The Abilities used a Strategy pattern, with each subclass containing a different 'algorithm' for executing their own ability, through this it's possible for enemies to use abilities without being aware of what the ability does or of what subtype of Ability it is.

Items have a similar strategy structure but the polymorphism there is not really used at runtime, due to the inventory requirements, it is not really possible to treat every item as an Item, but rather its needed to know what subclass it is in order to avoid items such as potions from being assigned into the current weapon or armour slot, the Item superclass therefore is used to reduce the code duplication, and to make it easier to extend the program by adding more item types.

The weapon and enchantment system use a decorator pattern, in order to allow any number of enchantments to be added to the weapon, this pattern allows decoupling and extensibility by making it easier for more enchantments to be added but also reduces coupling by making the weapon unaware of its enchantment(s).

This pattern requires the Enchantment to be a Weapon, which is a Item, so a Weapon abstract class exists which extends Item to grab the common attributes, and contains abstract methods that can be overridden by both the Enchantment abstract class's subtypes or by the WeaponImpl class which contains the implementation of an Weapon.

The decorator pattern allows you to create a weapon, and then wrap it using an enchantment subtype to create an enchanted weapon which can continue to be treated as a regular weapon since an Enchantment is also a Weapon, meanwhile weapons of type WeaponImpl can also be treated as Weapons since WeaponImpl extends the Weapon abstract class.

This means when a Weapon is equipped by the player, it is unaware of an enchantments on that player, which also applies to the weapon itself as it is also unaware of an enchantments, this allows the enchantment system to be decoupled from the Player and the Items. Since the player doesn't need to know the actual subtype or the layers of wrapping required to create a specific enchanted weapon.

The Enchantment decorator pattern is achieved by the Enchantment class containing a field named next which contains a reference to another Weapon, which could be either another Enchantment or the WeaponImpl, which leads to a sort of linked list of classes.

```
DamageEnchantment -> FireDamageEnchantment -> WeaponImpl
```

The DamageEnchantment will override the `Use()` method on a Weapon, and will simply do it's modification to the damage done by the weapon and add to that by calling the use on the `next` field, this means the program will iterate through this List of enchantments until in the end where it reaches a weapon which will return it's damage, and as the stack unwinds the damage will be modified accordingly and returned from the original enchanted weapon.

1.2 Controller

Due to the requirement that it be possible for multiple different ways to load Shops be possible, an interface known as ShopLoader is provided.

```
public interface ShopLoader
{
    Shop load(string location);
}
```

This interface is very simple and exists to allow other `ShopLoader`s to be written, because if a ShopLoader follows this interface, it can be easily swapped out for another ShopLoader at any given moment, meaning the standard FileIO ShopLoader could be replaced with a Network ShopLoader by simply writing the loader and replacing the loader constructed, since the shop loading relies on the behaviour enforced by the interface.

This is a very simple use of the strategy pattern to allow for other potential loaders to be written.

The use of the controllers and the division of their responsibilities is managed through a State pattern, which effectively makes each menu option it's own state, which controls it's own aspect, this basically makes each state a controller of their domain of work, for example the BattleState hands the entire player vs enemy battle and then returns control back to the MenuState. The GameEngine class contains a stack of states to allow this behaviour easily, as a new state can be added by pushing unto the stack, and any state can return to the previous calling state by simply popping itself off the stack.

This division of responsibilities through the different states effectively referring to different menu options makes it possible for the main menu to simply be able to push the selected state onto the stack, this allows separation of concerns as none of the states are concerned or rely on another state, and there is no need to know which state is currently active within the program.

Each state has access to the Context and the UserInterface, the context is simply a reference to the underlying GameEngine class that manages these states, this GameEngine class holds references to the Player model class, and the Shop model, this allows any of the states access to the player object, and the shop object, and the user interface allowing them to request using input.

This allows the system to be easily expanded by simply creating another state extending the State type, and switching to that state when needed. The GameEngine class is aware of states but unaware of which state is currently running, the states also do not know of each other and cannot impact each other directly.

An EnemyFactory class is used in order to construct enemies based on the current probabilities, this uses the inheritance structure mentioned earlier in order to construct a specific enemy type with an ability but return it as an Enemy superclass which allows us to fight an enemy without knowing what type of enemy it is, once the factory has spawned the dragon it sets a flag which can be checked to know whether we have reached the endgame situation where the game ends regardless, just depends on whether you win or lose.

1.3 View

There was not much in the View namespace that contained any specific patterns, it was a case of simple inheritance to provide each State with a specific view with access to more generic non state specific methods provided in the View class. This allowed consistency through the program as each view had some basic methods it could use, and also access to specific methods based on what the view needed. In a more GUI friendly application, I could use the composite pattern here to build a UI system, but in the case of this console application that would be considered unneeded, and add complexity to a portion that doesn't need it.

2 Alternative Designs

2.1 Observer Pattern

At the moment, the model is updated by the controller as standard, and then the controller needs to call the view in order to update the output to the user. Another method to do this would be to make a View an Observer, and the models Observables, this would mean when the model is updated by the controller it fires off an event to the views that are on the observer list for the model, this would mean the controller would no long need to be aware of the view itself, but also wouldn't increase coupling between the view and model since they still wouldn't be aware of each other but rather they would know about the observer system.

This would reduce code in the controller as model updates will automatically correspond to view updates, meaning the controller doesn't need to explicitly update the view after making a model change. This would also lead to a thinner controller as it would not need to control the views or even know of them, the model will simply update them by notifying all observers, which could refer to different UI elements.

Overall this would add a level of complexity unneeded for a console application but would serve a much better purpose with a GUI application as view updates could occur asynchronously.

2.2 Decorator Pattern on Abilities

Looking at the dragon's ability, it can easily be constructed from 2 different abilities combined into one, for this case a decorator pattern could be used for the ability system rather than assigning a single ability to a single enemy. A Decorator would allow better expandability into the future as any number of abilities could be combined together along with newer enemies in order to expand the game, although this is still possible but it requires more classes to be created to expand the ability choices, with the decorator pattern a unique set of abilities could be achieved at runtime in order to further expand the ability system.

Although this would have good future expandability, it provided nothing considering the specification provided, as the specific listed abilities as enemy specific rather than something that would be used to mix and match abilities onto any enemy.

This would be a significant change to the current ability system, and allow for many permutations of abilities that can be made from the base ability list.