Strategy Patterns Used and their achievements

Most of the program is centred on the Strategy Pattern with a small part of it applying the decorator pattern. To start of, the enemy classes all have the same methods that just have to be implemented differently and so I decided that it would be best to apply the strategy pattern to all the enemies. There is a central interface where all the methods are defined. They are then overridden and used accordingly by each individual class. I believe this design strategy for enemies is beneficial as it will allow any future extensibility of enemies to be easier. There is no significant editing of any code when adding new enemies but just a simple addition of a new sub class which will implement the already defined methods in the Enemies interface in its own way. In addition to this, the potential level of coupling that the enemy classes can have with other classes that want to use it, like the class containing the main method, is reduced. This is because with the level of encapsulation in each class and the method definition in the interface, I am able to declare each individual class through the Enemies interface, reducing coupling.

The strategy pattern is also applied when designing the inventory classes, although there is a decorator pattern aspect, I have applied with one of the inventory items. For inventory classes, their super class this time is an abstract class instead of an interface. This is to be able to better accommodate the Weapon class which I also made an abstract class. This allows the design to be more flexible in using classes that may actually need to be overridden in the subclasses of the Weapon class. Use of the strategy class again for the inventory allows the easier future extensibility of the program if there is ever a need to add different types of inventory. To add new types of inventory, a new class can just be created again to act as sub class of the Inventory abstract class and inherit and apply its methods in a different way. So the different types of inventory basically will bring the same types of advantages in coupling, extensibility and information hiding as the enemy classes, and they all address the same goal.

For the individual subclass Weapon in the design of inventory classes, I have made weapon a super class itself to other subclasses. This is because during the game, enchantments can be added to a weapon and so with this in mind, I applied the decorator design pattern to the weapon and enchantments classes. Making the Weapon class an abstract class also allows adding new abstract methods to be added which contain key characteristics of each enchantment. The Weapon class has a sub class which will contain characteristics of a normal weapon and another sub class which acts as the super class to the different enchantments. This subclass is an abstract class and it is this class that it key to the design of the Weapons because it will allow the characteristics of the weapon to be changed at run time by adding different enchantments. This design also allows easy addition of new enchantments in the future. All that must be done is the creation of a new sub class for the AddEnchantments super class.

There is a separate class for opening the file containing the shop contents. This class is only responsible for reading from the file and it is instantiated in the main method for displaying contents of a shop. The design supports a whole class just for that so it is easier to modify how the items in

the shop are read if there is ever need to. There is increased cohesion in the class so modification of it would not be difficult.

The design patterns used allow for code reusability and cohesion because they allow each class to do one well defined task. For example, with the classes used in the strategy pattern, they all perform similar goals but putting all of this in one class would have meant chaos in the cohesion of each class. I believe my chosen design achieves optimum levels of cohesion and coupling while also allowing for future code reusability and extensibility.

The PlayerCharacter class stands as its own class and not as part of a different pattern specific to the player. This is because there are not many things that can be added to the player's characteristics at runtime and so I did not think it would be optimum to have the class as part of a specific pattern. The class is very specific in what it does by carrying the player's characteristics and items, as well as being able to modify the player's attributes accordingly. The class has an aggregation relationship with Inventory, which actually has reduced coupling due to the design pattern that has been implemented for the Inventory classes. The multiplicity within the class would have been a problem if the different inventory classes were standalone, which would increase coupling exponentially.

I also decided to use a list container type when carrying the inventory that the player possesses. this makes retrieval for the values in the list easier and more mutable. The mutability of the list was important, as the player can constantly change the items at their disposal and retrieve them easily, and having maps or sets as the container would have made it somewhat more difficult to offer that mutability.

Design Alternatives

An alternative design could have been the greater use of the Decorator Pattern in the design. Instead of just using the pattern for adding enchantments to weapons. I primarily had the idea of having a Decorator pattern centred around the game character which would have the interface declaring all the methods to be used by the character, then a subclass which would act as a way to modify a game character at runtime. This design strategy might offer a more flexible way of being able to constantly change player attributes like the amount of inventory the player has or even which enemy the player fights at a particular time. The inventory classes could then act as subclasses of the character modifier abstract class so all the inventory that the character has at any point would be modified through that, instead of having to create inventory instances in the character class and the main method class. So each inventory class would act as a base implementation. The enemy classes would still have the strategy pattern and be aggregated to the main method class for use. Another way to also implement this would be to have the enemy classes as base implementations for the character modifier to allow the definition of each enemy at runtime. This would provide another workaround for how to structure the weapons. Ultimately, I feel having the character being centred on the decorator pattern could be achieved in a number of ways but I felt it was a more concrete method to design the program the way I designed it in the end.

Another good way of designing the program would have been through the use of the Factory Design Pattern. The Strategy Pattern could be replaced in both the Enemies design and the Inventory design by the Factory pattern. The type of enemy that the character will face for each battle is not known before runtime and so this design pattern would allow for the deciding of the enemy at runtime with fewer code redundancies and a more centralized approach to selecting an enemy. The pattern would also help in some aspects like reducing the instantiation of classes used to implement the enemy and inventory attributes, which would reduce the coupling by reducing dependencies and again help in extensibility of the application. The Enemies super class would be an abstract class having the method definitions of the enemy with the individual enemy subclasses inheriting from it just like in the Strategy Pattern. Where it would differ though would be that it would have a sort of controller class which creates enemy objects based on the requirements that the class with the main method is asking for. The controller class would create the required enemy and be used by the class containing the main method. This way of doing it really eliminates a lot of code redundancy. The design would be used in a similar way for the inventory classes, with the Inventory abstract class having the weapon, armour and potion subclasses, just like with the Strategy Pattern. The controller class which would create the required inventory would be connected to the character class instead. This is so that the character would be able to store a list of the inventory they own. From there, calling inventory owned by the character from main would be done through the character class.