OBJECT ORIENTED SOFTWARE ENGINEERING ASSIGNMENT REPORT (SEM. 1, 2020)

The work of Kristian Rados (19764285)

EXPLANATION OF DESIGN

The program starts with the GameEngine class which contains the main() method. From here a Player, ShopController and ShopUpdater instance are created, with the ShopUpdater being used to initialise the shop's inventory and the Player object hence being equipped with the cheapest Weapon and Armour available in the newly created shop inventory.

The ShopUpdater is an implementation of the strategy pattern and is used to update the Shop's inventory. The use of a strategy interface means that the ShopController can be decoupled from the implementation method that is being used to create the update. Because of this, the program can be easily modified to use either a database or an input file by simply writing a new strategy for the ShopUpdater. The method being used is decoupled from the ShopController using it, which means the ShopController does not have to change any of its logic to allow for these changes.

The Player, which is the most important object in the program, forms a part of the Entity inheritance hierarchy. Both it and the Enemy class extend from the Entity class as it represents the traits which are shared among the Player and any being it may encounter: health and the ability to engage in combat. What is more important design wise is the polymorphism that is brought by Enemy itself being an abstraction. There are different varieties of enemy that have varying special abilities that could be used in combat. By having them inherit from a common Enemy class it means that the player may attack the enemies via method calls without ever knowing what kind of enemy it is. Meanwhile, the Enemy can make use of a species-specific ability regardless, as we have specified that each type of Enemy will have its own ability. By also having a factory, an Enemy can be created by a controller based on an input, the number of battles, without the controller having an expectation of any specific Enemy type. Therefore, they can be both created and used by other classes with minimal coupling while retaining full functionality.

The ShopItem interface and its inheritance hierarchy is essential to the functioning of the shop. Breaking weapons, armours and potions into three different classes inheriting from a single ShopItem class allows them to be treated equally by the ShopController. It doesn't have to know about the details of the items it's storing other than their details and their values. Hence, these methods are methods are enforced as abstract methods in the interface.

The weapons are more complicated however as they need to be enchanted. To facilitate these enchantments, I have made use of the decorator pattern. This way the original base weapon is not modified or overwritten, but rather it is wrapper by a decorator class; an Enchantment. However, by having both the base weapon and the Enchantment inherit from the same Weapon interface they can be treated as the same by the shop. Hence, an Enchantment is a Weapon. However, the Enchantment does not actually implement all the details required of a weapon. It instead always points to base weapon and simply modifies the values that come in and out of the base weapon. The Enchantment can also point to other enchantments however, so this effect can be chained. The Enchantment implements the generic details required of the enchantments and its children classes simply implement the specific details of what is being modified, which allows a new enchantment to be created easily through the creation of a new subclass of Enchantment.

TWO PLAUSIBLE DESIGN ALTERNATIVES

ONE: THE USE OF THE OBSERVER PATTERN

A design alternative which I considered using was to application of observers to the View and the Model. Using observers could possibly simplify the program's function on a conceptual level as direct but nonetheless decoupled communication between user inputs and their respective Controllers, as well as model outputs and their respective view outputs could remove much of the clutter in the controller. In my currently implemented system of hierarchical method calls the Controller inevitably ties together everything by acting as the connection between the Model and the View. The disadvantage of this in my eyes is that it can make the source code of an otherwise simple program difficult to understand conceptually, as you have to follow the many paths of the Controller's conditional method calls.

The advantage of using a more event driven programming style with the observer pattern is that it could improve the testability and extensibility of the program by allowing the Model and View to further decouple. It would also make switching to a graphical user interface far simpler. However, it may technically overcomplicate the initial terminal/text-based use case of the program which is more naturally suited to the non-event driven style typically.

The Model and the View in particular would be redesigned to allow them to act as the observer sources. The View would handle its role of user input similarly to now, but the Model could be redesigned to play a more active role in the logic of the program by signalling its own observers. This would be a change from its current role as mostly just the pure storage of data.

TWO: IMPLEMENT CONTROLLERS AS OBJECTS

In my current design all of the Controllers, with the exception of the ShopController, are implemented as classes filled with static methods. They simply control the flow of the program within certain aspects of the program without actually representing those aspects of the program.

Each Controller, with the exception of the ShopUpdater strategy and the generic Controller, could be implemented as an actual representation of the 'screen' which it is managing. They would still play the primary role of controlling the logic of the program but being able to manage them as objects would allow a more customisable and expandable program, as different instances of a 'screen' could be created with their components saved. For example, the BatteController could represent a battle in itself rather than just control a battle, allowing battles to be saved or paused without having to necessarily worry about the program's flow breaking. It might store each turn by the player and by the enemy in a field within lists, allowing them to be retrieved later for example. When starting a new battle, the first battle will be left alone and instead a new instance will be created with a factory.