Object-Oriented Software Engineering [COMP2003]

OOSE Assignment - Semester 1, 2020

By Eric Wojcik

Student Number: 19142124

TABLE OF CONTENT

Dungeoniest Dungeon *is an epic text-based, turn-based, non-RTS, procedurally generated, open world, quasi-looter-shooter, high-fantasy, battle royale adventure game. It follows our hero, Burger 'Poor Man' Pimblan III - who awakens in a dark, dank dungeon its filled with monsters, gold…and suspense! Follow his (or her) journey through the Dungeoniest Dungeon. They will probably survive! Who knows! I don't!*

Section 1.0 - Overview

This report provides a brief discussion on the design decisions that modeled the development of ***Dungeoniest Dungeon***. Specifically, this will cover the core design aspects of the game, detailing any design patterns used and how they affect the coupling and future extensibility of the program. Other important facets of design such as testability and separation of concerns will also be explored, as well as the alternative designs considered during development.

Section 2.0 - Key Aspects of Design

2.1 Patterns Implementation

2.1.1 - Strategy Pattern

This implementation utilises the strategy pattern to dynamically choose between two different methods of loading shop data into the game. These methods are local file-reading, and remote file-reading, which are represented by the LocalFIleReader and RemoteFileReader classes respectively, both of which implement the FileReader class. The LocalFileReader is capable of reading the contents of a comma-separated text-file formatted as per the assignment specifications.

At runtime, this is made the default file reader so the game shop can be automatically populated with items before access, however, once the menu UI is initiated the user has the option to load more shop data using either method. This can be done at any point during the game, and will immediately update the shop if successful.

The requirements also specify the need for new types of data-loading methods to be easily integrated into the rest of the program. For this reason, the strategy pattern was selected as it allows us to encapsulate a family of distinct algorithms that are designed to achieve the same goal, in an easily extensible manner. Specifically, our LocalFileReader will use traditional file I/O to read in local data, whereas our RemoteFileReader would require an entirely different algorithm to feed information in from a remote database, however, both methods are engineered to load data into the game. Furthermore, assuming even more methods were to be included at some future point in time, the low coupling afforded by using the strategy pattern would easily permit this, as well as any

modifications to the existing algorithms, with minimal alterations to other areas of the program, making it the optimal solution for this particular functionality.

2.1.2 - Factory Pattern

The Factory Pattern is prominently featured throughout the program to control which sub-classes are instantiated at run-time. The game has a factory class for almost every base-class or interface that exists, and have been labeled appropriately. The ItemFactory is responsible for deciding whether to return a Weapon, Armour or Potion object back to the calling class, which is typically a reader as it uses this to populate the game's shop. The EnemyFactory functions identically, and returns either a Slime, Goblin, Ogre or Dragon on a per round basis. The ReaderFactory chooses between LocalReader, and RemoteReader, and the TurnFactory selects a PlayerTurn, or EnemyTurn depending on the circumstance of the call. The EnchantmentFactory is used when purchasing an enchantment in the game's store and returns one based on the player's selection.

This pattern method is primarily used to relocate all the coupling that results from creating a new object into a single location, or in this instance several locations as opposed to be scattered throughout the entire program. Furthermore, the Factory Pattern enforces information hiding to a high degree. For instance, in the BattleController when an enemy is generated for the round, an Enemy object 'e' is declared but never instantiated until an enemy type is randomly selected and sent to the EnemyFactory. The factory will then return the corresponding enemy sub-type as an Enemy object, so the calling method in BattleController will only receive the Enemy object. As such, it will never actually know what specific type of Enemy is being assigned to 'e', as those details are being hidden from it.

2.1.3 - Decorator Pattern

The Decorator Pattern is applied to the Item class, in the form of the Enchantment class and its various sub-classes. While any item can receive an enchantment, in the current version of the game only weapons can have enchantment applied to them. Doing so will provide the specified weapon with a statistic boost, typically in the form of power amplification.

A key requirement specified that these effects should stack, and a weapon should be able to have any combination of enchantments applied to them at any point in time. The Decorator Pattern is excellent for this purpose as it allows an item to be sequentially encapsulated with additional functionality, over and over, resulting in a chain of additional functionality layering over the original object. This is all accomplished without having to modify any of the pre-existing code.

In the game, each weapon enchantment affects an existing property of the weapon. Specifically, an enchantment will manipulate the power of the weapon, the sell value of that weapon, and the name of the weapon. The enchantment classes listed below all affect these properties in a different way:

- EnchantmentBasicDmg (+ 2 dmg)
- EnchantmentExtraDmg (+ 5 dmg)
- EnchantmentFireDmg (+5-10 dmg)
- EnchantmentPowerUp (*1.1 dmg)

So if I had a ‘Short Sword’ that dealt 6 dmg, and cost 42 gold, and I applied the basic dmg, fire damage, and power enchantments to it in that order exactly, we would end up with an obstructer structure like so:

```
( ( ( (Weapon) EnchantmentBasicDmg) EnchantmentFireDmg) EnchantmentPowerUp)
```

And the weapon details would print as follows:

```
'Short Sword + BASIC-DMG + FIRE_DMG + POWERUP || DMG: 18 || COST: 80'
```

Additionally, with the current set-up it is easy to create new types of enchantments for a weapon, as you only have to create a new concrete sub-class that extends the decorator abstract class. The same process can be applied to armour and potion enchantments if they are required to be added to the game in the future.

2.1.4 - Template Method

The game implements the Template Method Pattern to code functionality for player and enemy turns during a round. As such, this pattern is present in the abstract TurnTemplate class, with PlayerTurn and EnemyTurn being extensions of that.

As per the assignment specification, a player's turn consists of two potential actions. As the first option, the player can choose to attack an opponent directly, resulting in the enemy receiving some or no damage depending on the game’s damage modifier calculation. The player can also choose to use a potion, which can be either a healing potion or a weaponised potion used to damage enemies. A selection of either result in

their turn ending. On the other hand, an enemy turn consists of a single action - attack the player. However, they have a chance for an ability to execute which may also heal the enemy depending on the enemy type. This will replace the action for their turn if executed.

What was observed early in design was that these two systems shared many common elements. An entity will always be attacking the opponent or healing themselves. There will almost always be a calculation for the defence, a calculation of an effect value (whether that be damage dealt, or health recovered), and the application of that effect on the opponent or the entity themselves. If these were to be coded as separate algorithms, they would be mostly identical save for a few specific steps.

For this reason, the Template Method Pattern was used as it excels at abstracting common elements from several identical algorithms into a common 'template' algorithm, where the different aspects are turned into generic method calls. Each sub-class of the template method class will have its implementation of these generic methods. Not only does structuring the turn system this way streamline the code, as you only have to instantiate a single object and call a single method for both turns, but the complexity surrounding future modifications is reduced. Any fundamental changes to the turn algorithm would only require a change to the template method itself and any corresponding changes to the subclasses as a result of that.

We used the following hook methods across both PlayerTurn and EnemyTurn classes:

- **doAction():** Obtains the value of the action effect, which is from the player using an item (such as a weapon or a potion), or the enemy performing an attack or an ability
- **getDefence():** Obtains the defence property of either the player or the enemy,
- **setHealth():** Applies the final effect onto either the opponent or the

- **removeFromInventory():** Removes an item from inventory if used as part of the turn

2.2 - Testability

This application offers testability predominately through dependency injection, wherein an object (or in this case, static class) is used to provide the dependencies of other objects. Namely, new instances of objects should always be created within main or as close to main as possible, and then passed into whichever object as required. Because of this, this program declares and initialises nearly all of its objects within main, and any class that requires another object has them passed in as a parameter. The only other instances wherein new objects are created are within the factory classes, which are employed to separate the responsibility of creating classes from any other erroneous areas of code. As a result, any hard-coded dependencies are removed, leaving the program as loosely coupled as possible.

The use of non-static factories further promote testability by utilising a setTestObject() method, and a field to store the said test object. The factories in this program have specifically been constructed such that, if a test object is set, the normal series of operations are ignored and the testObject is returned, allowing the factory to be successfully mocked in test scenarios. This is important as unit-test code is designed to analyse isolated situations where the input can be controlled, and thus allowing the factories to be mocked facilitates this.

2.3 - Separation Of Concerns

2.3.1 - MVC

This implementation achieves separation of concerns via the Model-View-Controller format. As the name suggests, all of the game classes are broken down into models, views, and controllers. This way, each class has a distinctly defined responsibility. All the models - Player, Enemy, Item, etc - serve as storage for data. The controllers serve as a thin layer of communication between the interface and the model, performing any kind of decision making or logic operations based on the user's actions. Finally the interfaces serve as the view, and only exist to take user input, or display results.

This way, not only is everything more readable, but it simplifies maintenance as every class is segmented into clearly defined roles.

2.3.2 Exception Handling

Apart from using the Java API's inbuilt exceptions, several custom exception classes were created for the purposes of this program. The InvalidFileFormatException is used in the FIleReader classes to catch out any instances where the file being read does not strictly abide by the ideal file format specified by the program. That is, each line must contain comma separated values formatted as such:

- Value [1] = 'W', 'A' or 'P'
- Value [2] = Non-Empty String

- Value [3] = Number
- Value [4] = Number
- Value [5] = Number
- Value [6-N] = String

If a line does not meet these criteria, the file is rejected, and the InvalidFileException is thrown.

The second custom exception is CanNotLoadShopDataException, and is a somewhat more generic exception simply used to describe that something went wrong during the file loading process. It is thrown whenever an InvalidFileFormatException is caught, as we never want the calling methods to know the specificities of the file format error due to separation of concerns.

Section 3.0 - Alternative Designs

3.1 - Observer Pattern

The use of an observer pattern was considered during the early phases of development. Initially, the idea would be to have the Player class as a Subject, and enlist all the interfaces (BattleInterface, MenuInterface, ShopInterface, etc) as Observers, so that any time the player does anything significant that involves an internal state change - i.e. buy something at the store, get damaged in battle - the interfaces would be notified and display updated player statistics onto the screen as a result. This would be especially viable in the future if the game was to be implemented with a GUI, as it would effectively

allow this information to be instantly updated, and displayed across multiple views simultaneously.

Advantages:

- If need be, you can add any additional views as observers to the Player's list of observers at any point in time
- This loosens coupling between object interacting with each other, resulting in the said objects knowing the least amount of information about each other as possible. Particularly, the Player in our game scenario would know about the list of views that observer it, and that they implement Observer, but it does not know their concrete implementations

Disadvantages:

- Restrictions involved with using Java's Observer Interface which enforces inheritance over the use of interfaces, which means observers would not be able to extend anything else
- If used flagrantly, the observer pattern can increase the complexity of your design

3.2 - Decorator Pattern

An additional Decorator Pattern was considered to be implemented with the Enemy as part of their abilities. Currently, each enemy has a single baked-in ability, except for the Dragon which can randomly choose between two abilities. This is fairly limited as those enemy types will only ever exhibit those same abilities during battle. This may result in battles becoming very stale, very quickly, especially when expanding the scope of the program. For this reason, an additional Decorator Pattern could be used to diversify enemy behaviour via sub-abilities, that can stack and combine to form one, completely

unique ability. So now abilities that were once exclusive to a Slime for instance can now be shared, and re-mixed, creating interesting combat encounters.

For instance, an enemy could have ‘super-speed’, and ‘stealth’ and ‘heal’ applied to them, so that during a given game, the enemy will have a probability of executing either some or all of it’s sub-abilities whenever an ability is triggered, which itself will have its own occurrence probability, thus greatly increasing the randomness and complexity of the opponents faced.

Advantages:

- Good for permutation problems, as enemies can stack and interchange any combination of abilities to form wholly unique abilities at run-time and thus generate
- The Decorator Pattern is more flexible than inheritance as it offers the same amount of extensibility but at run-time instead of compile-time

Disadvantages:

- Some practical limitations regarding the type of abilities that can be performed
- This implementation would initially require a somewhat more complicated factory to randomly generate sub-abilities to assign to a randomly generated enemy, which also complicates the initialisation process of an enemy in general
- Can end up in a situation where we have many small classes, bogging down the design and somewhat defeating the ‘explosion of classes’ problem