# Object Oriented Software Engineering Assignment

**Benjamin Le**

**19798734**

## Introduction

The design and implementation of a simple turn-based combat game was the goal for this Object Oriented Software Engineering assignment. The player is tasked to fight a sequence of battles, with the use of a variety of different weapons, armour and potions. The aim of the game is to defeat the dragon before your characters health is reduced to 0.

In this report, I will be discussing what polymorphism is, the design patterns used throughout the assignment, testability and plausible alternative design choices.

## Polymorphism

Polymorphism is an important aspect of object oriented programming, as it allows programmers to perform a single action in many different ways. The implementation of interfaces and abstract classes are used in my assignment to show polymorphism, due to its ability to create many different objects that share a common interface or abstract class.

An example of this that was used in my assignment is my Enemy abstract class. In the assignment, there are a multitude of enemies, including a slime, goblin, ogre and dragon. The Enemy abstract class contains class fields and methods that all the enemies have in common. Each enemy has a name, maximum health, current health, minimum attack, maximum attack, minimum defence, maximum defence and gold. For the enemies listed above, they all share the same traits but they all have a different implementation of them. A dragon is much stronger than a slime, so for a method that returns the maximum attack of an enemy, a dragon would return a value of 30, while a slime would return a value of 5.

The reason why polymorphism is used is because for a game like this, there can be many instances where a programmer would want to further develop their game, and in doing so can create more types of enemies using the same abstract class structure or interface.

## Design Patterns

In software engineering, a design pattern is a general reusable solution to commonly reoccurring problems in software design. In this assignment there were many opportunities to incorporate many different patterns, but for my implementation of the assignment, I have chosen the following design patterns.

## Factory Method Pattern

I used the Factory Method Pattern in my assignment to create enemy objects and to create item objects. The Factory Method is a pattern that creates and returns an object by calling the constructor. This assignment required the player to undergo a sequence of battles between various monsters with different spawn rates. By creating a factory for enemies, it allows the player to fight monsters at random without it being a fixed sequence of monster fights, which in turn makes the design polymorphic. Future extensibility by adding more enemies into the game will be simple to achieve by changing the values of the initial spawn rates of the enemies and creating the newly added monster when a random number is generated.

Similarly, I chose to use the Factory Method for the creation of item objects too. An item in my program is considered to be either a weapon, armour or potion. By creating an item factory, it allows for different types of objects to be created and added to the game in the future, that will share the same common item interface. This makes future extensibility easy, as a programmer can just create a new object like a helmet for example, which will implement the same interface but be created depending on the different attributes it contains. In this example, weapons, armour and potions are all treated as items, but when it comes to creation time, it will create either a weapon object, armour object or potion object depending on the letter at index zero of the string array. The factory then returns the newly created item and is treated as an item, rather than their subclass components.

It is important for a class to have high cohesion and low coupling. The role of my EnemyFactory class is to create enemy objects depending on the randomly generated number which spawns the monster and to also modify the spawn rates after every battle. A Factory Pattern is used here to increase cohesion and decrease coupling because any changes in my Enemy class will not affect what my EnemyFactory does. They are independent and don't know much about each other.

## Decorator Pattern

The Decorator Pattern is demonstrated in my assignment through the use of enchantments. This pattern allows the player to add new functionality to an existing object without having to change anything about the original object. An example of this is for my Weapon class. Each weapon has a name, cost, minimum and maximum damage value, damage type and a weapon type.

Players are able to enchant their weapons to increase the base damage of the weapon which will then also increase the value.

Decorator Pattern is used in this scenario because it means that instead of having to create two different objects, one weapon object without the enchantment, and one with an enchantment, we can instead create a decorator class which will "wrap" the object and give it the extra stats that the add-ons provide. Imagine a program containing 5 different weapons type, and 5 different enchantment types. Instead of having to create a new class with a new weapon and new enchantment for each of the combination of weapons and enchantments, a decorator pattern just makes sense to implement in this case, since those classes are highly redundant, and it reduces the amount of classes required for the program. Keeping in mind that enchantments can also be stacked as well, which will further require more classes.

This pattern is useful for the future extensibility of the game because developers may want to add more enchantments such as WaterDamage (similar to FireDamage) that adds 5-10 damage to any weapon, or 10-15 damage to an ice-type weapon for example. The developer simply needs to create a class that extends from the abstract decorator class (Enchantment) and give the add-on some damage values and a cost.

Low coupling and high cohesion is achieved with this pattern because each add-on is a specific enchantment which is independent from the other enchantment add-ons. These classes don't know anything else about the others.

**Strategy Pattern**

I decided to implement the Strategy Pattern for my player and enemy attacks. Originally, I had them all hardcoded into my UserInterface class but after finishing the program, I noticed that there was a lot of reused code which I wanted to split up and the Strategy Pattern seemed to be a simple but effect pattern to implement for this situation.

The Strategy Pattern is an example of polymorphism and is used when there are several algorithms that have the same or a similar goal. In this assignment, a player and an enemy have the same attack pattern, in the sense that either player or enemy will attack and decrement the health of the other entity. A battle interface was created which the enemy and player attack concrete

strategy classes implement. A context class is then created to change the behaviour of the strategy depending on which one it chooses to use.

This pattern impacted the reuse of my code heavily because the way I designed my code was so that during a battle, a player will attack and then an enemy will attack right after. However, when a player uses a potion, the players attack is skipped and an enemy will attack after a potion is used (unless the damage type potion kills the enemy). My attack and potion options are split by a switch statement and so instead of having to copy paste the enemy attack sequence right after the player uses a potion, I can instead employ the enemy attack strategy using the strategy pattern which will contain the attacking sequence that the enemy does.

## Testability

Dependency Injection within my EnemyFactory class has helped achieve testability in my program. My EnemyFactory class is a class used to create enemies, and so there are no new instances of an enemy being created anywhere else in my code. This allowed me to test if the math behind my player and enemy attack/defence was correct, and if the special abilities of the enemies were working as intended. I achieve this by forcing a certain enemy to spawn 100% of the time through the use of my factory.

By splitting my program into a Model View Controller (MVC) pattern, it is easier for my code to be testable, as each class will be split into their respective folders and will make it easier to find certain classes to test on. This is known as Separation of Concerns.

A Model class represents real-world concepts that the system deals with. These classes store the information of the object and will contain simple methods such as accessors and mutators. For example: Character, Enemy, Armour, Dragon, Slime.

The View class is the user interface of the program and will handle the user input and output. It represents the visualisation of the data that model classes contain.

The Controller class contains the algorithms that my program uses to run certain functions. It updates the view whenever data has changed. For example: EnemyFactory will create a new enemy object, the view will print out the enemy object that was created along with its health and stats.

## Alternate Design Choices

While designing my code, there were many different patterns I could have used that didn't make the final cut in the assignment. These choices include

### Observer Pattern

The Observer Pattern was a design pattern I could have implemented in the battle section of my program. By using the Observer Pattern to check for an event (whether a player or enemy is dead), an observer class would be able to receive the event and perform an action accordingly.

**Pros:**

- Observer pattern loosely couples the objects that interact with one another
- Allows sending data to other objects without having to make changes in the subject or observer classes.
- For extensibility, if there are other events that need to be tracked in the future, the observer pattern allows observers to be added and removed at any point in time.

**Cons:**

- The use of an interface is forced for this design pattern.

### Template Pattern

The Template Pattern was a design pattern I could have implemented in the battle section of my program. Although it is similar to Strategy pattern, the main difference between the two patterns is when the concrete algorithm is chosen.

**Pros:**

- No code duplication
- Only a few methods need to be overridden, so code reuse is a feature when using the template design pattern since it uses inheritance.
- Allows the subclasses (player attack and enemy attack) to decide how to implement the algorithm used when attacking.

**Cons:**

- Debugging and understanding the flow in this pattern can be confusing at times
- You may be implementing an abstract method that one of the subclasses doesn't require at all.