# Decorator Pattern

Preslav Gerchev 2487136  
Dimitar Vikentiev 2464020

# Introduction

In week 3 we had to develop a Subway system shop application using the Decorator Pattern.  
The following document will reflect on the project and on the pattern itself- positive and negative consequences, reusability, maintainability, extensibility.

# The Pattern

The formal definition for the pattern is as following**:**

**“Attach additional responsibilities to an object dynamically, keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.”**

# Reusability

In terms of reusability the Decorator Pattern does pretty well since it relies on an interface and not concrete implementations. In fact the whole purpose of the Decorator pattern is to reuse one object and “decorate” it (hence the pattern’s name) during runtime with more and more functionality. How this is done is described in the “Extensibility” section.

The big advantage of this pattern is that there is one interface that is reused throughout the whole process (via polymorphism) and that allows for adding responsibilities to one object while keeping its type to that same interface.(So essentially we could do something as what is shown below)

*IOrder order=new SesameBread();  
order=new Ham(order);*

# Extensibility

In terms of extensibility the Decorator Factory pattern has both positives and negatives.

The biggest advantage of the pattern is that the object can be extended during runtime – by using composition which can be changed dynamically during runtime and not using static inheritance – which makes changes impossible during runtime. This is something that we have already seen in the Strategy Pattern where we also relied on composition rather than inheritance (and that is one of the main design principles- “favor composition over inheritance”). New decorating classes can be easily added with minimal fuss and can be used immediately.

The disadvantage of the Decorator pattern is the increased amount of classes we have in an application.

Lots of them have duplicate code with minimal difference (something that can be seen in our application – most of the IngredientDecorator classes have just few words in difference).

This is a trade-off however that is worth it – it is better to have a compositional behavior that can be changed during runtime and have an increased amount of objects than having static inheritance where extending the objects is impossible.

# Maintainability

The Decorator pattern strives for improving the maintainability of the application. By breaking the parts in different classes it allows us to keep them separated from each other and changing one will not affect the others.

The downside is if we decide that the main interface must be changed (something that should be unlikely) and the huge amount of duplicate code across the application. Most of the classes are almost similar with minor differences.

# Extra

The Decorator pattern also helps for keeping the “Open-Closed” principle which goes as following –“Classes should be open for extension but closed for modification”. To further clarify this I will give an example using our application –we’ve got an abstract class called IngredientsDecorator that will be used to “decorate” the objects. This class is closed for modification – we do not touch it, we do not change anything inside. However it is open for extension – by subclassing it (as we have with classes such as Ham, Cheese, etc.) we can add extra functionality and/or override the existing functionality of the base class.

# Implementation

In our implementation that is done in C# we have decided to go with properties rather than methods. The reason for that is although having methods such as getPrice() and getDescription() would be much easier we wanted to stay true to the C# good practices/conventions and use properties – Price and Description. However on low-level it doesn’t really matter – properties are just “syntactic sugar” for getters and setters.

# Summary

The decorator pattern finds its usage in applications where we have to change objects during runtime while keeping the same interface. This pattern helps for keeping the “aggregation over inheritance” and “open-closed” principles but it is worth noting that it should not be overused. Often it can lead to unwanted complexity, lots of small and almost unused classes and lots of duplicate code across those classes.