SWD fall 2019

Assignment 1: Design patterns

The Decorator Pattern

Group 11

|  |  |
| --- | --- |
| au554955 | Thomas Vittrup Bærentsen |
| au575154 | Marcus Bech |
| au593758 | Martin Krøjmand |
| au543213 | Marie-Louise Dybdahl Troelsen |

21-10-2019

A close up of a logo

Description automatically generated

Contents

[Introduction 2](#_Toc22569510)

[The Decorator Pattern 3](#_Toc22569511)

[Simple pizza example 5](#_Toc22569512)

[Signalprocessing example 7](#_Toc22569513)

[The software design of Decorator 8](#_Toc22569514)

[Software quality 8](#_Toc22569515)

[SOLID principles 8](#_Toc22569516)

[Comparison with other design patterns 9](#_Toc22569517)

[Adapter pattern 9](#_Toc22569518)

[Composite Pattern 9](#_Toc22569519)

[Strategy Pattern 10](#_Toc22569520)

[Mixins 11](#_Toc22569521)

[Usage today 12](#_Toc22569522)

[Conclusion 13](#_Toc22569523)

[Bibliography 14](#_Toc22569524)

# Introduction

The Decorator pattern is a structural design pattern, that makes it possible to decorate an already written implementation with new additional responsibilities. The base implementation is called a component, and the additional implementations to this component are called decorators, hence the name Decorator pattern. The clever feature of this pattern is that you can make as many decorators as you want.

In this report we will examine the Decorator pattern with its pros and cons. Additionally, we will compare the pattern to other similar design patterns. There will of course be some class diagrams and sequence diagrams to illustrate the pattern as well, and you will see some examples of the Decorator pattern in use. The examples are implemented in C#, and the project is attached to this report.

Finally, we will examine the usage of this design pattern and recap it all in a conclusion.

# The Decorator Pattern

The Decorator pattern is useful in cases where a base implementation must be able to be extended with extra responsibilities along with its primary responsibility. An example of such a situation could be a coffee shop, where the class *black coffee* needs to be extended with either milk, sugar, a decaf option or any number of other extensions (see Figure 1). In such a case, implementing a derived class for each possible permutation would quickly grow impractical when more than a few modifiers become available. An illustration of this can be seen in Figure 2.

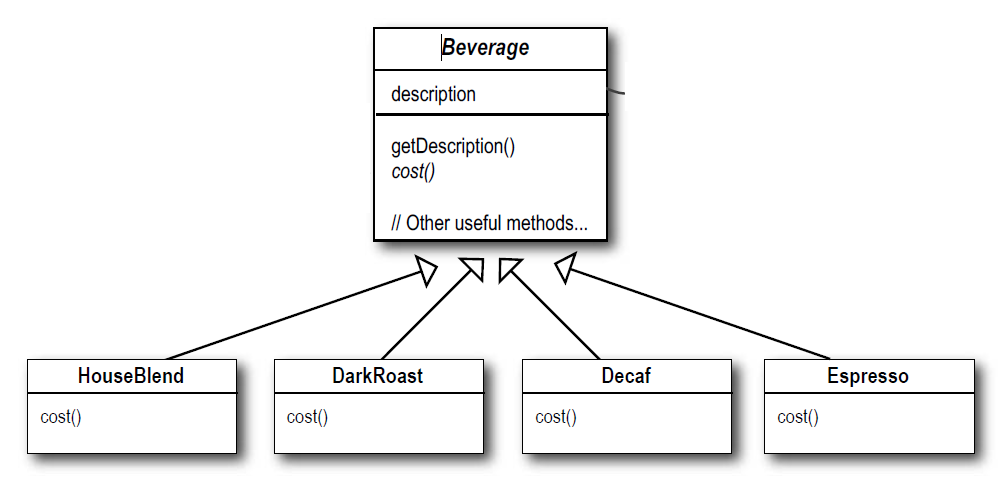


Figure 1: Extending coffee/beverage (source: (Freeman, et al., 2014 p. 82))

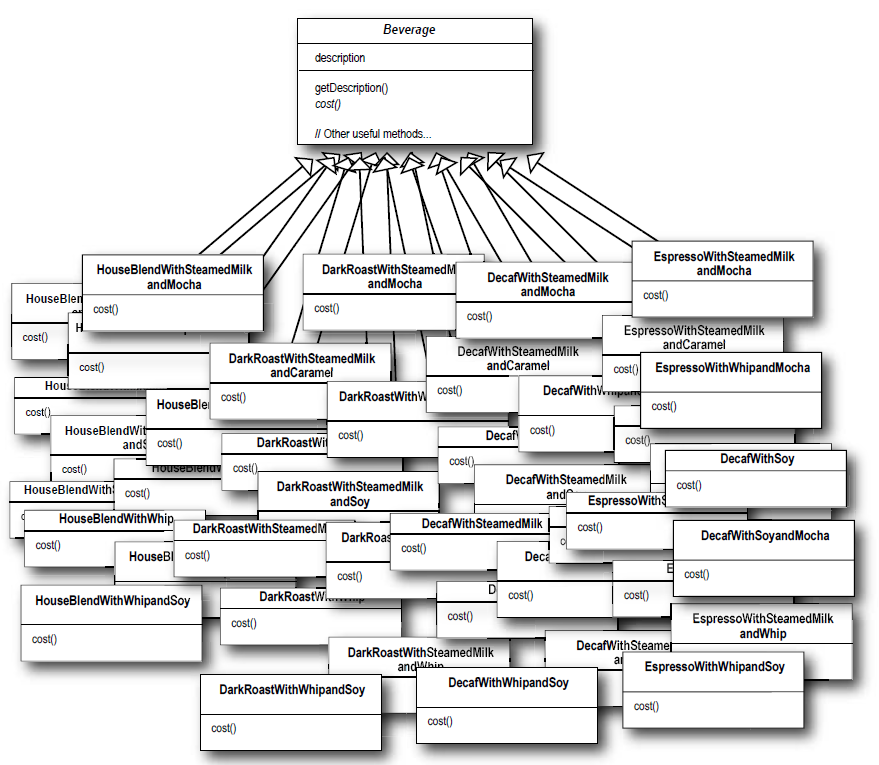


Figure 2: Extension nightmare (source: (Freeman, et al., 2014 p. 83))

The decorator pattern solves this problem by implementing *decorators* instead of separate derived classes. Each decorator class, when added to the base class, adds some bit of functionality, while still retaining what came before, including any other added decorators. A diagram showing the pattern can be seen in Figure 3, where the decorator interface or abstract class contains a reference to the Component interface, ensuring that it is always possible to “return” to the original base class, by simply following the trail of references through the added decorators. Note that the decorator interface is derived from the component base class, ensuring that its type always mirrors that of the base class.



Figure 3: Decorator UML class diagram

# Simple pizza example

With the pizza example we are building a system that calculates the price of a pizza. This pizza can have many different kinds of toppings and combinations. The system should be able to calculate the total price of the pizza based on which toppings are chosen.

The abstract component class is called Pizza and the abstract Decorator class is called AddIngredient. The Pizza class have 2 abstract functions called getCost() and getDescription(). They are overridden by the concrete Component Margarita or EcoMargarita. Here, the core implementation for these methods are defined. In the AddIngredient there is a reference of a type Pizza, which can be both a concrete component or a decorator. This technique makes it possible for the 2 functions to be wrapper functions. A wrapper function calls an other subroutine in other instance and/or level. Now you can put decorators on it again and again unlimited times (not in practice). On figure 4, there are only some of the decorators from the source code project. All the concrete decorators must implement the 2 methods to apply the abstract Pizza class.



Figure 4 UML class diagram for the Pizza example. Not all decorators included

It is pointed out that the example uses the technique of wrapper functions. It can be described through a sequence diagram:



Figure 5 The sequence diagram for the usage of Pizza Decorator with 2 function calls

Normally, you will not specify a return value like “double + double + double” in a sequence diagram, but is used here to illustrate how the decorators extends behavior. You can see that you call the decorator, which you wrap around the pizza object last. Then it continues call the next decorators until it ends at the concrete component. Here the component will return to the decorator in reverse order compare to the function call order. At last the Program class will receive the last return value.

# Signalprocessing example

In the solution attached to this assignment, there is a Signalprocessing example. It is similar to the pizza example. They both use the Decorator pattern. In the Signalprocessing example, there are some signal sources. In this case it is WindmillProcessing and Carprocessing. These are concrete components. Then you can add some signal manipulators like gain, high pass filter and lowpass filter. These are the concrete decorators. SignalProcessing is the abstract component class, and AddManipulator is the abstract decorator class. The order of the decorators has an influence. If you add a gain on 10 and add a lowpass filter, you will get another result than switching the order. There is a class diagram on figure 6.



Figure 6 Class diagram of SignalProcessing

# The software design of Decorator

## Software quality

The Decorator pattern contributes to the classic software quality criteria. First of all is it reusable, because we reuse existing code by wrapping around it. It is also testable, if you are making your test correct. If you have tested the concreate component right with all the scenarios, there is no reason to test it again when you put decorators around. You must be aware that your tests are adequate because you expose the concrete component with a new situation by wrapping a decorator around. You can also say that you increase the readability by avoiding class explosion. In any case, it is not manageable with too many classes.

## SOLID principles

The Decorator pattern apply some of the SOLID’s design principles. These principles ensure a good software quality, if they are applied correctly. The design applies 3 out of 5 SOLID-principles.

**S**ingle-Responsibility principle can be used in this pattern. The decorators are often small classes with only one reason to change. The fact that you have the capability to add an unlimited number of decorators to a behavior, there is ample opportunity to separate the functionality in multiple classes. Of course, if you make bad and heavy decorators, it is possible to not apply the Single-responsibility principle for your application.

**O**pen-Closed Principle is also used in this pattern. The whole point of the decorator pattern is that you can always add extra implementations without editing the existing implementation. It fulfills the open-closed principle because you don’t modify the component class, instead you add some extra implementation to make new behaviors. Therefore, you avoid copying paste implementation code by reusing existing code.

**L**iskov Responsibility principle is a way to work with a class without knowing what it specifically is. It is fulfilled with polymorphism. The decorators don’t know if they call a method in another decorator, or they call the method in the base component concrete class. They should not know it, because it would make a looser coupling, and you can therefore add as many decorators as you want (unless you get stack overflow). The postcondition for the inherited method will always be stronger and stronger by adding new decorators. It could for example be incrementing the prize for a pizza by adding new ingredients (decorators).

# Comparison with other design patterns

There are some design patterns that have some similarities with the Decorator Pattern. In this section some of the design patterns that look most like the Decorator Pattern will be examined and compared. The decision is chosen by the group. The similarities and the differences will be described.

## Adapter pattern

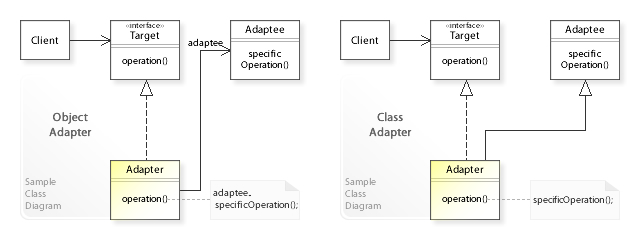


Figure 7: Adapter UML class diagram for the 2 types of design (source: (Wikipedia))

The Adapter pattern transforms objects’ interface to a new interface. The Decorator pattern changes the object’s responsibility to a new one, but it still applies to the abstract component class. Both design patterns have different intentions. The adapter changes the interface. Det Decorator changes the behavior. They seems to be very different, but they are both called wrappers. A wrapper is a function that will continue call other objects or functions by calling the wrapper. It is like the decorators that are continued calling the other decorators until they reach the component. Here it will often return in the previous path reversed. The Adapter wraps around an Adaptee. The Adaptee is the class, the client wants to convert to apply the target interface. So, they are both using the wrapper technique, but the intention is very different.

## Composite Pattern

If you look at the UML diagram for the Composite Pattern and the Decorator pattern, there are a lot of similarity. They both have a class which both inherited from and holds a reference to Component. It makes it possible to use the recursive principle like the Decorator Pattern. You can always wrap an extra decorator around. The main difference is how they are structured. The Composite Pattern creates a tree hierarchy with nodes and leaves. The nodes are called composites and can be seen as a root for each node’s perspective. It is a recursive way of thinking like a Binary Search Tree. The leaves are the bottom of the hierarchical structure. The Decorator Pattern on the other hand has a structure like an onion. In the middle, there is a concrete component, and there are decorators around it like shells in an onion. It is two different ways to structure functionality and avoid duplicating code. If you want to make a program where you have a menu with sub menus and sub-sub menus, then the Decorator Pattern is not the best choice for your design. The Composite Pattern will be an obvious choice for your software design. If you want to make graphic controls, where you for example begin with a label and then extends it with borders around, a background, shadows and so on: The Decorator Pattern will be a good choice for your design.

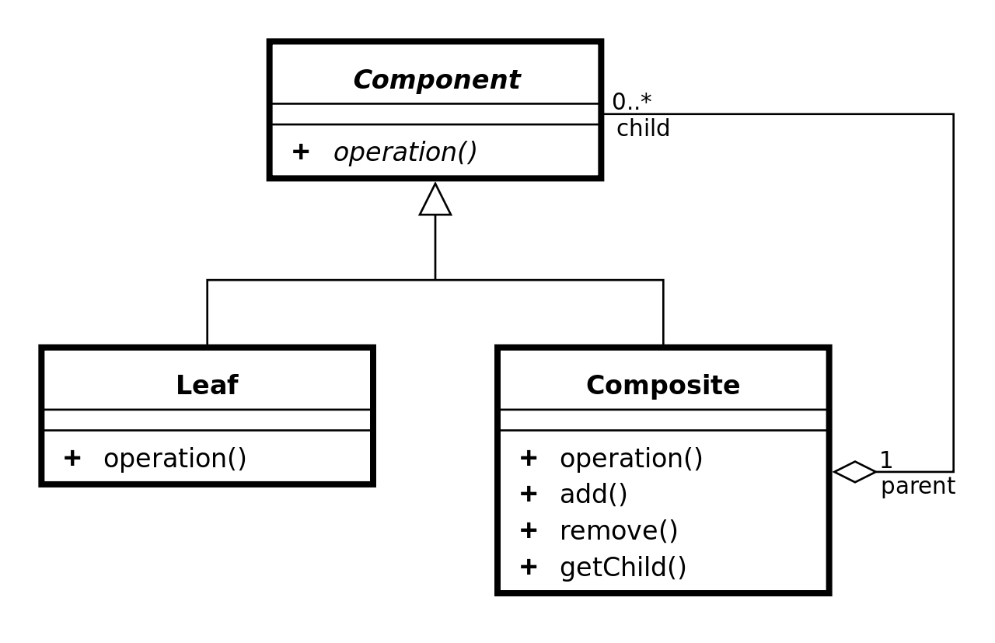


Figure 8: Composite UML class diagram (source: (Wikipedia))

So; the right pattern depends of what you are extending. The disadvantage of the Composite Pattern is that you in some implementation have the add(), remove() and getChild() in the base class Component because you will force the composite classes to override these methods. If the composite classes don’t have implementations to these methods, then the design doesn’t apply the Composite Pattern correct. On the image on Figure 7, the Component class doesn’t have the 3 methods, but in some implementations they have. It is a problem because the leaves will inherit from the Component, and they are therefore forced to implement the methods. It doesn’t make any sense that the leaves have a getChild() method, because they never have a child, when they are leaves! It is a good example of not applying the Interface Segregation Principle. The component class is not an interface but an abstract class, but it is still a relevant principle here. There is a fat abstract class called Component that force all the inherited classes to inherit the content even though it doesn’t make sense for all the inherited classes. It is the same problem with the Decorator Pattern. Often, you will add extra methods in the abstract Component class, which is only relevant for the concrete decorators or concrete components. All the concrete components and concrete decorators will inherit it, if it is placed in the abstract Component class.

## Strategy Pattern

The Decorator Pattern has some similarity with the Strategy Pattern. You can change the functionality of an object run time in both design patterns. In Strategy Pattern you change a whole algorithm or method for an object run time. In Decorator you change the behavior of an object by wrapping extra functionality around. Here is the different. If you overlook the fact what you can add decorators to an object, the dynamically exchanging between concrete components looks very similar to the Strategy Pattern. The different is the Strategy Pattern use interfaces to apply polymorphism. In the Decorator pattern you have an abstract class. You can I both cases switch between implementations, but you can have default implementation in a abstract class if you really want. The disadvantage for Strategy pattern is you can end up with a lot of classes with overlap. The Disadvantage for Decorator is that you can end up with small classes, and the client is responsible to decorate the object, correctly. As a client of an object designed as the Decorator Pattern, there is not a black box view on the object like in the Strategy Pattern. In the Decorator Pattern, the clients should build it up by themselves.



Figure 9 Class diagram of the concept of Strategy Pattern. It is inspired by (Wikipedia)

## Mixins

The Mixins is a way to extend the functionality by inheritance. The Decorator pattern extends functionality by having a reference of type abstract component. It could be either to a concrete component or decorator cf. polymorphism. You can say it is through composition in Decorator Pattern. In the Mixins you are often needed to use multiple inheritance. That doesn’t work in C# and Java, but in other languages like C++, you can use it. The Mixins is great to have a lot of different classes to inherit. It means that we don’t have fat abstract classes or interfaces. In that way the Mixins apply to the interface Segregation Principle. Abstract classes are not interfaces, but they are so similar besides the abstract classes can have default implementation. This is different from the Decorator Pattern that have one abstract class (abstract component) which the concrete Decorators and components inherit.

# Usage today

The Decorator pattern is used in different software projects and frameworks today. Primarily, it is used in frameworks, where elements can be created in many different ways. An example is WPF where many Frameworks elements derived from Decorator. It contains ButtonChrome and other things. It is useful because the client of WPF can use it to make own Control Templates.

There are other usage of it today in the many different applications around the world, but it is in the frameworks you will see it often, primarily.

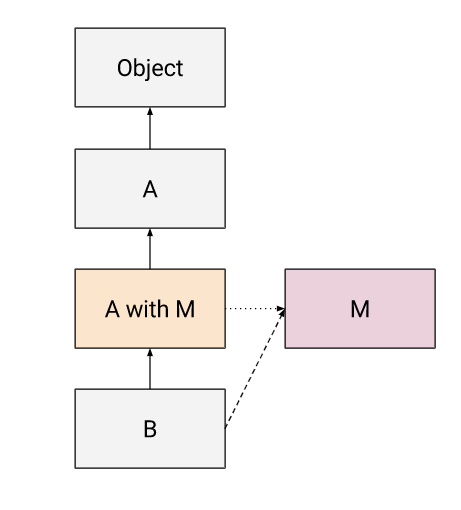


Figure 10: Class diagram for the general Mixins pattern. Here, M is the Mixin. (Source: (Fagnani, 2015))

# Conclusion

The Decorator pattern is a good way to apply the Open-closed principle. You extend the functionality by wrapping additional behavior to make new responsibilities. It is like the layers in an onion. Unfortunately, it sets some limitations, when you have to extend it this way. It means a structure based on one-dimensional layers. Therefore, you can also use other similar design patterns like the composite pattern, which set up a tree hierarchy structure. It is useful in some situations, and in other situations the Decorator pattern is better. It depends! The Decorator apply SOL in SOLID, but it is not applying the interface Segregation Principle. Often the clients will put extra methods in the abstract Component class. The intention for these methods will often be on the decorators or the concrete components. The fact is that every class inherit the methods. On the other hand, you can reuse a lot of code, and you can avoid class explosion by mix the decorators in your desired order. If you are doing it correct, you can avoid testing the same core functionality again and again by reusing it via the Decorator Pattern.

# Bibliography

**Fagnani, Justin. 2015.** "Real" Mixins with JavaScript Classes. *Justin Fagnani: Ramblings on programming and other things.* [Online] December 21, 2015. [Cited: October 16, 2019.] http://justinfagnani.com/2015/12/21/real-mixins-with-javascript-classes/.

**Freeman, Eric, et al. 2014.** *Head First: Design Patterns.* s.l. : O' Reilly Media, 2014.

**Wikipedia.** Adapter pattern. *Wikipedia.* [Online] [Cited: October 8, 2019.] https://en.wikipedia.org/wiki/Adapter\_pattern.

**—.** Composite pattern. *Wikipedia.* [Online] [Cited: October 8, 2019.] https://en.wikipedia.org/wiki/Composite\_pattern.

**—.** Strategy Pattern. *Wikipedia.* [Online] [Cited: October 11, 2019.] https://en.wikipedia.org/wiki/Strategy\_pattern.