Decorator (Wrapper) Pattern Presentation

Leighton Chen, Yash Patel, Jathusan Thiruchelvanathan, Priyansh Vyas

Summary

The Decorator pattern is a design pattern that essentially allows users to add new functionality to existing objects without needing to alter its original structure. That is, this pattern relies on a decorator class which wraps the original class and provides additional functionality while keeping class methods signature intact.

Intended Use

- Allows you to modify an object dynamically, as opposed to creating new objects with new desired attributes
- Works independently from other objects of the same class
- Used when you want the capabilities of inheritance using subclasses, but need to add additional functionality during runtime
- More flexible than inheritance, simplifies code because you add functionality using many simple classes rather than several complex inherited classes
- Rather than rewrite old code, extend functionality with new code

Structure

- A new decorator class “wraps” the original class
- The “wrapping” is usually done by passing in a reference to an instance of the Component class into the constructor of the Decorator class
- The passed in reference is then used to derive the behaviour of the “decorated” object by adding to its existing functionality

Vocabulary

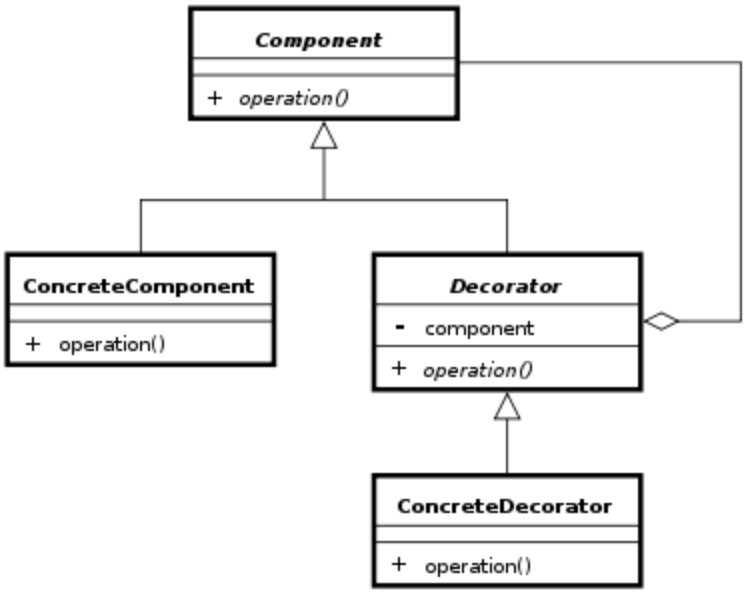
Decorator: The act of “decorating” an object refers to modifying an existing instantiation of a class by passing a reference to that object into a decorator object. The decorator object then defines its own modified behaviour which is derived from the behaviour of the existing object, but modifying it based on the type of decorator.

Component: The interface that both ConcreteComponent and Decorator classes inherit from. This class will have methods defined by the characteristics of the real world object that is modeled. The inherited classes will implement these methods based on what they represent.

ConcreteComponent: The concrete class that implements the Component class. This will usually be the base object in which no decorators have been applied yet.

Decorator: The abstract class that implements the Component class. Since there can be multiple type of decorations applied to an object, it is necessary to define an abstract class. The constructor is defined to take in a reference to an object of the Component class in which subclasses of the Decorator class can define behaviour to modify the object.

ConcreteDecorator: Different types of "decorations" that can be applied to a Component object can define their behaviour in different ConcreteDecorator classes.



Pros

1) One of the most important benefits of the decorator pattern is that it is more flexible than inheritance, making it a potentially easier alternative for extending functionality.

2) The decorator pattern also allows behaviour modification at runtime, rather than going back and having to change old code

Cons

1) The decorator pattern can make the process of initializing the component to be more complicated than necessary (i.e if you have a ton of decorators)

2) The decorators can lead to a system that contains many “little objects” that can look extremely similar (e.g pizza toppings). These objects would differ only in the way that they are connected, not by their class or variable values - this can be hard for the developer to maintain, or even a new developer to learn.

Performance

- For each additional behaviour attribute that is desired to be added onto the base object, only one new ConcreteDecorator class needs to be defined
- If using inheritance instead, every possible combination of multiple attributes must be defined as a new class
- For above, behaviour is duplicated for same attributes and must be defined for each different class

NFPs (Non-functional Properties):

The decorator pattern can help to improve non-functional properties in several systems and applications. Specifically, the decorator design pattern is extremely scalable as it is built on the fundamental concept of “decorating” an existing base object. This allows for the system to always be adapted to meet scope requirements by building on top of existing objects as needed and helps to facilitate and enforce that components are separated effectively.

Similarly, the decorator pattern is extremely evolvable over time as you can always introduce new object types which build on top of existing ones and make the system evolve to fit requirements and needs. Additionally, the decorator pattern is extremely simple and efficient to use as it is allows for behaviour modification during runtime, making systems arguably more flexible than basic inheritance allows for.