**Decorator Design Pattern**

**Definition:**

In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

**Structural Design Pattern.**

1. Decorator wraps an object in itself so client object does not need to change.
2. Alternative way to subclassing for extending functionality of existing classes.

UML Diagram :

Component

process() : void

comp.process()

addnewProcess()

Decorator

process():void

addnewProcess()

ConComponent

process() : void

Reference of component hold by Decorator

**Implementation :**

1. Component defines interface which already used by client.
2. ConcreteComponent implements the Component.
3. Decorator implements the Component and also holds the reference to concrete component.
4. In decorator we can provide additional new behaviour.

Decorator can be abstract and depend on the subclasses to provide functionality.

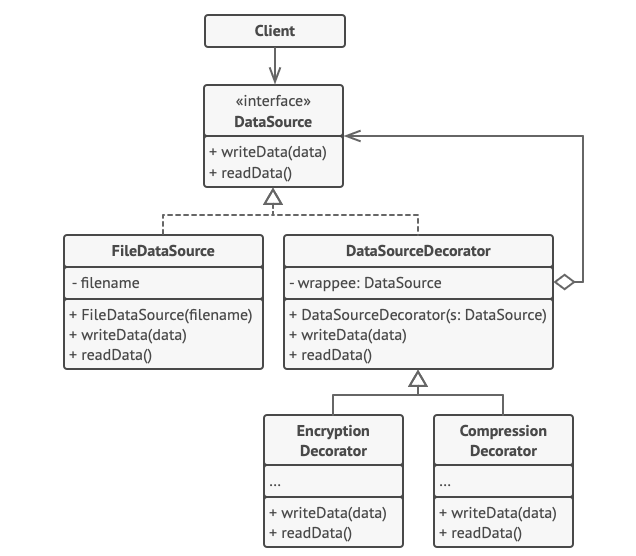
Note : Equals and hashCode in decorator , we have to decide if decorated object is equal to same instance without

decorator.

Example from JDK : Java I/O Package.

For example java.io.BufferedOutPutStream class decorates java.io.outputStream object and add buffering to write operations.

**Example :**

****

source = new FileDataSource("somefile.dat")

source.writeData(salaryRecords)

// The target file has been written with plain data.

source = new CompressionDecorator(source)

source.writeData(salaryRecords)

// The target file has been written with compressed

// data.

source = new EncryptionDecorator(source)

// The source variable now contains this:

// Encryption > Compression > FileDataSource

source.writeData(salaryRecords)

// The file has been written with compressed and

// encrypted data.

**Part1: Code Implementation :**

One shop has Sandwich with WhiteBread and BrownBread.

After some time there are more variations added like : Addition of White and BrownBread with Corn, cheese and both.

Sandwich has two subclasses : WhiteBread and BrownBread.

We have Sandwich decorator and its base subclasses : CornSandwich , CheeseSandwich.

**Its based on Single Responsibility Pattern : where code can applicable for extension but not for modification.**

Notes :

1. The Decorator must be of the same type of object, which they are decorating. This can be achieved either by implementing the [interface](http://javarevisited.blogspot.sg/2013/04/10-abstract-class-and-interface-interview-question-java-answers.html#axzz4pk4W5ie3) of the object or by extending an [abstract class](http://www.java67.com/2012/09/what-is-difference-between-interface-abstract-class-java.html) of the original class.

2. The decorator pattern is based on Composition, which means it needs an original object to decorate it. This is achieved by creating a constructor on the decorator class that accepts a base type of original object. like in this example constructor of CheeseDecorator accepts Sandwich object.

**Compare :**

**Composite :**

Decorator is “add to” existing behaviour of existing object.

Composite : Object Aggregation only

Decorator : work with only one component.

Composite : Supports any number of components in aggregation.

**Relations with Other Patterns**

1. [Adapter](https://refactoring.guru/design-patterns/adapter) changes the interface of an existing object, while [Decorator](https://refactoring.guru/design-patterns/decorator) enhances an object without changing its interface. In addition, Decorator supports recursive composition, which isn’t possible when you use Adapter.
2. [Adapter](https://refactoring.guru/design-patterns/adapter) provides a different interface to the wrapped object, [Proxy](https://refactoring.guru/design-patterns/proxy) provides it with the same interface, and [Decorator](https://refactoring.guru/design-patterns/decorator) provides it with an enhanced interface.
3. [Chain of Responsibility](https://refactoring.guru/design-patterns/chain-of-responsibility) and [Decorator](https://refactoring.guru/design-patterns/decorator) have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

The CoR handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various Decorators can extend the object’s behavior while keeping it consistent with the base interface. In addition, **decorators aren’t allowed to break the flow of the request.**

1. [Composite](https://refactoring.guru/design-patterns/composite) and [Decorator](https://refactoring.guru/design-patterns/decorator) have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.

A Decorator is like a Composite but only has one child component. There’s another significant difference: Decorator adds additional responsibilities to the wrapped object, while Composite just “sums up” its children’s results.

However, the patterns can also cooperate: you can use Decorator to extend the behavior of a specific object in the Composite tree.

1. Designs that make heavy use of [Composite](https://refactoring.guru/design-patterns/composite) and [Decorator](https://refactoring.guru/design-patterns/decorator) can often benefit from using [Prototype](https://refactoring.guru/design-patterns/prototype). Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
2. [Decorator](https://refactoring.guru/design-patterns/decorator) lets you change the skin of an object, while [Strategy](https://refactoring.guru/design-patterns/strategy) lets you change the guts.
3. [Decorator](https://refactoring.guru/design-patterns/decorator) and [Proxy](https://refactoring.guru/design-patterns/proxy) have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a Proxy usually manages the life cycle of its service object on its own, whereas the composition of Decorators is always controlled by the client.

Reference Links :

https://www.codiwan.com/decorator-design-pattern-real-world-example-java/

https://www.java67.com/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html

https://refactoring.guru/design-patterns/decorator