Decorator Design Pattern:

**Other name (if any)**

The Decorator Pattern is also known as **Wrapper.**

**What it does**

It is used to reduce class explosion (It is situation of making huge amount of classes to create different versions of the system for different combination and permutations of the features.). Decorator design pattern is used in a situation where we have to add additional features (means extend the current system) and have to make different versions of the existing system  using permutations and combination of different features. Abstract decorator class has "has a" and "is a" relationship with abstract base class. Means it will extends base class and also has a object of this base class inside it.

**Where to use**

1. When you want to transparently and dynamically add responsibilities to objects without affecting other objects.  
2. When you want to add responsibilities to an object that you may want to change in future.  
3. In **Coffee or Food Customization.**  
4. When you want to adhere to the Open/Closed Principle by allowing classes to be easily extended with new functionality without modifying existing code.  
5. When you want to promote a modular design by breaking down complex behaviors into smaller, more manageable decorators, facilitating code maintenance and reuse.  
6. When you want to integrate with third-party libraries or frameworks by adding additional functionality to their classes without modifying their source code, ensuring compatibility and ease of updates.

**Steps**

1. Abstract base/component class: Create an abstract class of the system.   
2. Concrete base/component class: It will extend the abstract base class based on the core different version of the system.  
3. Abstract decorator class: Create an abstract decorator class for features.  
4. Concrete decorator class: It will extends the abstract decorator class. These concrete classes are classes of different features.  
5. Client code: Objects of different versions of the system will be created here using abstract base class, concrete base class and concrete decorator class. For example  
AbstractBaseClass obj = new ConcreteDecoratorClass2( new ConcreteDecoratorClass1( new ConcreteBaseClass() ) );

**Advantages**

1. The decorator pattern follows the open-closed principle, which states that classes should be open for extension but closed for modification. This means you can introduce new functionality to an existing class without changing its source code.  
2. It allows you to add or remove responsibilities (i.e., behaviors) from objects at runtime. This flexibility makes it easy to create complex object structures with varying combinations of behaviors.  
3. Unlike traditional inheritance, which can lead to a deep and inflexible class hierarchy, the decorator pattern uses composition. You can compose objects with different decorators to achieve the desired functionality, avoiding the drawbacks of inheritance, such as tight coupling and rigid hierarchies.  
4. Encourages the creation of reusable decorators that can be applied to different objects to provide similar enhancements, reducing code duplication.  
5. Supports incremental building of object behavior by adding decorators in a layered fashion, allowing for the gradual enhancement of functionality.  
6. Enables the easy removal of added features or responsibilities by removing corresponding decorators, without affecting the core object.

**Disadvantages**

1. As you add more decorators to an object, the code can become more complex and harder to understand. The nesting of decorators can make the codebase difficult to navigate and debug, especially when there are many decorators involved.  
2. When using the Decorator pattern, you often end up with a large number of small, specialized decorator classes. This can lead to a proliferation of classes in your codebase, which may increase maintenance overhead.  
3. The order in which decorators are applied can affect the final behavior of the object. If decorators are not applied in the correct order, it can lead to unexpected results. Managing the order of decorators can be challenging, especially in complex scenarios.  
4. Debugging decorated objects may be more challenging, as it may require tracing through multiple layers of decorators to understand the complete behavior of the object.  
5. Implementing decorators and managing their interactions can require additional development time and effort, especially for complex object hierarchies.

**Code**

Coding Concept

**Difference with similar pattern**

May be chain of responsibility and builder design pattern but [[later]]

**Diagram**

JavaTpoint