**Use Case**:Imagine you are creating an online photo editing application. Users can apply various filters and effects to their photos. The Decorator Pattern allows you to add multiple filters dynamically to a photo without changing the original photo object.

**Working:**

Interfaces & Abstract Class

Image Interface: This is our foundation. It defines two essential methods

describe: Returns a description of the image.

show: Displays the image. In our case, this is simulated with a printout.

1. **ImageDecorator Abstract Class**: This class builds on the `Image` interface. It keeps a reference to an `Image` object in a protected field named `image`. The describe and show()methods here simply call the corresponding methods of the wrapped image. This setup allows decorators to extend the behavior of the image without altering its core functionality.

2. **Concrete SimpleImage:** This class implements the `Image` interface and represents a basic image. It has a name, and its describe and `show methods provide straightforward information about the image, like the file name.

3. **Decorator Classes**: BnwFilter and SepiaEffect: These are our decorators that extend the ImageDecorator class. Each one represents a specific filter—Black and White and Sepia, respectively.

When you create an instance of either filter, you pass in an existing `Image` object . Inside these classes, the describe method is overridden to add details about the filter to the image’s original description.

The show method first calls the base show method of the wrapped image to display it, then adds a message about the filter being applied. This keeps everything organized and modular.

4. **Main Class:Main**: Here’s where we bring everything together. The Main class creates a SimpleImage object, which represents a vacation photo.

It then wraps this image with a BnwFilter decorator, showing the result and applying the Black and White filter.

Next, it further wraps the already-filtered image with a SepiaEffect decorator. This allows us to see the description and effects of both filters being applied together.

**Benefits of Using the Decorator Pattern**

1. Flexibility: Adding new filters is easy! You can simply create new decorator classes that extend ImageDecorator allowing for endless possibilities without modifying existing code.

2. Reusability: The SimpleImage class can be reused with different filters seamlessly. You can combine and apply filters in various ways without altering the core image class.

3. Dynamic Modification: You can add or remove filters at runtime by wrapping the image object with different decorators. This gives you great flexibility in how images are processed.

**Conclusion**

In summary, the Decorator Pattern is a powerful way to add functionalities like image filters dynamically. It maintains a clean separation of concerns, promotes reusability of the base image class, and provides a flexible framework for enhancing images with various effects. This way, you can create rich image processing applications without cluttering your original codebase.