Adapter Design Pattern

Bridging the incompatibility between two interfaces or classes, allowing them to work together.
The Adapter pattern resolves the problem of interface or class incompatibility, enabling collaboration between components with differing interfaces. It is useful in scenarios where you need to integrate existing code or third-party libraries into your system without modifying their interfaces.

Key Components:
- **Target Interface**: Defines the interface expected by the client.
- **Adaptee**: Represents the existing class or interface that needs adaptation.
- **Adapter**: Implements the target interface, delegates requests to the adaptee, and acts as a bridge.

Advantages:
- **Compatibility**: Enables collaboration between incompatible interfaces or classes.
- **Reuse**: Reuses existing classes without modifying their code.
- **Encapsulation**: Isolates the adaptational logic in the adapter class.
- **Flexibility**: Supports adding multiple adapters for different interfaces.

Disadvantages:
- **Complexity**: Introducing adapters can increase code complexity.
- **Runtime Overhead**: May introduce a slight performance overhead due to delegation.

Use Cases:
- **Legacy Code Integration**: Making legacy components compatible with modern systems.
- **Library Compatibility**: Adapting third-party libraries to fit your application's interfaces.
- **API Versioning**: Maintaining backward compatibility while upgrading APIs.
- **Language Translation**: Converting data or messages between languages or formats.
- **User Interface Elements**: Adapting user interface components for different platforms.

Decorator Design Pattern

Adding behavior or responsibilities to objects dynamically without altering their code.

Key Components:
- **Component**: Defines the interface for objects that can be decorated.
- **Concrete Component**: Implements the component interface, the base object being decorated.
- **Decorator**: Maintains a reference to a component and implements the component interface.
- **Concrete Decorators**: Extend the decorator class, adding or modifying behavior.

Advantages:
- **Composition over Inheritance**: Provides a flexible alternative to subclassing.
- **Dynamic Behavior**: Allows adding or changing behavior at runtime.

Disadvantages:
- **Complexity**: Introducing multiple decorators can make the system more complex.

Use Cases:
- **Text Formatting**: Adding styles (bold, italic) to text in a word processor.
- **Graphic Objects**: Enhancing graphics with borders, shadows, or transparency.
- **Coffee Ordering**: Customizing coffee with condiments (e.g., milk, sugar).