**Structural Design Patterns**

Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

**Adapter :**

Adapter Design Pattern is a structural design pattern used to enable the collaboration between two incompatible interfaces. It allows the objects with different interfaces to work together, making it easier to reuse existing code or integrate external components into an application without modifying their source code.

Ex: When working with third-party libraries or APIs that do not match the expected interface of your application, you can use adapters to make the two systems work together without modifying either the library or your application's code.

Ex: Adapters can be employed to convert data from one format to another. For instance, when reading data from a file, the adapter can convert it into the appropriate data structures used by the application.

**Bridge :**

The Bridge pattern is designed to decouple an abstraction from its implementation, allowing both to vary independently. It achieves this by creating two separate hierarchies: one for the abstraction (interface or abstract class) and another for the implementation (concrete class). By doing so, changes to one hierarchy do not affect the other, providing a flexible and extensible system.

Ex: Consider another example of the Bridge Design Pattern in the context of a music player application.   
Abstraction Hierarchy: MusicPlayer (Abstraction), BasicMusicPlayer (Refined Abstraction), AdvancedMusicPlayer (Refined Abstraction)

Implementation Hierarchy: AudioOutput (Implementation), InternalSpeaker (Concrete Implementation), ExternalSpeaker (Concrete Implementation)

In this example, the MusicPlayer hierarchy represents different types of music players with varying levels of features, while the Implementation hierarchy represents different audio output options available for the music players.

We can create any combination using this structure easily.

**Composite :**

The Composite design pattern is a structural design pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly, making it easier to work with complex hierarchies.

Ex: Consider a file system as an example. In this case, a file can be a leaf element, and a directory can be a composite element. Both files and directories share common operations such as listing, moving, or deleting. With the Composite pattern, you can treat a file and a directory uniformly using the common component interface. This allows you to work with both individual files and entire directories with the same set of operations, making the code more manageable and flexible.

Ex: we want to create a simple GUI that contains a main window (composite) with two buttons (leaves) and a sub-panel (composite) that holds a text box (leaf) and a submit button (leaf). Without the Composite pattern, we would need separate code to handle individual elements and containers. With the pattern, we can treat both elements and containers uniformly, making the code more cohesive.

**Decorator :**

The Decorator design pattern is a structural design pattern that allows behavior to be added to an object, dynamically, without affecting the behavior of other objects from the same class.

Ex: Consider a text editor application where you have a basic TextComponent representing simple text. You can then create ConcreteDecorator classes like BoldDecorator, ItalicDecorator, and UnderlineDecorator. By combining these decorators, you can dynamically add different styles (bold, italic, underline) to the text at runtime without changing the original TextComponent class.

Ex: Imagine you have a base class called Coffee that represents a basic coffee with properties like cost and description. The Coffee class has a method for getting the cost and another method for getting the description. Now, you want to provide the customers with the option to customize their coffee orders with various add-ons, such as Milk, Sugar, and Whipped Cream. You also want to give them the flexibility to mix and match these add-ons.