Design patterns?

1- Creational design patterns?

- Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.
- Singleton => Ensure a class only has one instance, and provide a global point of access to it.
- One of the main advantages of using the singleton pattern is that it can reduce the memory usage of your application. By creating only one instance of a class, you can avoid allocating and deallocating memory for multiple objects of the same type.
- Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.
- Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.
- The singleton class provides a global access point to get the instance of the class.

2- Structural patterns?

- Adapter or Wrapper. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Decorator or Wrapper. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

3- Behavioral patterns?

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.
- Observer. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Inherited Widget:

- Type: Structural
- Explanation: Inherited Widget is a Flutter-specific design pattern that allows you to propagate information down the widget tree efficiently. It's often used for passing data that needs to be accessible by multiple widgets without having to pass it explicitly.

```
class MyInheritedWidget extends InheritedWidget {
  final String data;

  MyInheritedWidget({required this.data, required Widget child})
      : super(child: child);

  static MyInheritedWidget of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<MyInheritedWidget>()!;
  }

  @override
  bool updateShouldNotify(MyInheritedWidget oldWidget) {
    return oldWidget.data != data;
  }
}
```

Singleton:

- Type: Creational
- Explanation: Ensures that a class has only one instance and provides a global point to access it.

```
class Singleton {
  static Singleton? _instance;

  factory Singleton() {
    _instance ??= Singleton._();
    return _instance!;
  }

  Singleton._();
}
```

Dependency Injection:

- Type: Structural
- Explanation: It's a design pattern where a class receives its dependencies from external sources rather than creating them itself. This promotes code reusability, testability, and flexibility.

```
class SomeService {
  // Some service implementation
}

class MyClass {
  final SomeService _service;

  MyClass(this._service);

  // Use _service in the class
}
```

Adapter:

- Type: Structural
- Explanation: Allows the interface of an existing class to be used as another interface. It's often used to make existing classes work with others without modifying their source code.

```
class OldClass {
  void doSomethingOld() {
    // Old implementation
  }
}

class Adapter {
  final OldClass _oldClass;

  Adapter(this._oldClass);

  void doSomethingNew() {
    _oldClass.doSomethingOld();
    // Add new functionality if needed
  }
}
```

Decorator:

- Type: Structural

- Explanation: Attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

```
abstract class Component {
  void operation();
}

class ConcreteComponent implements Component {
  @override
  void operation() {
    // Basic operation
  }
}

class Decorator implements Component {
  final Component _component;

  Decorator(this._component);

  @override
  void operation() {
    _component.operation();
    // Additional operation
  }
}
```

Observer:

- Type: Behavioral
- Explanation: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
class Observer {
  void update(String message) {
    // Update logic
  }
}

class Subject {
  List<Observer> _observers = [];

  void addObserver(Observer observer) {
    _observers.add(observer);
  }

  void removeObserver(Observer observer) {
    _observers.remove(observer);
  }

  void notifyObservers(String message) {
    for (var observer in _observers) {
      observer.update(message);
    }
  }
}
```

Repository Pattern:

- Type: Architectural
- Explanation: Separates the logic that retrieves data from the underlying storage, allowing a clean separation between domain code and data access code.

```
class UserRepository {
  Future<User> getUserById(String userId) {
    // Logic to fetch user from storage
  }

  Future<void> saveUser(User user) {
    // Logic to save user to storage
  }
}
```