### **Patrones de Diseño**

* **Creacionales:** Se ocupan de la creación de objetos.
* **Factory:** Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases decidan qué instancia crear.
* **Abstract Factory:** Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
* **Builder:** Separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones.
* **Singleton:** Garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.
* **Prototype:** Crea nuevos objetos clonando un objeto existente.
* **Estructurales:** Se ocupan de componer objetos para formar estructuras más grandes.
* **Adapter:** Convierte la interfaz de una clase en otra que el cliente espera.
* **Bridge:** Separa una abstracción de su implementación para que ambas puedan variar independientemente.
* **Composite:** Compone objetos en estructuras de árbol para representar jerarquías parte-todo.
* **Decorator:** Añade responsabilidades adicionales a un objeto dinámicamente.
* **Facade:** Proporciona una interfaz unificada a un conjunto de interfaces en un subsistema.
* **Flyweight:** Usa compartición para sustancialmente reducir la cantidad de objetos.
* **Proxy:** Funciona como sustituto de otro objeto para controlar el acceso a él.
* **De comportamiento:** Se ocupan de la asignación de responsabilidades entre objetos.
* **Strategy:** Define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
* **Observer:** Define una dependencia uno a muchos entre objetos de manera que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
* **Command:** Encapsula una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, colas o registros de solicitudes, y soportar operaciones que se deshacen.
* **Iterator:** Proporciona una manera de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
* **Template Method:** Define el esqueleto de un algoritmo en un método, permitiendo que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
* **State:** Permite a un objeto alterar su comportamiento cuando su estado interno cambia.
* **Visitor:** Representa una operación a ser realizada sobre elementos de una estructura de objetos.
* **Mediator:** Define un objeto que encapsula cómo un conjunto de objetos interactúa.
* **Memento:** Captura y externaliza el estado interno de un objeto de manera que el objeto pueda ser restaurado a este estado posteriormente.
* **Chain of Responsibility:** Enlaza objetos en una cadena y pasa una solicitud a lo largo de la cadena hasta que un objeto maneje la solicitud.

### **Patrones de Arquitectura**

* **MVC (Model-View-Controller):** Separa los datos (modelo), la interfaz de usuario (vista) y la lógica de control.
* **MVP (Model-View-Presenter):** Similar a MVC, pero el presentador actúa como intermediario entre la vista y el modelo, controlando el flujo de datos.
* **MVVM (Model-View-ViewModel):** Enlaza las vistas a los modelos a través de viewmodels, que exponen los datos de una manera conveniente para la vista y contienen la lógica de presentación.
* **Microservicios:** Descompone una aplicación en pequeños servicios independientes que se comunican a través de una API.
* **Hexagonal Architecture (Ports and Adapters):** Separa el núcleo de una aplicación de sus interfaces externas.
* **Clean Architecture:** Enfatiza la independencia de las capas internas de una aplicación de los marcos externos.

**¿En qué patrón de diseño o arquitectura te gustaría profundizar?** Con gusto puedo brindarte más detalles y ejemplos.

**Consideraciones adicionales:**

* **No existe un patrón único para todas las situaciones.** La elección del patrón adecuado depende de los requisitos específicos de tu aplicación.
* **Los patrones pueden combinarse.** A menudo, se utilizan múltiples patrones en una misma aplicación para resolver diferentes problemas.
* **La comprensión de los principios SOLID** (Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle) es fundamental para aplicar correctamente los patrones de diseño.