

# PATRONES DE DISEÑO

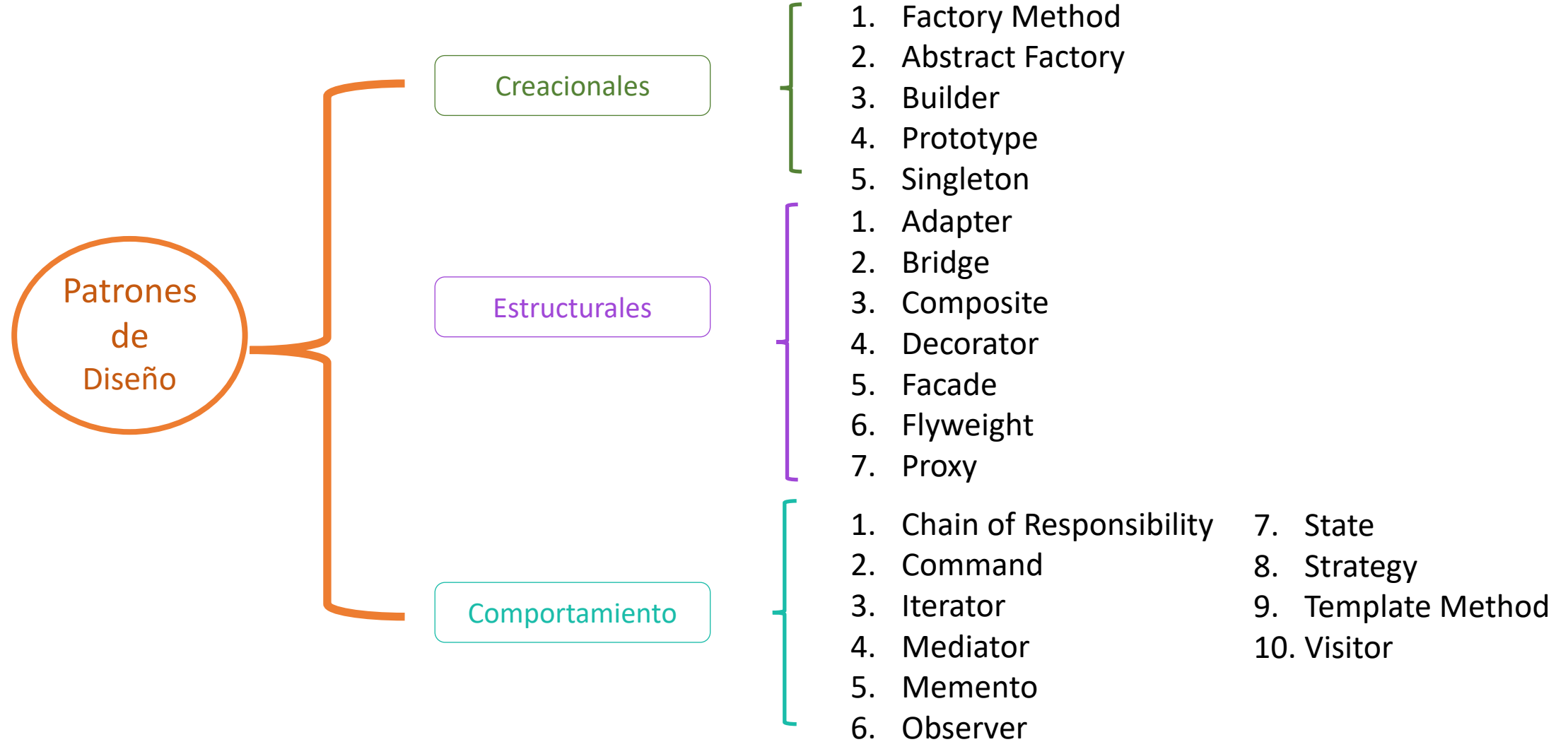
- INTRODUCCIÓN PATRONES DE DISEÑO
- CLASIFICACIÓN PATRONES DE DISEÑO
- PATRONES CREACIONALES
- PATRONES ESTRUCTURALES
- PATRONES COMPORTAMIENTO

**Camila Alejandra Cabrera Tapia**

# INTRODUCCIÓN PATRONES DE DISEÑO



# CLASIFICACIÓN PATRONES DE DISEÑO



# PATRONES CREACIONALES

Definición: Mecanismos de creación de objetos

## Clasificación:

Patrón	Factory Method		Abstract Factory		Builder		Prototype		Singleton	
Propósito	Interfaz para crear objetos en una superclase, subclasses pueden alterar		Producir familias de objetos relacionados sin especificar sus clases concretas.		Construir objetos complejos paso a paso.		Copiar objetos existentes sin que el código dependa de sus clases		Una clase tenga una única instancia	
Problema	Crear objetos sin especificar la clase exacta		Crear múltiples objetos que están interrelacionados o dependen entre sí		Objeto complejo que requiere una inicialización laboriosa		Crear una copia exacta de un objeto		Garantizar que una clase tenga una única instancia	
Solución	Clase base común, que invoque a un método <i>fábrica</i> especial		Clase base común, que declara métodos para crear cada tipo de objeto de la familia de productos		En la clase del objeto sacar métodos del constructor para crear el objeto.		Delegar el copiado al propio objeto		Hacer privado el constructor por defecto, crear un método de creación estático	
Razón	No conocer de antemano las dependencias y los tipos exactos de los objetos		Deba funcionar con varias familias de productos relacionados		Construir objetos paso a paso, utilizando tan solo los pasos necesarios		No depender de las clases concretas de objetos que se copian		Una clase tan solo deba tener una instancia	
Pros y Contras	Incorporar nuevos tipos de productos en el programa	Muchas subclasses	Certeza de compatibilidad entre objetos	Muchas interfaces y clases adicionales	Construir objetos paso a paso	Varias clases nuevas	Clonar objetos sin acoplarlos a sus clases concretas	Clonar objetos complejos con referencias circulares	Certeza de que una clase tiene una única instancia	Vulnera el Principio de responsabilidad única.

## Clasificación:

Patrón	Adapter		Bridge		Composite		Decorator		Facade	
Propósito	Colaboración entre objetos con interfaces incompatibles		Dividir una clase grande, en dos jerarquías separadas		Componer objetos en estructuras de árbol		Añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores		Proporciona una interfaz simplificada a una biblioteca	
Problema	componentes que necesitan colaborar, diseñados de manera diferente		Separar la interfaz de una clase de su implementación subyacente		Cómo manejar la composición de objetos de manera uniforme		Cómo extender la funcionalidad de un objeto sin tener que crear subclasses		Reducir la complejidad y la dependencia entre clientes	
Solución	Objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla		Extrae una de las dimensiones a una jerarquía de clases separada		Los objetos individuales y las agrupaciones de objetos deben tratarse de manera homogénea		Definir una serie de clases de "decorador" que envuelven al objeto original y agregan funcionalidad		Proporcionar una clase "fachada" que envuelve a todo el conjunto de clases o subsistemas	
Razón	Usar una clase existente, pero cuya interfaz no sea compatible con el resto del código		Dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad		Implementar una estructura de objetos con forma de árbol.		Asignar funcionalidades adicionales a objetos durante el tiempo de ejecución		Interfaz limitada pero directa a un subsistema complejo	
Pros y Contras	Separar la interfaz o el código de conversión	Introducir un grupo de nuevas interfaces y clases	Crear clases y aplicaciones independientes de plataforma	Clase muy cohesionada	Trabajar con estructuras de árbol complejas con mayor comodidad	Proporcionar una interfaz común	Extender el comportamiento de un objeto sin crear una nueva subclase	Dependa del orden en la pila de decoradores	Aislar tu código de la complejidad de un subsistema	Peligro de acoplar todas las clases de un proyecto

## Clasificación:

Patrón	Flyweight	Proxy
Propósito	Mantener más objetos dentro de la cantidad disponible de RAM	Proporcionar un sustituto o marcador de posición para otro objeto
Problema	Gran número de objetos similares necesitan ser creados	Objeto que consume gran cantidad de recursos, se necesita de vez en cuando
Solución	Dividir los objetos en dos partes: el estado intrínseco y el estado extrínseco.	Nueva clase proxy con la misma interfaz que un objeto de servicio original
Razón	El programa deba soportar una enorme cantidad de objetos	Objeto de servicio muy pesado que utiliza muchos recursos del sistema
Pros y Contras	Ahorrar uso de RAM RAM por ciclos CPU cuando deba calcularse de nuevo	Gestionar el ciclo de vida del objeto de servicio La respuesta del servicio puede retrasarse

## Clasificación:

Patrón	Chain of Responsibility		Command		Iterator		Mediator		Memento	
Propósito	Diseño de comportamiento que permite pasar solicitudes a lo largo de una cadena de manejadores		Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud		Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).		Reducir las dependencias caóticas entre objetos		Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación	
Problema	Procesar una solicitud a través de una serie de objetos, uno será capaz de manejarla.		Necesidad de encapsular una solicitud como un objeto		recorrer los elementos de una colección, independientemente de la estructura		Necesidad de gestionar las interacciones complejas entre varios objetos		Necesidad de guardar y restaurar el estado interno de un objeto	
Solución	Crear una cadena de objetos receptores, donde cada objeto en la cadena tiene un enlace a su sucesor.		Crear una clase para cada comando que encapsule la solicitud como un objeto.		Definir una interfaz común para recorrer los elementos de una colección, llamada iterador.		Introducir un objeto mediador que coordine las interacciones entre los objetos del sistema		Definir tres objetos principales: el Originator (Origen), el Memento (Recuerdo) y el Caretaker (Cuidador).	
Razón	Parametrizar objetos con operaciones.		Parametrizar objetos con operaciones		Colección que tenga una estructura de datos compleja a nivel interno,		Difícil cambiar algunas de las clases porque están fuertemente acopladas		Producir instantáneas del estado del objeto para poder restaurar un estado previo del objeto.	
Pros y Contras	Desacoplar las clases que invocan operaciones de las que realizan	Algunas solicitudes pueden acabar sin ser gestionadas.	Desacoplar las clases que invocan operaciones	Nueva capa entre emisores y receptores.	extraer algoritmos de recorrido y colocarlos en clases independientes	Proporcionar una interfaz común	Reduce la dependencia directa entre los objetos del sistema	Centraliza la lógica de comunicación y control en un único objeto	Producir instantáneas del estado del objeto	Consumir mucha memoria RAM

# PATRONES DE COMPORTAMIENTO

Definición: Tratan con algoritmos y la asignación de responsabilidades entre objetos

## Clasificación:

Patrón	Observer		State		Strategy		Template Method		Visitor	
Propósito	Definir un mecanismo de suscripción para notificar a varios objetos		Permite a un objeto alterar su comportamiento cuando su estado interno cambia		Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada		Añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores		Permite separar algoritmos de los objetos sobre los que operan	
Problema	Asegurar que múltiples objetos sean notificados y actualizados cuando otro objeto cambia de estado		Implementar de manera eficiente un comportamiento que varía según el estado de un objeto		cómo encapsular algoritmos en objetos separados, el cliente pueda seleccionar		Cómo definir un algoritmo general en una clase base		cómo extender la funcionalidad de una estructura de objetos sin alterar su diseño	
Solución	definir una relación uno a muchos entre un objeto sujeto a cambios y uno o más objetos que dependen de él		Modelar cada estado posible del objeto como una clase separada que implementa una interfaz común		Definir una familia de algoritmos encapsulados en clases separadas		Definir un método plantilla en la clase base que contiene la estructura general del algoritmo		Definir una interfaz Visitor que contiene métodos para visitar cada tipo de objeto en la estructura.	
Razón	Los cambios en el estado de un objeto puedan necesitar cambiar otros objetos		Un objeto que se comporta de forma diferente dependiendo de su estado actual,		Utiliza distintas variantes de un algoritmo dentro de un objeto		Permitir que extiendan únicamente pasos particulares de un algoritmo		Necesidad de realizar una operación sobre todos los elementos de una compleja estructura de objetos	
Pros y Contras	Introducir nuevas clases suscriptoras	Los suscriptores son notificados en un orden aleatorio	Organiza el código relacionado con estados particulares	Excesivo si una máquina de estados sólo tiene unos pocos estados	Intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución	El programa en exceso con nuevas clases	Sobrescribir tan solo ciertas partes de un algoritmo grande	Limitados por el esqueleto proporcionado	Introducir un nuevo comportamiento o que puede funcionar	Actualizar todos los visitantes cada vez que una clase se añade



# OPINIÓN PERSONAL

## OPINIÓN 1:

En mi experiencia he podido únicamente trabajar y entender por completo el patrón de diseño creacional “Singleton”. Esto dado que tenía la necesidad de asegurarme de crear solo una instancia de la clase conexión base de datos. Por tanto, considero que el aprendizaje completo de un patrón se da cuando uno interactúa con un entorno real. Además, el uso de un patrón se da siempre fruto de una necesidad, y no creo que sea correcto solo usar patrones sin verdaderamente requerirlos. En mi opinión, un patrón por definición es una solución a un problema no una decoración al código.

## OPINIÓN 2:

En la investigación observe que en muchas ocasiones la desventaja de usar un patrón de diseño es que puede aumentar la complejidad del código porque se aumenta el número de clases y las relaciones. Por ende, es importante determinar buenas prácticas de código limpio al momento de implementar un patrón.

## OPINIÓN 3:

Los patrones que más me llamaron la atención y que desconocía completamente de sus existencias fueron Flyweight y Proxy. Flyweight me llamó la atención por el análisis muy certero que se debería hacer para determinar las propiedades intrínsecas y extrínsecas, considero que debe ser alguien que esté muy claro en el código para poder aplicarlo sin complicaciones. Por otro lado, el patrón Proxy me pareció interesante por el nivel de detalle que propone al definir como serían los accesos indirectos y funcionalidades respecto al objeto, lo que sugiere que se debe tener un buen conocimiento del flujo del Sistema. En general observe la importancia de conocer los aspectos del código para poder implementar cualquier tipo de patrón y no complicar el entendimiento de este.

## OPINIÓN 4:

Me llamó la atención los patrones de comportamiento referentes al cambio de estado y notificaciones a los demás objetos, “Observer” y “State”. Esto porque actualmente hay muchas tecnologías que se enfocan en los notificadores de estado como es el caso del framework flutter e inclusive en .net core si se aplica el modelo mvvm se puede acceder a la biblioteca que controla los cambios de estado. Esto me pareció curioso y muy oportuno de aprender porque es interesante ver cómo se puede correlacionar cada arista del conocimiento para hacer más preciso y amplio.



# FUENTE

---

- Refactoring.Guru. (2024). *Refactoring.Guru*. Obtenido de Patrones de diseño: <https://refactoring.guru/es/design-patterns>