PATRONES DE DISEÑO

OBJETIVOS

- Conocer cuáles son los patrones de diseño estructurales.
- Identificar cuándo aplicar cada uno de los patrones de diseño estructurales.
- Conocer cuáles son los patrones de diseño de comportamiento.
- Identificar cuándo aplicar cada uno de los patrones de diseño de comportamiento.

1

REPASO

Preguntas sobre la última clase ...

¿QUÉ ES UN PATRÓN?

En el contexto de diseño de software ...

¿QUÉ SON LOS PATRONES DE DISEÑO?

- Descripciones de objetos y clases que se comunican (relaciones) que han sido personalizados para resolver un problema de diseño general en un contexto particular.
- 'Alias' microarquitectura.
- Ayudan a los programadores en la optimización de su tiempo.

¿CÓMO SE CLASIFICAN SEGÚN SU PROPÓSITO?

Clasificación de los Patrones de Diseño.

DESCRIPCIÓN SEGÚN PROPÓSITO

Creacionales

- Creación de objetos.
- Flexibilizar el código.
- Reutilizar el código.

Estructurales

- Organizan clases y objetos en estructuras.
- Estructuras flexibles.
- Estructuras eficientes.

De comportamiento

- Relacionado a algoritmos.
- Responsabilidades entre objetos.

CLASIFICACIÓN DE LOS PATRONES DE DISEÑO

		Propósito		
		Creacional	**Estructural**	**De comportamiento**
Alcance	**Clase**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	**Adapter** Bridge Composite **Decorator** **Facade** **Proxy**	**Chain of Responsibility** **Command** **Iterator** **Mediator** Memento Flyweight **Observer** **State** Strategy Visitor

2

PATRONES DE DISEÑO ESTRUCTURALES

Detallemos algunos de los patrones ...

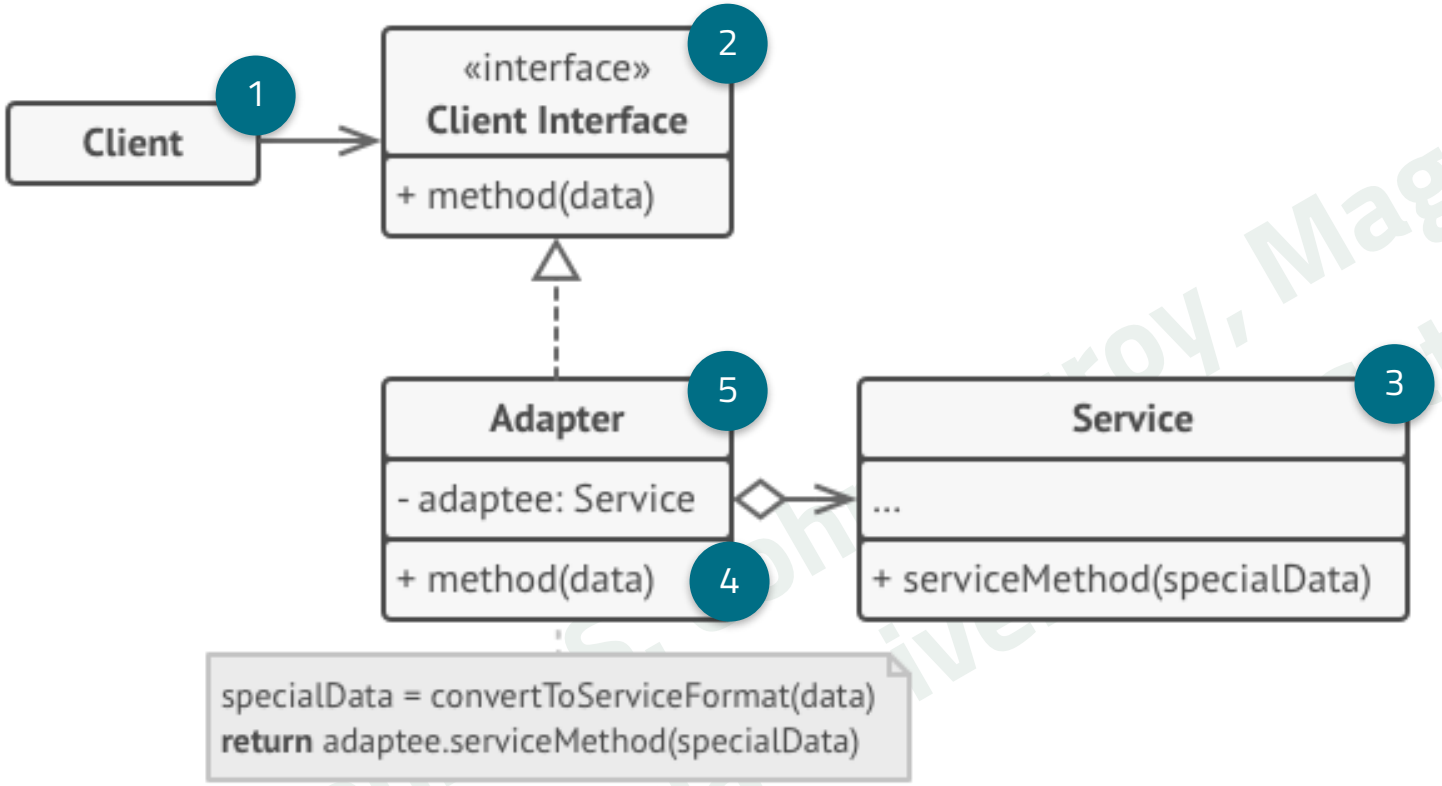
PATRÓN: ADAPTER

PROBLEMA QUE CUBRE

- Utilizar una clase que no es compatible con el resto del código.

¿CÓMO LO HACE?

Permite que dos objetos con interfaces incompatibles puedan comunicarse a través de un “Adaptador”.



1. Cliente tiene cierta lógica de negocio pre-existente.
2. La interfaz describe los protocolos que otras clases deben de seguir.
3. Clase (posiblemente externa) que no es compatible con la interfaz.
4. Adaptador entre el Cliente (implementa su interfaz) y el Servicio (al cual referencia).
5. La clase Adaptadora puede ser reemplazada de forma transparente al Cliente.

CONSECUENCIAS: ADAPTER

- (+) Separa la interfaz o el código de conversión de datos de la lógica de negocio del programa.
- (+) Puedes introducir nuevos tipos de adaptadores al programa de forma transparente para el Cliente.
- (-) Mayor complejidad debido a un grupo de nuevas interfaces y clases.

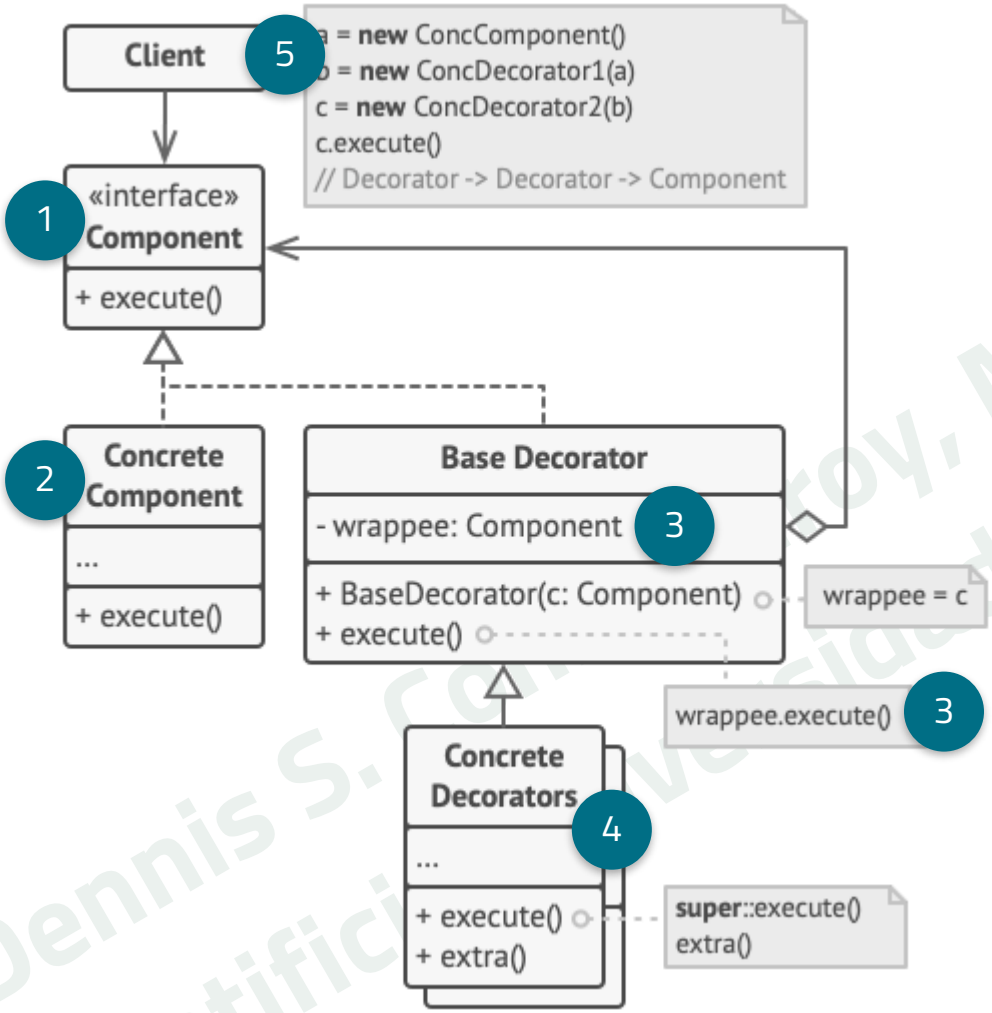
PATRÓN: DECORATOR

PROBLEMA QUE CUBRE

- Agregar funcionalidades a algunos objetos, no a toda la clase.
- Remover funcionalidades a algunos objetos.

¿CÓMO LO HACE?

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores (wrappers) que contienen estas funcionalidades.



1. Interfaz común para objetos encapsulados y que encapsularán.
2. Define el comportamiento básico que los decoradores extenderán.
3. Encapsula al objeto concreto. Delega operaciones al objeto concreto.
4. Definen funcionalidades adicionales y/o sobrescriben funciones del decorador base.
5. El Cliente crea al objeto y lo asocia a los decoradores.

CONSECUENCIAS: DECORATOR

- (+) Permite extender el comportamiento de un objeto sin crear una nueva subclase.
- (+) Permite añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- (+) Permite dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas
- (-) Complejo remover un decorador de una pila de decoradores.
- (-) Comportamiento del decorador depende de su orden en la pila.

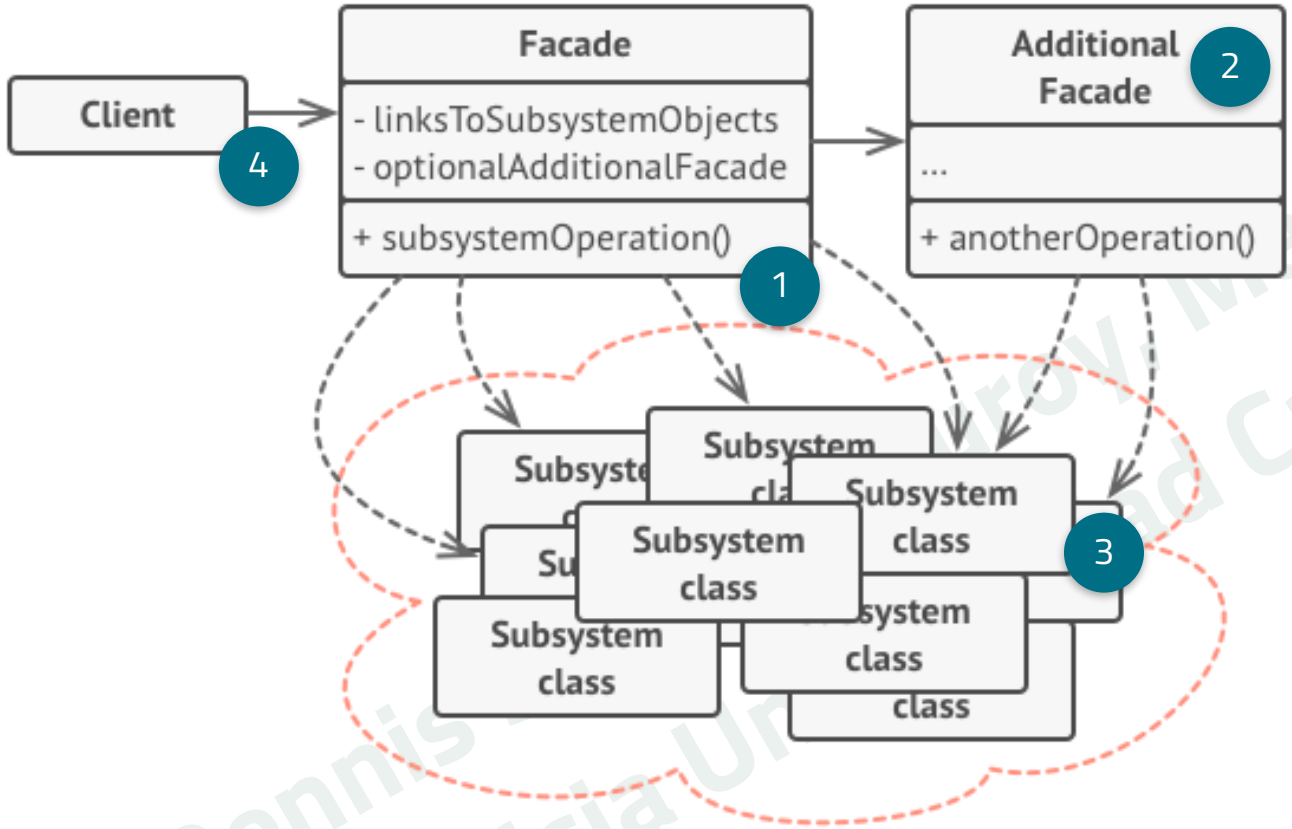
PATRÓN: FACADE

PROBLEMA QUE CUBRE

- Contar con una interfaz para un subsistema complejo.
- Estructurar un subsistema en capas.

¿CÓMO LO HACE?

Provee una interfaz simplificada a un conjunto de librerías, a un framework y/o a un conjunto complejo de clases.



1. Provee acceso a una parte particular del subsistema.

2. Elemento opcional para separar la complejidad de la lógica de la Fachada.

3. Objeto del subsistema que se comunican entre ellos; pero desconocen la existencia de una Fachada.

4. Cliente utiliza la Fachada en lugar de invocar directamente a los objetos del subsistema.

CONSECUENCIAS: FACADE

- (+) Permite aislar el código Cliente de la complejidad de un subsistema.
- (-) Puede convertirse en un gran objeto que acople a todas las clases de la aplicación.

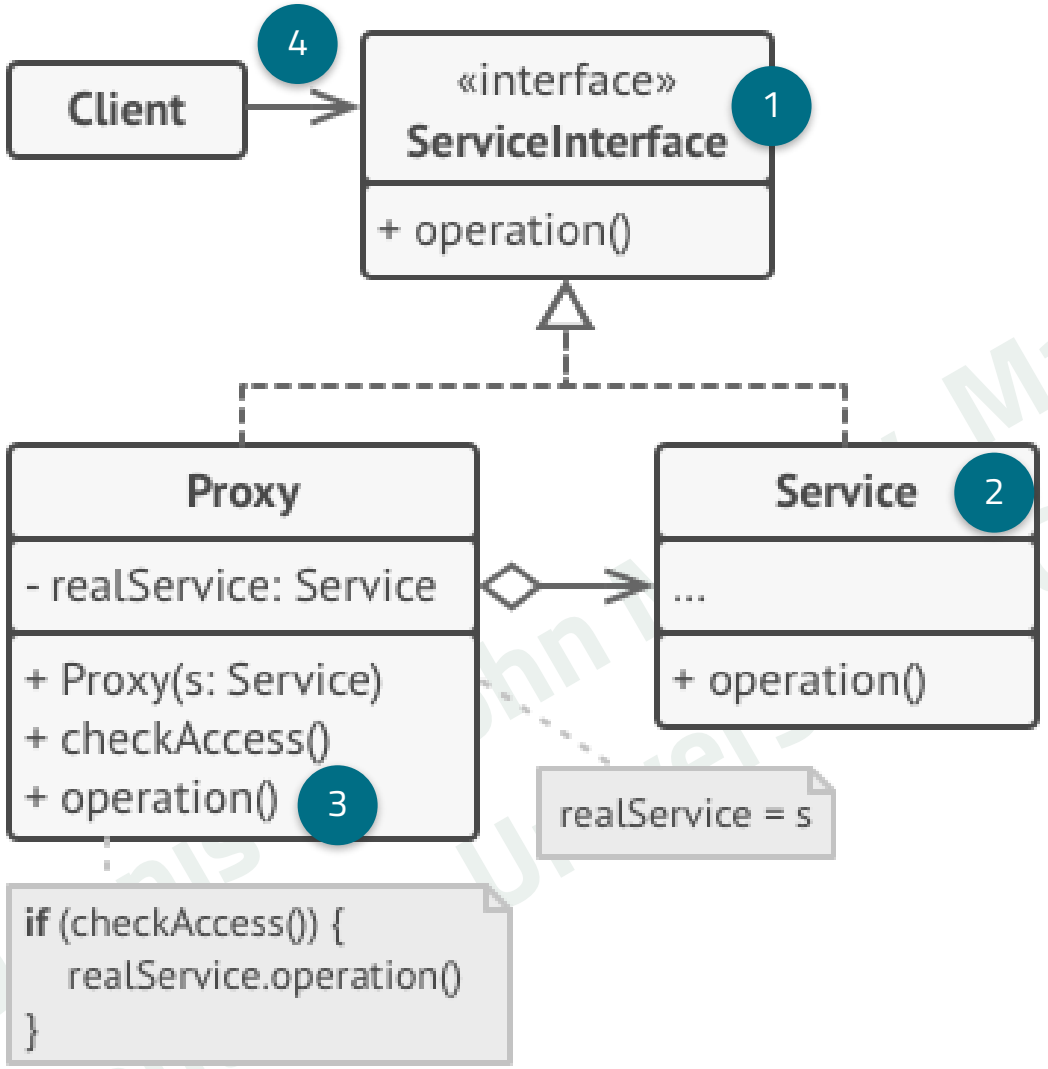
PATRÓN: PROXY

PROBLEMA QUE CUBRE

- Creación de objetos “pesados”.
- Control de acceso.
- Gestionar logs.
- Manejo de caché.

¿CÓMO LO HACE?

Controla el acceso al objeto original. Ejecuta código antes o después de invocar a los métodos del objeto original.



1. El proxy debe seguir esta interfaz para poder camuflarse como objeto de servicio.
2. Lógica de negocio a “Ocultar”.
3. Cuando finaliza su procesamiento (inicialización diferida, registro, gestión de acceso y de caché), pasa la solicitud al servicio.
4. El Cliente debe de trabajar con la interfaz para poder reemplazar el Servicio con un Proxy.

CONSECUENCIAS: PROXY

- (+) Permite controlar el objeto de servicio sin que los clientes lo sepan.
- (+) Funciona si el objeto de servicio no está listo o no está disponible.
- (+) Permite introducir nuevos proxies sin cambiar el servicio o los clientes.
- (-) Mayor complejidad por el mayor número de clases.
- (-) Puede afectar los tiempos de respuesta del servicio.

3

PATRONES DE DISEÑO DE COMPORTAMIENTO

Detallemos algunos de los patrones ...

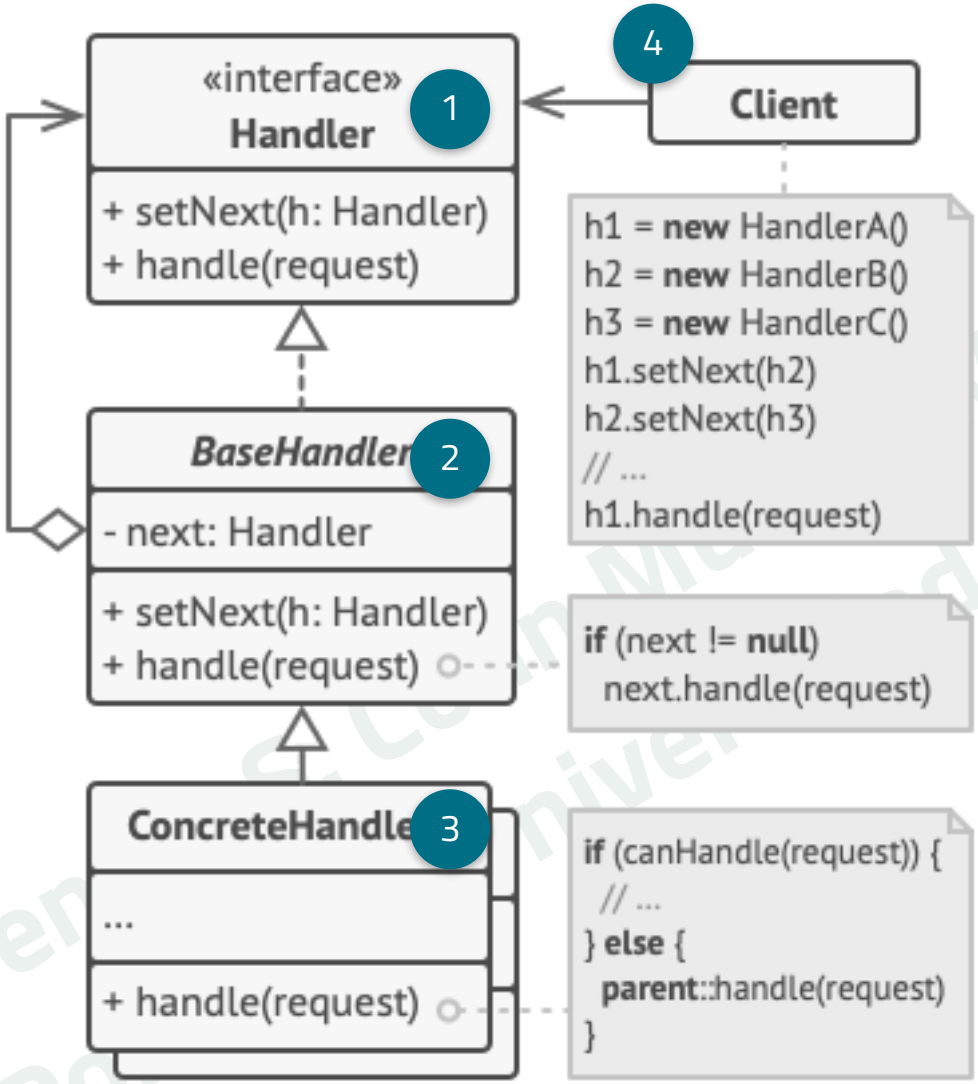
PATRÓN: CHAIN OF RESPONSIBILITY

PROBLEMA QUE CUBRE

- Más de un objeto (no conocido a priori) puede gestionar una solicitud.
  - No se desea especificar al manejador de forma explícita.
  - El listado de manejadores debe ser dinámico.

¿CÓMO LO HACE?

Permite pasar solicitudes a lo largo de una cadena de manejadores. Cada manejador decide si procesa la solicitud o si la pasa al siguiente manejador de la cadena.



1. Interfaz para los Manejadores. Tiene un método para atender la solicitud y otro para remitirla a otro manejador.
2. Manejador Base Abstracto. Incluye la referencia al siguiente manejador.
3. Manejador Concreto contiene la lógica para atender la solicitud (decide si la atiende y/o la pasa al siguiente manejador).
4. Cliente arma la cadena de manejadores (estática o dinámicamente) y le envía la solicitud.

CONSECUENCIAS: CHAIN OF RESPONSIBILITY

- (+) Permite controlar el orden de control de solicitudes.
- (+) Permite introducir nuevos manejadores en la aplicación sin descomponer el código cliente existente.
- (-) Impacto en el tiempo de ejecución y recursos de procesamiento; al tenerse objetos instanciado que solo reenviarán la solicitud.

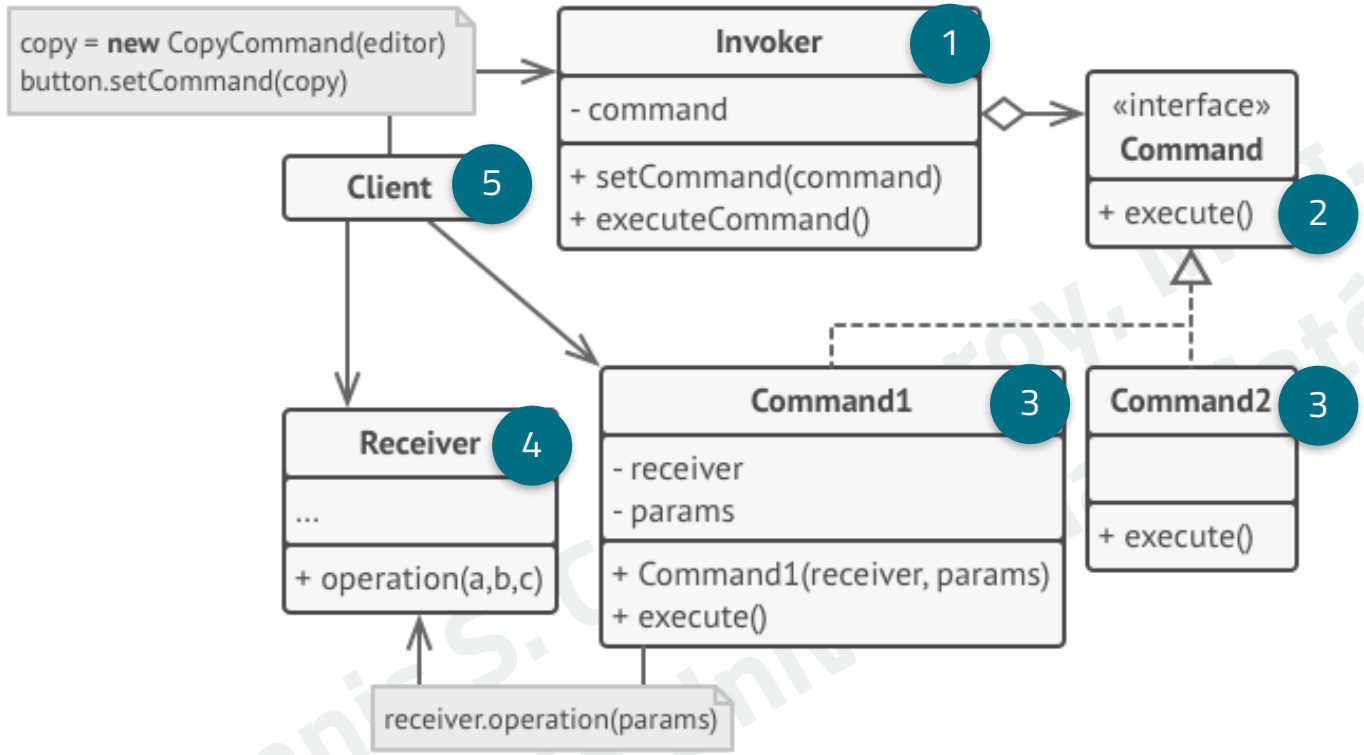
PATRÓN: COMMAND

PROBLEMA QUE CUBRE

- Encolar operaciones o ejecutarlas de forma remota.
- Parametrizar objetos con operaciones.
- Soportar deshacer.
- Contar con datos de trazabilidad de una operación.

¿CÓMO LO HACE?

Transforma solicitudes (requests) en objetos que contienen toda la información de la solicitud.



1. Invocador inicializa las solicitudes. Almacena una referencia a un objeto comanda. Llama a este objeto en lugar de enviar la solicitud al receptor.

2. Normalmente solo cuenta con un método execute().

3. Los parámetros son atributos del objeto. Arma la llamada para la lógica de negocio.

4. Objeto que recibe el comando. Ejecuta el trabajo.

5. Crea y configura el objeto Comanda (parámetros y Receiver). Asocia la Comanda con el Invocador.

CONSECUENCIAS: COMMAND

- (+) Desacopla las clases que invocan operaciones de las que realizan esas operaciones.
- (+) Permite introducir nuevos comandos en la aplicación sin descomponer el código cliente existente.
- (+) Permite implementar deshacer/rehacer
- (-) Mayor complejidad en el código; se tiene una nueva capa (el invocador) entre el emisor y el receptor.

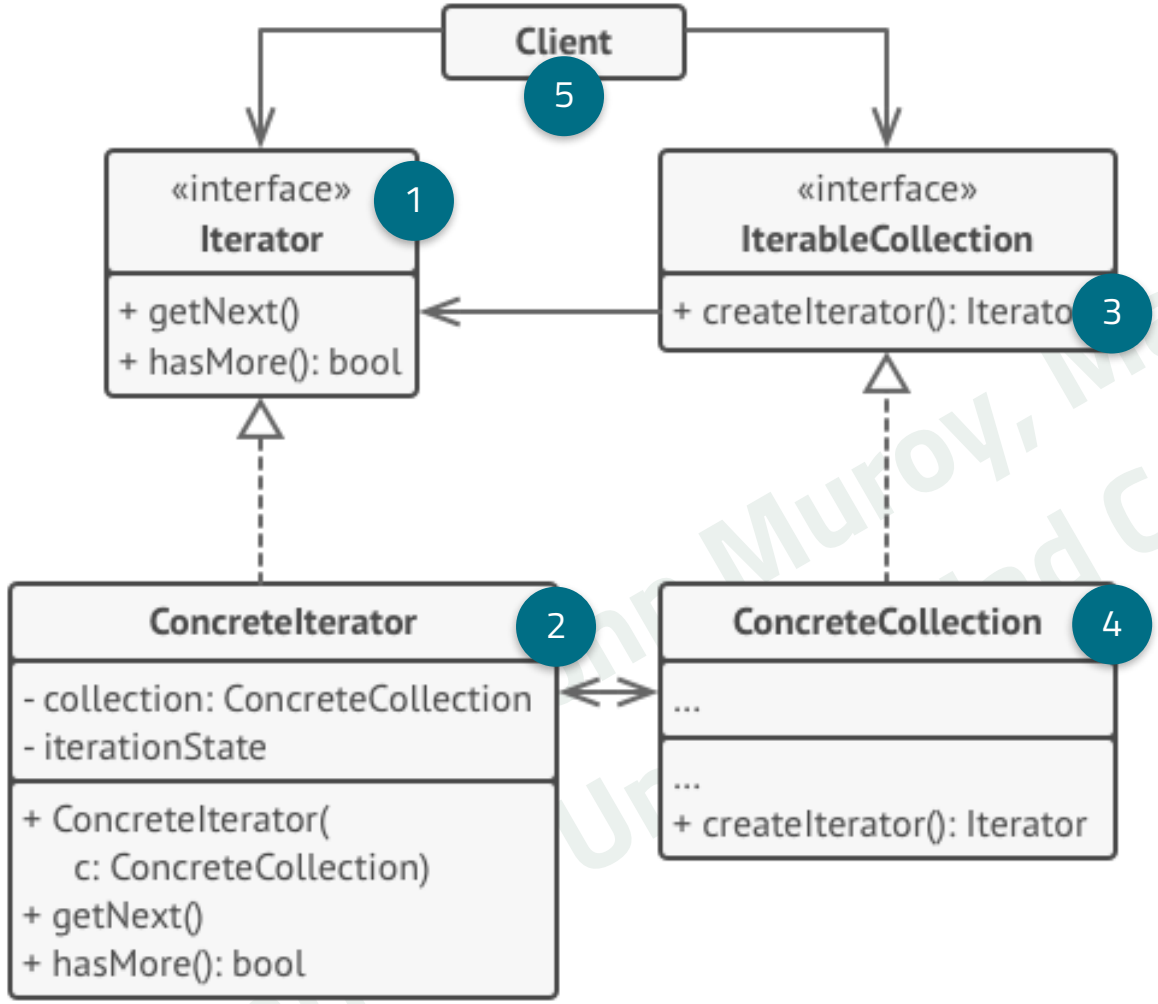
PATRÓN: ITERATOR

PROBLEMA QUE CUBRE

- Mantener una estructura compleja (y privada por conveniencia o seguridad).
- Reduce duplicación al implementar iteradores.

¿CÓMO LO HACE?

Permite recorrer los elementos de una colección sin exponer su estructura (lista, árbol, pila, etc.).



1. Declara los métodos para recorrer una la colección.
2. Implementa los métodos para recorrer la colección.
3. Declara el método para obtener un iterador compatible con la colección.
4. Implementa la colección y el método para obtener un iterador compatible.
5. Trabaja con el iterador y la colección a través de sus interfaces a fin de no generar acoplamiento.

CONSECUENCIAS: ITERATOR

- (+) Las estructuras y sus iteradores tienen clases dedicadas.
- (+) Permite reemplazar las colecciones e iteradores.
- (-) No se recomienda su uso en caso se trabaje con colecciones sencillas.

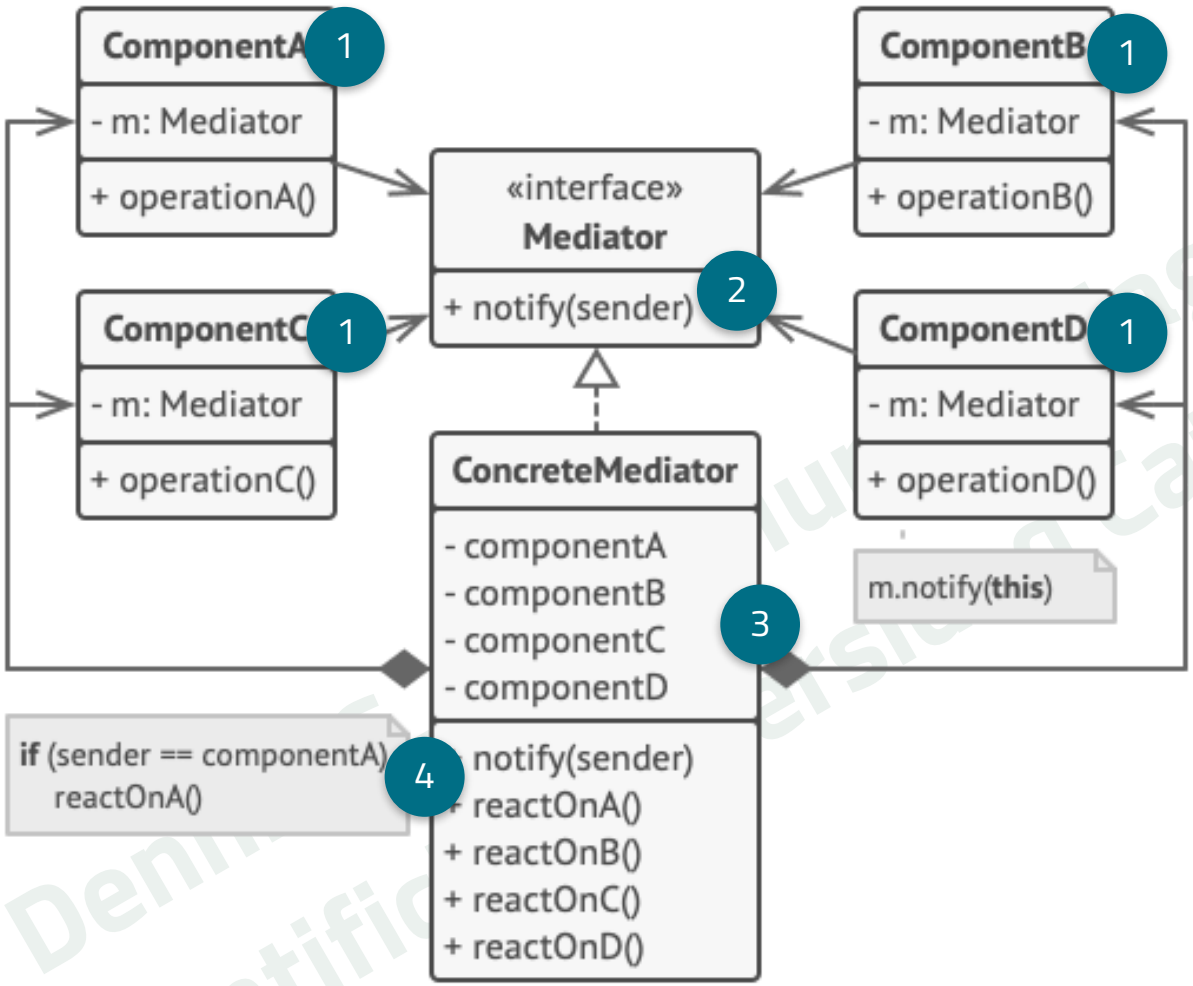
PATRÓN: MEDIATOR

PROBLEMA QUE CUBRE

- Un conjunto de objetos se comunica de forma compleja; ello impacta en la legibilidad de la solución.
- La reutilización de objetos es compleja dado su fuerte acoplamiento.

¿CÓMO LO HACE?

Reduce las dependencias caóticas entre objetos. Restringe las comunicaciones directas entre los objetos al utilizar un objeto mediador.



1. Componentes que contienen lógica de negocio. Referencia a la interfaz mediadora.

2. Interfaz que declara métodos de comunicación con componentes. Normalmente incluyen un único método de notificación.

3. Mediadores Concretos encapsulan las relaciones entre varios componentes.

4. Los componentes no deben conocer otros componentes. Cada componente sólo debe notificar a la interfaz mediadora. Cuando la mediadora recibe la notificación, debe poder identificar fácilmente al emisor.

CONSECUENCIAS: MEDIATOR

- (+) Permite extraer las comunicaciones entre varios componentes dentro de un único sitio; facilitando el mantenimiento.
- (+) Permite introducir nuevos mediadores sin tener que cambiar los propios componentes.
- (-) Un mediador podría evolucionar en un objeto complejo y crítico.

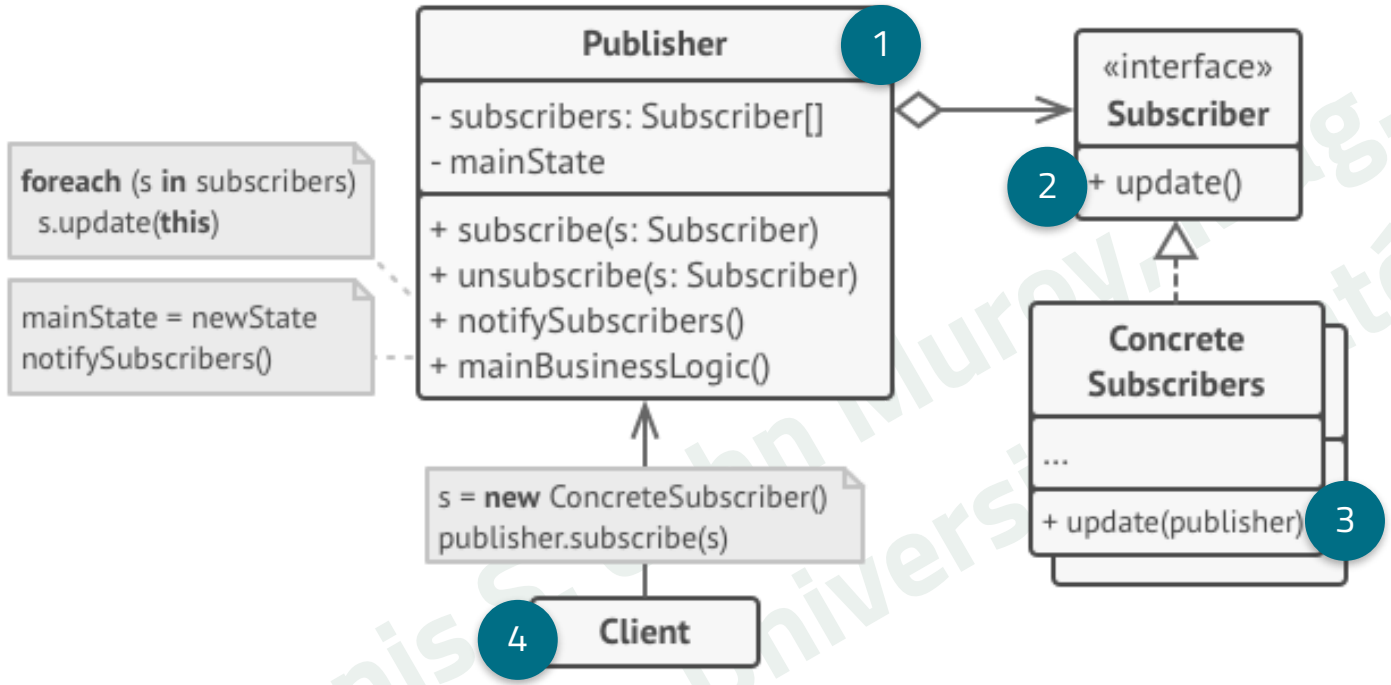
PATRÓN: OBSERVER

PROBLEMA QUE CUBRE

- Cuando un cambio de estado en un objeto debe de ser notificado a otros objetos (desconocidos o dinámicos).
- Cuando un objeto requiere observar a otro bajo condiciones determinadas.

¿CÓMO LO HACE?

Define un mecanismo de subscripción a un objeto. Cualquier cambio de estado sobre el objeto es remitido a los objetos subscritos.



1. Envía eventos de interés a otros objetos. Permite a los objetos subscribirse.
2. El publicador invoca el método de notificación de cada subscritor.
3. Los subscriptores efectúan alguna acción en base a la información recibida desde el publicador.
4. Durante la inicialización, crea los objetos Suscriptores y el objeto Publicador; y los asocia.

CONSECUENCIAS: OBSERVER

- (+) Permite introducir nuevas clases subscriptoras sin tener que cambiar el código de la publicadora.
- (-) La recepción de las notificaciones por parte de los suscriptores es asíncrono. No hay un control en el orden en que estas notificaciones son recibidas por cada objeto subscriptor.

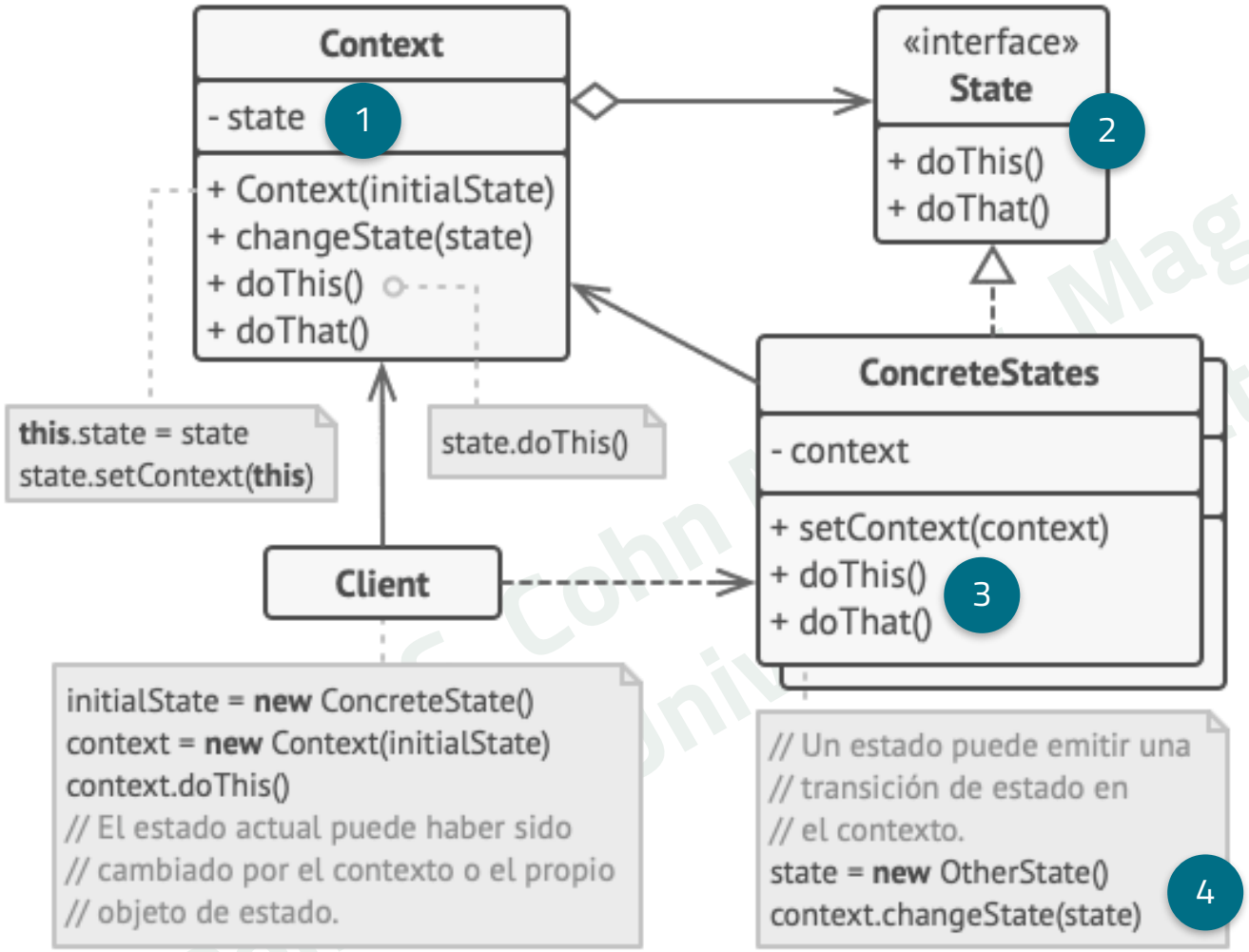
PATRÓN: STATE

PROBLEMA QUE CUBRE

- El comportamiento de un objeto debe depender de su estado.
- Estructuras condicionales complejas relacionadas al estado del objeto.

¿CÓMO LO HACE?

Permite a un objeto alterar su comportamiento cuando su estado interno cambia.



1. Clase cuyo comportamiento cambia según su estado. Almacena una referencia a un objeto de estado y le delega el trabajo específico del estado
2. La interfaz Estado declara los métodos comunes para todos los estados.
3. Implementa los métodos específicos del estado. Pueden almacenar una referencia al Contexto para extraer datos requeridos.
4. Pueden efectuar un cambio de estado del Contexto.

CONSECUENCIAS: STATE

- (+) Organiza el código relacionado a los estados en clases separadas.
- (+) Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.
- (+) Elimina estructuras condicionales complejas.
- (-) No se recomienda su uso si se manejan pocos estados o si estos cambian con poca frecuencia.

4

REFERENCIAS

BIBLIOGRAFÍA

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH.
- Freeman, E., & Robson, E. (2020). Head First Design Patterns. O'Reilly Media.
- Design patterns and refactoring. (n.d.). https://sourcemaking.com/design_patterns

Créditos:

- Plantilla de la presentación por SlidesCarnival
- Fotografías por Unsplash
- Diseño del fondo Hero Patterns