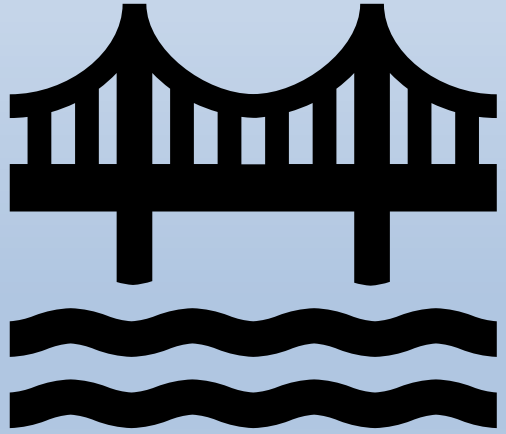
Patrones de diseño

Patrones Estructurales



Bloque 1 – Patrones de diseño
2º DAW – Desarrollo Web en Entorno Servidor
Fernando Díaz-Alonso Dorado

Patrones estructurales

¿Qué son?

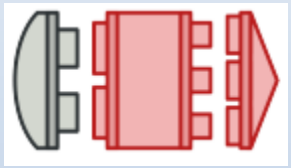
Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

Patrones estructurales

Principales patrones estructurales

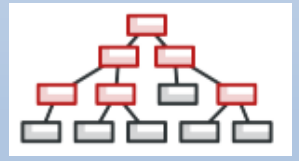
Algunos patrones creacionales son:

- Adapter

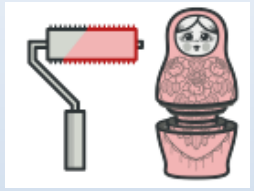


- Bridge

- Composite

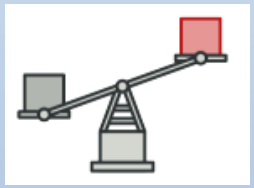


- Decorator

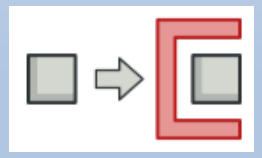


- Facade

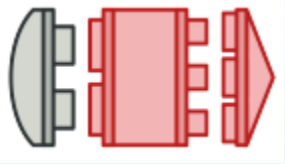
- Flyweight



- Proxy



Adapter



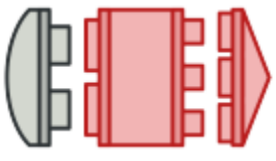
Adaptador, envoltorio, wrapper

Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Aplicabilidad:

- Cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

Adapter



Adaptador, envoltorio, wrapper II

Pros y contras:

- ✓ *Principio de responsabilidad única*. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.
- ✗ La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Bridge

Puente

Es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

Aplicabilidad:

- Cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).
- Cuando o necesites extender una clase en varias dimensiones ortogonales (independientes)
- Cuando necesites poder cambiar implementaciones durante el tiempo de ejecución.

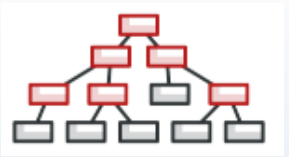
Bridge

Puente II

Pros y contras:

- ✓ Puedes crear clases y aplicaciones independientes de plataforma.
- ✓ El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
- ✓ *Principio de responsabilidad única*. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
- × Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.

Composite



Objeto compuesto, Object Tree

Es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Aplicabilidad:

- Cuando tengas que implementar una estructura de objetos con forma de árbol.
- Cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.

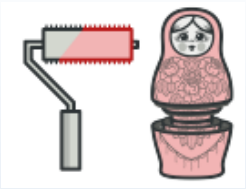
Composite

Objeto compuesto, Object Tree II

Pros y contras:

- ✓ Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.
- ✓ Principio de abierto/cerrado. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.
- ✗ Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Decorator



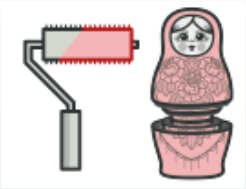
Decorador, Envoltorio, Wrapper

Es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

Aplicabilidad:

- Cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
- Cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

Decorator

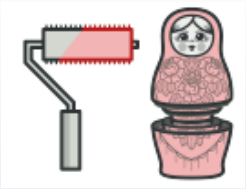


Decorador, Envoltorio, Wrapper II

Pros:

- ✓ Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- ✓ Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- ✓ Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- ✓ Principio de responsabilidad única. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas

Decorator



Decorador, Envoltorio, Wrapper III

Contras:

- Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- El código de configuración inicial de las capas pueden tener un aspecto desagradable

Facade

Fachada

Es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Aplicabilidad:

- Cuando necesites una interfaz limitada pero directa a un subsistema complejo.
- Cuando quieras estructurar un subsistema en capas.

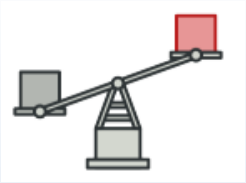
Facade

Fachada II

Pros y contras:

- ✓ Puedes aislar tu código de la complejidad de un subsistema.
- ✗ Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.

Flyweight



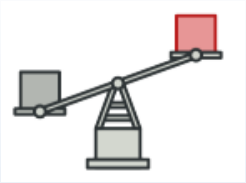
Peso mosca, peso ligero, Cache

Es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

Aplicabilidad:

- Únicamente cuando tu programa deba soportar una enorme cantidad de objetos que apenas quepan en la RAM disponible.

Flyweight

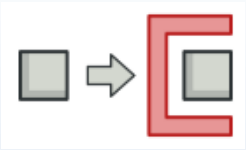


Peso mosca, peso ligero, Cache II

Pros y contras:

- ✓ Puedes ahorrar mucha RAM, siempre que tu programa tenga toneladas de objetos similares.
- ✗ Puede que estés cambiando RAM por ciclos CPU cuando deba calcularse de nuevo parte de la información de contexto cada vez que alguien invoque un método flyweight.

Proxy

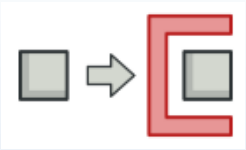


Es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

Aplicabilidad:

Existen decenas de formas de utilizar este patrón, vamos a mencionar los más conocidos.

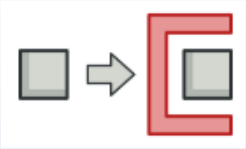
Proxy



Aplicabilidad

- ***Inicialización diferida (proxy virtual*).** Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando
- ***Control de acceso (proxy de protección).*** Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio, por ejemplo, cuando tus objetos son partes fundamentales de un sistema operativo y los clientes son varias aplicaciones lanzadas (incluyendo maliciosas)
- ***Ejecución local de un servicio remoto (proxy remoto)***. Es cuando el objeto de servicio se ubica en un servidor remoto.

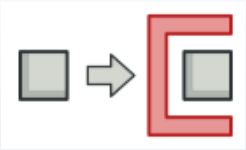
Proxy



Aplicabilidad II

- ***Solicitudes de registro (proxy de registro).*** Es cuando quieres mantener un historial de solicitudes al objeto de servicio.
- ***Resultados de solicitudes en caché (proxy de caché).*** Es cuando necesitas guardar en caché resultados de solicitudes de clientes y gestionar el ciclo de vida de ese caché, especialmente si los resultados son muchos.
- ***Referencia inteligente.*** Es cuando debes ser capaz de desechar un objeto pesado una vez que no haya clientes que lo utilicen

Proxy



Pros y contras:

- ✓ Puedes controlar el objeto de servicio sin que los clientes lo sepan.
- ✓ Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.
- ✓ El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.
- ✗ El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.
- ✗ La respuesta del servicio puede retrasarse.

Principios del diseño de software

Bibliografía

- Patrones de Diseño – Alexander Shvets