TRABAJO AUTÓNOMO

1. Datos Generales

Carrera:	**Tecnología Superior en Desarrollo de Software**
Período académico:	**Noviembre 2022 – Marzo 2023**
Asignatura:	**Tendencias Actuales de Programación**
Unidad Nº:	**1**
Tema:	**Patrones de Diseño**
Ciclo-Paralelo:	**M5A**
Estudiante:	**Daniel Andrade – Alejandro Coraspe – Bryan Curillo**
Docente:	**Ing. Diego Cale Mgtr.**

2. Contenido

2.1 Introducción

Los patrones de diseño son soluciones probadas y reconocidas para problemas comunes en el diseño de software. Estas soluciones se basan en la experiencia acumulada de expertos en diseño y permiten desarrollar software más eficiente, mantenible y reutilizable.

Los patrones de diseño proporcionan pautas y plantillas que ayudan a los desarrolladores a abordar problemas comunes de manera sistemática y estructurada. Cada patrón describe un problema específico y propone una solución generalmente aplicable, utilizando un lenguaje común y comprensible.

Existen diferentes categorías de patrones de diseño, como patrones creacionales, estructurales y de comportamiento. Los patrones creacionales se enfocan en la creación de objetos, los estructurales en la composición de clases y objetos, y los de comportamiento en las interacciones entre ellos.

Al utilizar patrones de diseño, los desarrolladores pueden aprovechar soluciones probadas y evitar reinventar la rueda. Esto conduce a un código más limpio, modular y flexible, que se puede adaptar más fácilmente a los cambios futuros.

2.2 Objetivo

- Conocer los patrones de diseño en java, más concretamente los patrones estructurales.
- Comprender como actúan estos patrones de diseños y como estos se relacionan entre sí.
- Plantear casos de la vida real donde se evidencien estos patrones.
- Analizar los pros y contras que tiene cada uno de estos patrones para facilitar la elección de estos en futuros protectos.
- Elaborar ejercicios con estos patrones para una mejor comprensión.

2.3 Materiales, herramientas, equipos y software

- Internet.
- Computadora.
- Word.

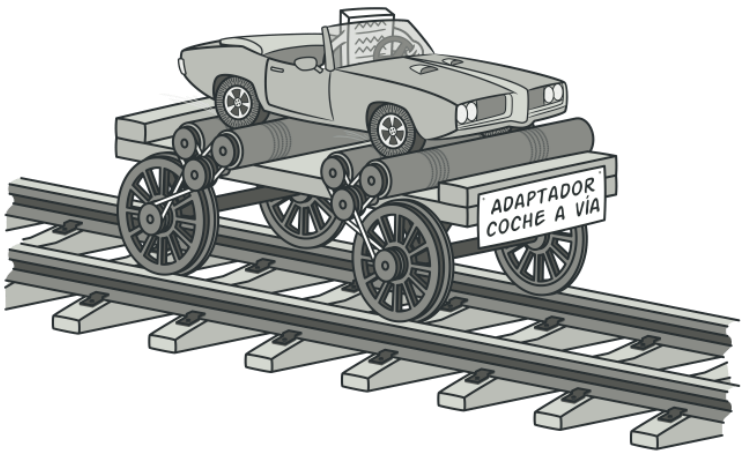
2.4 Desarrollo

ADAPTER

Propósito.

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

El adaptador se encarga de traducir las solicitudes de uno de los objetos en términos que el otro objeto pueda entender.



Problema

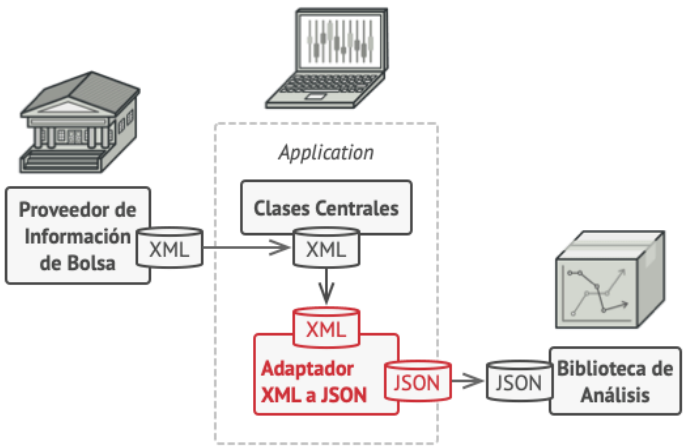
Imagina que estás creando una aplicación. La aplicación descarga la información en formato XML para presentarla al usuario.

En cierto momento, decides mejorar la aplicación integrando una biblioteca de análisis, pero lla biblioteca de análisis solo funciona con datos en formato JSON.

Solución

Se trata de un objeto que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

El patrón de diseño del Adaptador implica que un objeto envuelva a otro con el fin de ocultar la complejidad de la conversión.



Analogía en el mundo real

Supongamos que viajas de Ecuador a Estados Unidos, seguramente no podrás cargar ni tu computadora portátil o tu celular ya que los tipos de enchufe son diferentes en cada país, por lo que un enchufe común para nosotros no sirve en Estados Unidos. El

problema puede solucionarse utilizando un adaptador que incluya el enchufe de los dos tipos.

Aplicabilidad

Usa la clase adaptadora cuando quieras usar una clase existente que no sea compatible con otro código.

Usa el patrón cuando quieras reutilizar subclases existentes que no tenga alguna función que no pueda añadirse

Cómo implementarlo

1. Primero que nada, debes tener en cuenta tener dos o más clases incompatibles
2. Declara la interfaz y describe el modo en que la primera clase se comunican con la segunda.
3. Crea la clase adaptadora y haz que siga la interfaz con la primera clase.
4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto. Lo común es inicializar este campo a través del constructor
5. Uno por uno, implementa todos los métodos de la interfaz primaria en la clase adaptadora
6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente.

Pros y contras

- **Ventajas**

Separar el código de la lógica principal del programa implica mantener una distinción entre cómo los datos se manejan y se presentan

Puedes introducir nuevos tipos de adaptadores al programa sin descomponer

- **Desventajas**

La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases.

Relaciones con otros patrones

- Bridge suele diseñarse por anticipado, pero, Adapter se utiliza habitualmente con una aplicación existente.
- Adapter cambia una interfaz de un objeto, decorator mejora la misma.
- Facade, crea interfaces por objetos existentes, adaptar hace que la misma interfaz sea usable.
- Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras similares ya que, todos se basan en delegar trabajo a otros objetos.

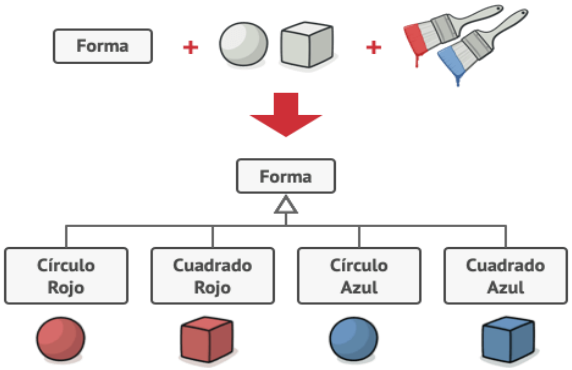
BRIDGE

Propósito

Bridge es un patrón de diseño que te permite dividir una clase grande, o un grupo de clases relacionadas, en dos separadas (abs e imp) que pueden desarrollarse independientemente la una de la otra, pero no pierden la relación.

Problema

Supongamos que tienes una clase llamada "Forma" en el ámbito de la geometría, la cual tiene dos subclases: "Círculo" y "Cuadrado". Ahora deseas ampliar esta jerarquía de clases para incluir información sobre colores, por lo tanto, planeas crear dos nuevas subclases de la clase "Forma" llamadas "Rojo" y "Azul". Sin embargo, dado que ya tienes dos subclases existentes, deberás crear cuatro combinaciones adicionales de clases, como por ejemplo "CírculoAzul" y "CuadradoRojo".

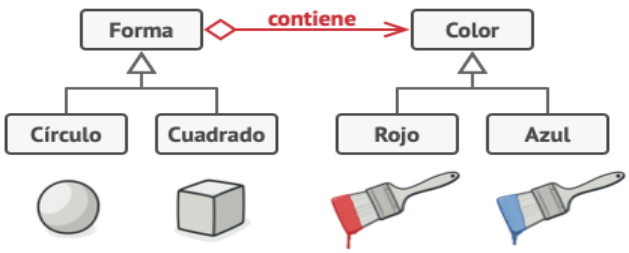


Solución

El patrón Bridge intenta resolver este problema pasando de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.

Pasando de la herencia a la composición del objeto.

Extrae una de las dimensiones a una jerarquía de clases separada.



Aplicabilidad

Usa el patrón Bridge cuando quieras dividir una clase monolítica que tenga muchas variantes de una funcionalidad.

Utiliza el patrón cuando necesites extender una clase en varias dimensiones.

Cómo implementarlo

1. Identifica las dimensiones principales de tus clases,
2. Define los métodos necesarios en la clase de abstracción
3. Crea clases de implementación concretas
4. Dentro de la clase de abstracción, agrega un campo de referencia para el tipo de implementación.
5. Si hay múltiples variantes de lógica, crea abstracciones extendiendo la clase base de abstracción para cada variante.
6. En el código del cliente, pasa un objeto de implementación al constructor de la abstracción para asociarlos

Pros y contras

- **Ventajas**

Puedes crear clases y aplicaciones independientes

Puedes introducir nuevas abstracciones e implementaciones

Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.

- **Desventajas**

Puede ser que el código se complique si aplicas a una clase muy relacionada.

Relación con otros patrones

- Bridge suele diseñarse por anticipado, pero, Adapter se utiliza habitualmente con una aplicación existente
- Puedes utilizar Abs. Factory junto a Bridge. Este emparejamiento resulta útil cuando algunas abstracciones sólo pueden funcionar con implementaciones específicas.
- Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras similares ya que, todos se basan delegar trabajo a otros objetos.

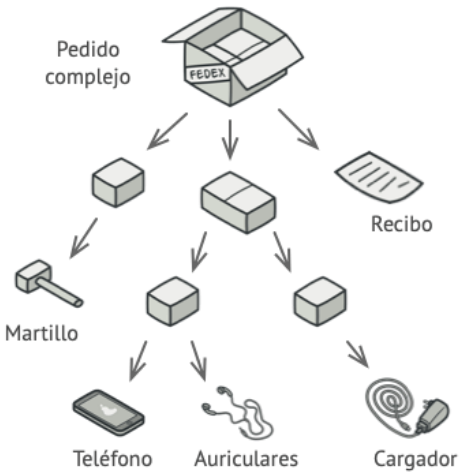
COMPOSITE

Propósito

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Problema

Su uso sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.



"Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo."

Solución

Sugiere trabajar con un método que:

- Para un producto, sencillamente devuelve el precio del producto.
- Para una caja, recorre cada artículo que contiene la caja.
- Si uno de esos artículos fuera una caja más pequeña, repasara su contenido y así sucesivamente

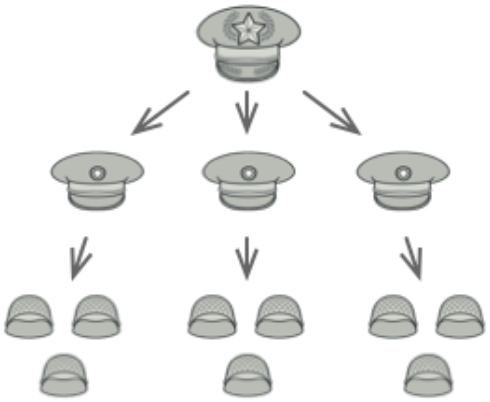
"El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos."

Analogía del mundo real

Los ejércitos de la mayoría de países se estructuran como jerarquías.

Un ejército está formado por varias divisiones.

- Una división es un grupo de brigadas
- una brigada está formada por pelotones,
- Un pelotón en escuadrones.
- Un escuadrón en grupos de soldados.



“Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo.”

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.

Cómo implementarlo

1. Asegurarse que el modelo puede representarse con estructura árbol e intenta dividirlo en elementos simples y contenedores.
2. Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
3. Crea una clase hoja para representar elementos simples.
4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos.
5. Define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

Pros y contras

- Pros

Puedes trabajar con estructuras de árbol complejas con mayor comodidad.

Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código.

- Contras

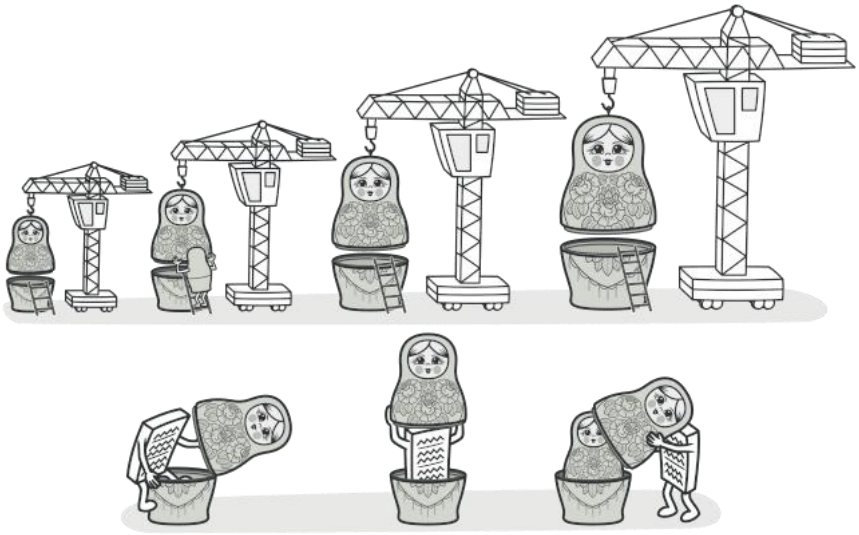
Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado.

Relaciones con otros patrones

- Puedes utilizar Builder al crear árboles Composite complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.
- Chain of Responsibility cuando un componente hoja recibe una solicitud, puede pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.
- Iteradores para recorrer árboles Composite.
- Visitor para ejecutar una operación sobre un árbol Composite entero.
- Puedes implementar nodos de hoja compartidos del árbol Composite como Flyweights para ahorrar memoria RAM.
- **Composite** y **Decorator** tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva.

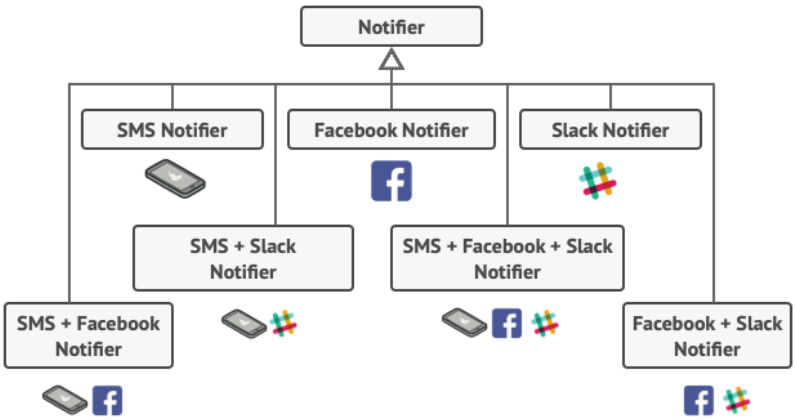
DECORATOR

Propósito: Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



Problema

El problema viene cuando necesitamos alterar la funcionalidad de un objeto, lo más fácil suele ser extender una clase y combinar métodos, lo que resulta en un código inflado.

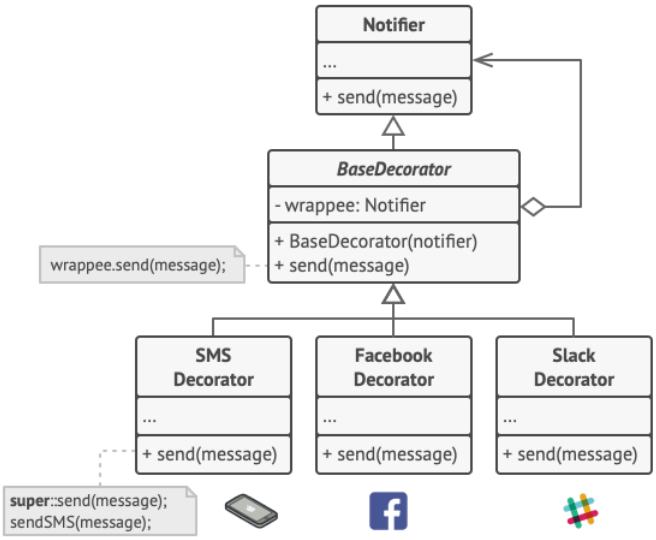


Solución

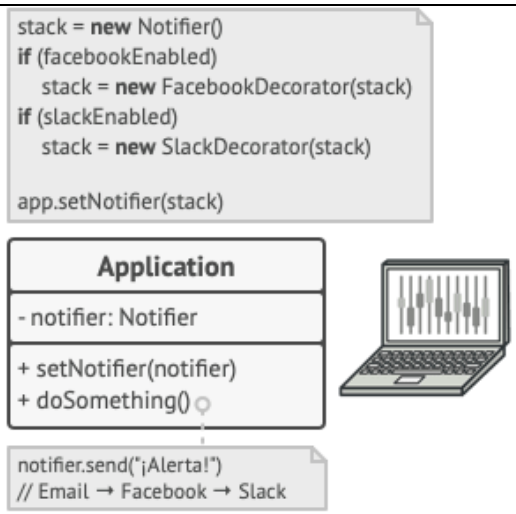
Una de las formas de superar las limitaciones de la herencia es usando la "Agregación" o la "Composición"

Con esto un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas.

Un objeto tiene una referencia a otro y le delega parte del trabajo, mientras que, con la herencia, el propio objeto hereda el comportamiento de su superclase.



El wrapper 0 decorator contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.



Analogía en el mundo real

Vestir ropa es un ejemplo del uso de decoradores. Todas estas prendas “extienden” tu comportamiento básico, pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo desees.

Aplicabilidad:

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

Cómo implementarlo

1. Asegúrate de que puede representar con un componente con varias capas.
2. Elegir los métodos más comunes y crear una interfaz con estos.
3. Crea una clase concreta de componente y define en ella el comportamiento base.
4. Crea una clase base decoradora. Debe tener un campo para almacenar una referencia a un objeto envuelto.

5. Asegúrate de que todas las clases implementan la interfaz de componente.
6. Crea decoradores concretos extendiéndolos a partir de la decoradora base.
7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

Pros y Contras

- **PROS**

Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.

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.

- **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 puede tener un aspecto desagradable.

Relaciones con otros patrones

- Adapter cambia la interfaz de un objeto existente mientras que Decorator mejora un objeto sin cambiar su interfaz.

- Proxy le proporciona la misma interfaz y Decorator le proporciona una interfaz mejorada.
- Chain of Responsibility y Decorator. Los manejadores de CoR pueden ejecutar operaciones arbitrarias con independencia entre sí. También pueden dejar de pasar la solicitud en cualquier momento. Por otro lado, varios decoradores pueden extender el comportamiento del objeto manteniendo su consistencia con la interfaz base. Además, los decoradores no pueden romper el flujo de la solicitud.
- Composite y Decorator pueden beneficiarse de Prototype. Aplicar el patrón te permite clonar estructuras complejas.
- Decorator te permite cambiar la piel de un objeto, mientras que Strategy te permite cambiar sus entrañas.
- Decorator y Proxy, Proxy gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los Decoradores siempre está controlada por el cliente.
- Composite y Decorator: Decorator añade responsabilidades adicionales al objeto envuelto, mientras que Composite se limita a "recapitular" los resultados de sus hijos.

FACADE

Propósito

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

Problema

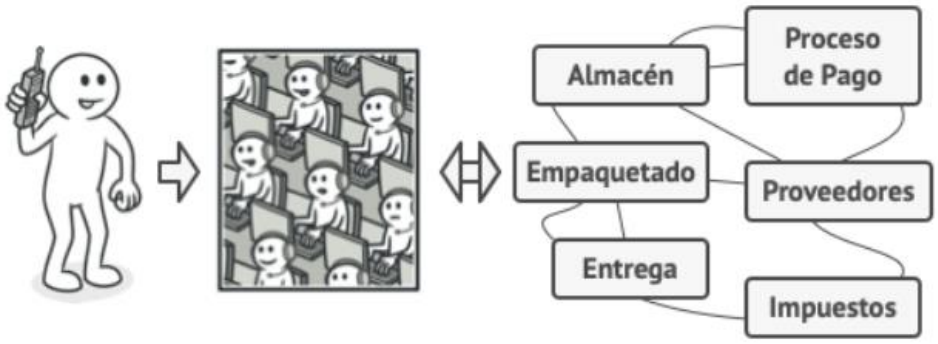
Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o framework. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.

Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.

Solución

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.

Analogía en el mundo real



Haciendo pedidos por teléfono.

Cuando llamas a una tienda para hacer un pedido por teléfono, un operador es tu fachada a todos los servicios y departamentos de la tienda. El operador te proporciona una sencilla interfaz de voz al sistema de pedidos, pasarelas de pago y varios servicios de entrega.

Aplicabilidad

- Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
- Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.

Cómo implementarlo

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente. Estás bien encaminado si esta interfaz hace que el código cliente sea independiente de muchas de las clases del subsistema.
2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. La fachada deberá ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a no ser que el código cliente ya lo haga.
3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, sólo tendrás que modificar el código de la fachada.
4. Si la fachada se vuelve demasiado grande, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

Pros Y Contras

Pros	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.

Relaciones con otros patrones

- Facade define una nueva interfaz para objetos existentes, mientras que Adapter intenta hacer que la interfaz existente sea utilizable. Normalmente Adapter sólo envuelve un objeto, mientras que Facade trabaja con todo un subsistema de objetos.
- Abstract Factory puede servir como alternativa a Facade cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.
- Flyweight muestra cómo crear muchos pequeños objetos, mientras que Facade muestra cómo crear un único objeto que represente un subsistema completo.
- Una clase fachada a menudo puede transformarse en una Singleton, ya que un único objeto fachada es suficiente en la mayoría de los casos.
- Facade es similar a Proxy en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializarla por su cuenta. Al contrario que Facade, Proxy tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.

FLYWEIGHT

Propósito

Flyweight 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.

Problema

Para divertirte un poco después de largas horas de trabajo, decides crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí. Decides implementar un sistema de partículas realistas que lo distinga de otros

juegos. Grandes cantidades de balas, misiles y metralla de las explosiones volarán por todo el mapa, ofreciendo una apasionante experiencia al jugador.

Al terminarlo, subes el último cambio, compilas el juego y se lo envías a un amigo para una partida de prueba. Aunque el juego funcionaba sin problemas en tu máquina, tu amigo no logró jugar durante mucho tiempo. En su computadora el juego se paraba a los pocos minutos de empezar. Tras dedicar varias horas a revisar los registros de depuración, descubres que el juego se paraba debido a una cantidad insuficiente de RAM. Resulta que el equipo de tu amigo es mucho menos potente que tu computadora, y esa es la razón por la que el problema surgió tan rápido en su máquina.

El problema estaba relacionado con tu sistema de partículas. Cada partícula, como una bala, un misil o un trozo de metralla, estaba representada por un objeto separado que contenía gran cantidad de datos. En cierto momento, cuando la masacre alcanzaba su punto culminante en la pantalla del jugador, las partículas recién creadas ya no cabían en el resto de RAM, provocando que el programa fallara.

Solución

El patrón Flyweight sugiere que dejemos de almacenar el estado extrínseco dentro del objeto. En lugar de eso, debes pasar este estado a métodos específicos que dependen de él. Tan solo el estado intrínseco se mantiene dentro del objeto, permitiendo que lo reutilices en distintos contextos. Como resultado, necesitarás menos de estos objetos, ya que sólo se diferencian en el estado intrínseco, que cuenta con muchas menos variaciones que el extrínseco.

Aplicabilidad

Utiliza el patrón Flyweight únicamente cuando tu programa deba soportar una enorme cantidad de objetos que apenas quepan en la RAM disponible.

La ventaja de aplicar el patrón depende en gran medida de cómo y dónde se utiliza. Resulta más útil cuando:

- La aplicación necesita generar una cantidad enorme de objetos similares
- Esto consume toda la ram disponible de un dispositivo objetivo
- Los objetos contienen estados duplicados que se pueden extraer y compartir entre varios objetos

Cómo implementarlo

1. Divide los campos de una clase que se convertirá en flyweight en dos partes:
    - El estado intrínseco: los campos que contienen información invariable duplicada a través de varios objetos
    - El estado extrínseco: los campos que contienen información contextual única de cada objeto
2. Deja los campos que representan el estado intrínseco en la clase, pero asegúrate de que sean inmutables. Deben llevar sus valores iniciales únicamente dentro del constructor.
3. Repasa los métodos que utilizan campos del estado extrínseco. Para cada campo utilizado en el método, introduce un nuevo parámetro y utilízalo en lugar del campo.
4. Opcionalmente, crea una clase fábrica para gestionar el grupo de objetos flyweight, buscando uno existente antes de crear uno nuevo. Una vez que la fábrica esté en su sitio, los clientes sólo deberán solicitar objetos flyweight a través de ella. Deberán describir el flyweight deseado pasando su estado intrínseco a la fábrica.

5. El cliente deberá almacenar o calcular valores del estado extrínseco (contexto) para poder invocar métodos de objetos flyweight. Por comodidad, el estado extrínseco puede moverse a una clase contexto separada junto con el campo referenciador del flyweight.

Pros Y Contras

Pros	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. El código se complica mucho. Los nuevos miembros del equipo siempre estarán preguntándose por qué el estado de una entidad se separó de tal manera.

Relaciones con otros patrones

- Puedes implementar nodos de hoja compartidos del árbol **Composite** como **Flyweights** para ahorrar memoria RAM.
- **Flyweight** muestra cómo crear muchos pequeños objetos, mientras que **Facade** muestra cómo crear un único objeto que represente un subsistema completo.
- **Flyweight** podría asemejarse a **Singleton** si de algún modo pudieras reducir todos los estados compartidos de los objetos a un único objeto flyweight. Pero existen dos diferencias fundamentales entre estos patrones:
    1. Solo debe haber una instancia Singleton, mientras que una clase *Flyweight* puede tener varias instancias con distintos estados intrínsecos.

2. El objeto *Singleton* puede ser mutable. Los objetos flyweight son inmutables.

PROXY

Propósito

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.

Problema

¿Por qué querrías controlar el acceso a un objeto? Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.

Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario. Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.

En un mundo ideal, querríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

Solución

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto

proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.

Analogía en el mundo real

Una tarjeta de crédito es un proxy de una cuenta bancaria, que, a su vez, es un proxy de un manojo de billetes. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un pago. El consumidor se siente genial porque no necesita llevar un montón de efectivo encima. El dueño de la tienda también está contento porque los ingresos de la transacción se añaden electrónicamente a la cuenta bancaria de la tienda sin el riesgo de perder el depósito o sufrir un robo de camino al banco.

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.
- 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.

Cómo implementarlo

1. Si no hay una interfaz de servicio preexistente, crea una para que los objetos de proxy y de servicio sean intercambiables. No siempre resulta posible extraer la interfaz de la clase servicio, porque tienes que cambiar todos los clientes del servicio para utilizar esa interfaz. El plan B consiste en convertir el proxy en una subclase de la clase servicio, de forma que herede la interfaz del servicio.
2. Crea la clase proxy. Debe tener un campo para almacenar una referencia al servicio. Normalmente los proxis crean y gestionan el ciclo de vida completo de sus servicios. En raras ocasiones, el cliente pasa un servicio al proxy a través de un constructor.
3. Implementa los métodos del proxy según sus propósitos. En la mayoría de los casos, después de hacer cierta labor, el proxy debería delegar el trabajo a un objeto de servicio.
4. Considera introducir un método de creación que decida si el cliente obtiene un proxy o un servicio real. Puede tratarse de un simple método estático en la clase proxy o de todo un método de fábrica.
5. Considera implementar la inicialización diferida para el objeto de servicio.

Pros Y Contras

Pros	Contras
Puedes controlar el objeto de servicio sin que los clientes lo sepan.	El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.

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 proxis sin cambiar el servicio o los clientes.	La respuesta del servicio puede retrasarse.

Relaciones con otros patrones

- Adapter proporciona una interfaz diferente al objeto envuelto, Proxy le proporciona la misma interfaz y Decorator le proporciona una interfaz mejorada.
- Facade es similar a Proxy en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializarla por su cuenta. Al contrario que Facade, Proxy tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.
- Decorator y Proxy tienen estructuras similares, pero propósitos muy distintos. Ambos patrones se basan en el principio de composición, por el que un objeto debe delegar parte del trabajo a otro. La diferencia es que, normalmente, un Proxy gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los Decoradores siempre está controlada por el cliente.

2.5 Conclusiones

Los patrones de diseño estructurales son herramientas útiles para abordar problemas de diseño en el desarrollo de software. Cada uno de estos patrones ofrece soluciones específicas para problemas comunes, como la incompatibilidad de interfaces, la composición de objetos complejos, la extensión de funcionalidades y la simplificación de sistemas complejos.

Estos patrones permiten mejorar el modularidad, la reutilización de código y la flexibilidad del diseño, al separar preocupaciones, reducir la dependencia entre clases y facilitar la adición o modificación de funcionalidades sin afectar el código existente.

Sin embargo, también se mencionan algunas desventajas, como el aumento de la complejidad del código al introducir nuevas interfaces y clases, y la posible dificultad para proporcionar una interfaz común para clases con funcionalidades diferentes.

En resumen, los patrones de diseño estructurales ofrecen soluciones prácticas y probadas para problemas de diseño en el desarrollo de software, pero es importante evaluar cuidadosamente su aplicabilidad y considerar tanto las ventajas como las desventajas antes de implementarlos.

Ejemplos:

https://github.com/BryanCurillo/Patrones_Estructurales.git

Diapositivas:

https://www.canva.com/design/DAFjNVDB6x4/RL-sCTT5nbW8rR9FOYNRLw/edit?utm_content=DAFjNVDB6x4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

2.6 Bibliografía

Descripción en norma APA
Refactoring.guru. (s.f.). Adapter. Recuperado de https://refactoring.guru/design-patterns/adapter Refactoring.guru. (s.f.). Bridge. Recuperado de https://refactoring.guru/design-patterns/bridg Refactoring.guru. (s.f.). Composite. Recuperado de https://refactoring.guru/design-patterns/composite Refactoring.guru. (s.f.). Decorator. Recuperado de https://refactoring.guru/design-patterns/decorator

Refactoring.guru. (s.f.). Facade.

Recuperado de https://refactoring.guru/design-patterns/facade

Refactoring.guru. (s.f.). Flyweight.

Recuperado de https://refactoring.guru/design-patterns/flyweight

Refactoring.guru. (s.f.). Proxy.

Recuperado de https://refactoring.guru/design-patterns/proxy