4 SOFTWARE DEVELOPERS

AUTOR
JORDY ANDRES
RODRIGUEZ ARANGO

PROGRAMA DE FORMA GENÉRICA

UTILIZANDO PATRONES DE DISEÑO

INCLUYE EJEMPLOS DE CÓDIGO CON JAVA Y TYPESCRIPT

www.4softwaredevelopers.com

Índice

Índice **2**
Consideraciones **5**
Acerca del Autor **8**
Contacto **9**
Introducción **10**
Objetivos de los patrones de diseño **12**
Tipos de patrones **12**
Patrones de arquitectura 13
Patrones de diseño 13
Patrones de dialecto 13
Relación de patrones de diseño (GoF) **14**
Patrones de creación **14**
Abstract Factory 14
Factory Method 21
Diferencias entre Abstract Factory y Factory Method 26
Singleton 27
Builder 31
Prototype 38
Conclusión 43
Patrones de estructura **46**
Adapter 46
¿Qué rol ocupa cada clase? 50
Bridge 51
Composite 62
Decorator 67
Facade 72
Flyweight 77
Proxy 82
Conclusión 88
Patrones de Comportamiento **90**
Chain of Responsibility 90
Command 98
Interpreter 108
Iterator 114

Mediator 121
Memento 128
Observer 133
State 140
Strategy 151
Template Method 157
Visitor 163
Conclusión 168
Conclusiones generales **171**
La importancia de los patrones de diseño 171

Consideraciones

Bienvenido a este ebook sobre patrones de diseño en TypeScript y Java. Antes de sumergirte en los ejemplos y conceptos que aquí se presentan, es importante tener en cuenta algunas consideraciones iniciales que te ayudarán a aprovechar al máximo este recurso.

Simplicidad con Aplicabilidad Real

Los ejemplos que encontrarás en este ebook han sido cuidadosamente diseñados para ser simples y comprensibles. Sin embargo, no te dejes engañar por su aparente simplicidad. A pesar de su enfoque claro y conciso, cada ejemplo es una representación tangible de un patrón de diseño aplicable en el mundo real.

Por ejemplo, puedes encontrarte con un caso de "Lector de Documentos" que no busca ser una aplicación completa, sino más bien un bosquejo que demuestra cómo se aplica un patrón de diseño en la práctica. Este enfoque se ha elegido deliberadamente para ayudarte a comprender los conceptos esenciales detrás de cada patrón y cómo se traducen en situaciones reales de desarrollo.

Comprender los Principios

Los patrones de diseño son herramientas poderosas, pero su verdadero valor radica en comprender los principios subyacentes que los respaldan. Cada patrón tiene un propósito y una utilidad específica en el diseño de software, y es fundamental comprender cuándo y por qué aplicarlos.

A medida que explores los ejemplos, presta atención a los principios detrás de cada patrón y cómo se traducen en soluciones

prácticas. Esto te permitirá aplicar los patrones de manera efectiva en tus propios proyectos, incluso cuando las situaciones sean más complejas que los ejemplos proporcionados.

Fomentar la Creatividad

Aunque los patrones de diseño ofrecen guías sólidas, no son soluciones inflexibles. En el mundo real, a menudo se requiere creatividad para adaptar los patrones a situaciones específicas. A medida que te familiarices con estos patrones, te alentamos a explorar cómo puedes combinarlos y personalizarlos para resolver desafíos únicos.

Disponibilidad del código en GitHub

No olvides solicitar el acceso a nuestro repositorio de GitHub con cada uno de los ejemplos de código, es posible que el ebook a pesar de nuestros esfuerzos no resulte el entorno más cómodo para ver o editar el código, así que en cada ejemplo, tendrás un link hacía la ruta exacta donde se encuentra el patrón de diseño, ya sea en JavaScript o TypeScript, con tu compra debes haber recibido un archivo con las instrucciones llamado **“README.csv”**

Aprender y Experimentar

El aprendizaje de los patrones de diseño es un proceso continuo. A medida que avanzas en tu carrera como desarrollador, te encontrarás con una variedad de desafíos y oportunidades para aplicar estos patrones de manera efectiva. No dudes en experimentar con ellos y adaptarlos según sea necesario.

En resumen, este ebook está diseñado para ayudarte a comprender los patrones de diseño y cómo se aplican en TypeScript y Java. Cada ejemplo, aunque simple, es un paso hacia la comprensión más profunda de estas herramientas esenciales en el mundo del

desarrollo de software. Disfruta del viaje de aprendizaje y aplícalo para crear software robusto y eficiente.

¡Comencemos a explorar los patrones de diseño juntos!

Acerca del Autor

Hola soy Jordy Rodríguez CEO y fundador de 4SoftwareDevelopers, CTO y Co-Fundador de HUSO Group SAS.

Soy un desarrollador de software con 10 años de experiencia en todo ese tiempo he tenido la oportunidad de trabajar en muchísimos proyectos en muchos de ellos he utilizado lenguajes y herramientas como Java, TypeScript, Spring Framework, Hibernate, NestJS, Angular, React Native, Python y NodeJS.

También he liderado múltiples proyectos en distintas organizaciones. Me considero a mi mismo como un apasionado por la academia, trato de vivir día a día, en busca de nuevos retos y tecnologías informáticas.

Contacto

Retroalimentación:

Creo firmemente en la mejora continua, la cual obedece al principio de la constante retroalimentación, por eso, de antemano agradezco si usted desea apoyar mi proceso de mejora continua con cualquier comentario acerca de este título, solo debe enviar un correo a ebooks@4softwaredevelopers.com, no olvide mencionar en el asunto el título del libro.

Piratería:

Si encuentras copias ilegales de esta obra no dudes en denunciarlas para ello puedes enviar un correo a Jordy Rodriguez, no olvides incluir en el cuerpo del correo un enlace al sitio donde se encuentra la copia ilegal.

Redes Sociales

Puedes tener un contacto más directo conmigo en Twitter.

Introducción

Antes de iniciar debemos tener claro que son los patrones de diseño y su origen, los patrones de diseño son una técnica que nos permite resolver problemas comunes a la hora de desarrollar software, de una forma muy resumida los patrones de diseño son la solución a un problema de diseño.

Algo a tener en cuenta es que para que una pieza de código sea considerada un patrón de diseño debe cumplir ciertas características una de ellas es que todo ya ha sido probado resolviendo problemas anteriormente por lo cual es efectivo, además de esto debe ser reutilizable por lo cual podremos usar el mismo patrón en diferentes circunstancias.

Resulta curioso pensar que los patrones de diseño tienen un inicio directamente relacionado con la arquitectura y no con la arquitectura de software, en el año 1979 el arquitecto Christopher Alexander escribió el libro ***The Timeless Way of Building*** en el cual propone el uso de una serie de patrones para la construcción de edificios de mayor calidad.

Probablemente la frase más famosa del autor es la siguiente: *“Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez.”*

Resulta curioso pensar que esta frase no está relacionada con el desarrollo de software sin embargo es muy acertada.

Más adelante en 1987 Ward Cunningham y Kent Beck crearon un libro llamado ***Pattern Languages for OO Programs***, ellos dos

notaron una similitud entre la buena arquitectura propuesta por Alexander y por la buena arquitectura de la Programación Orientada a Objetos (POO).

Sin embargo no fue sino hasta 1990 que los patrones de diseño tomaron una gran popularidad en la industria del desarrollo de software todo esto a partir de la publicación del libro ***Design Patterns***.

Este libro fue escrito por el grupo Gang of Four **(GoF)** compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides en el que se recogían 23 patrones de diseño comunes en estos patrones de diseño nos vamos a enfocar a lo largo de este libro.

El objetivo de este libro es proporcionar el conocimiento suficiente para que tú como desarrollador puedas encontrar una mejor solución a los diferentes problemas de código a los cuales te enfrentas día a día.

Probablemente muchas veces haz sentido que tienes código que se repite constantemente o que la solución que le diste a un problema no es del todo correcta, te seré sincero este libro no te va a quitar esa sensación sin embargo te dará las herramientas para que tú mismo dejes de sentirte así y resuelvas problemas de una mejor forma.

Objetivos de los patrones de diseño

Los patrones de diseño cuentan con una serie de objetivos que nos garantiza y fomentan su uso:

- Proporcionar elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente (no reinventemos la rueda).
- Formalizar un vocabulario común entre diseñadores y desarrolladores de software.
- Estandarizar la manera en la que construimos software.
- Facilitar el aprendizaje para las nuevas generaciones de desarrolladores.
- Cabe resaltar que los patrones de diseños en ningún momento pretenden: Imponer la forma en que creamos código.
- Eliminar el proceso creativo que conlleva el desarrollo de software.

De lo anterior podemos concluir que no es obligatorio utilizar patrones de diseño, vale la pena plantearnos su uso en los casos en que tengamos un problema igual o similar al que soluciona determinado patrón, sin embargo pueden existir casos en los que los patrones de diseño no sean aplicables y adicionalmente forzar su uso puede ser un grave error.

Tipos de patrones

Generalmente los patrones de diseño se clasifican de acuerdo a su nivel de abstracción, teniendo en cuenta esto, podríamos decir que existen tres tipos de patrones de diseño:

Patrones de arquitectura

Los patrones de arquitectura son aquellos que determinan la estructura de un sistema de información aquí podríamos dar como ejemplo la arquitectura de microservicios, el patrón MVC y la programación por capas.

Patrones de diseño

Este tipo de patrón expresa decisiones ligadas al diseño o sus relaciones, generalmente nos indican una serie de pasos a seguir para resolver un problema a nivel de código o para implementar código altamente reutilizable y fácil de implementar.

En las secciones posteriores de este ebook iremos describiendo los diferentes patrones de diseño con sus respectivas subcategorías, además podrás ver la implementación de cada patrón en Java y Typescript.

Patrones de dialecto

Son aquellos exclusivos a un lenguaje de programación en específico o para un entorno en concreto.

Relación de patrones de diseño (GoF)

Patrones de creación

Corresponden a patrones de diseño de software que solucionan específicamente problemas relacionados con la creación de instancias, de esta forma nosotros podemos encapsular y abstraer dicha relación.

La instancia excesiva de una o muchas clases nos puede indicar que tenemos algunos problemas de diseño en nuestro código, a diferencia de lo que muchos pueden pensar esto no se soluciona únicamente creando métodos o miembros de clase estáticos.

El grupo de los GoF diseñó una serie de patrones que nos pueden ayudar de acuerdo a la necesidad por la que estemos atravesando en determinado momento, veamos.

Abstract Factory

Debido a que es un patrón del tipo creacional ya sabemos que tipo de problemas nos puede ayudar a resolver, este patrón básicamente nos permite trabajar con objetos de distintas familias, evitando que se mezclen entre sí pero teniendo claro que su objetivo es que estos objetos puedan trabajar juntos, vamos a plantear un ejemplo simple.

Imaginemos que estamos construyendo un juego de rol (RPG) y necesitamos crear diferentes tipos de personajes y armas.

Veamos primero el ejemplo en Java.

Primero, definiremos las interfaces que representarán a las familias de objetos: `Character` y `Weapon`.

```
// Interfaces para personajes y armas
public interface Character {
    String getDescription();
}

public interface Weapon {
    String getDescription();
}
```

Luego, crearemos implementaciones concretas de esas interfaces para dos diferentes tipos de personajes: elfos y orcos, y dos tipos de armas: espadas y arcos.

```
// Implementaciones concretas de personajes
public class Elf implements Character {
    @Override
    public String getDescription() {
        return "Este es un elfo.";
    }
}
public class Orc implements Character {
    @Override
    public String getDescription() {
        return "Este es un orco.";
    }
}


// Implementaciones concretas de armas
public class Sword implements Weapon {
    @Override
```

```
    public String getDescription() {
        return "Esta es una espada.";
    }
}

public class Bow implements Weapon {
    @Override
    public String getDescription() {
        return "Este es un arco";
    }
}
```

Finalmente, definiremos `AbstractFactory` que tendrá métodos para crear personajes y armas.

```
// Fábrica abstracta
public interface AbstractFactory {
    Character createCharacter();
    Weapon createWeapon();
}

// Implementaciones concretas de la fábrica abstracta
public class ElfFactory implements AbstractFactory {
    @Override
    public Character createCharacter() {
        return new Elf();
    }
    @Override
    public Weapon createWeapon() {
        return new Bow();
    }
}
```

```
public class OrcFactory implements AbstractFactory {
```

```
    @Override
    public Character createCharacter() {
        return new Orc();
    }

    @Override
    public Weapon createWeapon() {
        return new Sword();
    }
}
```

```
public class Main {

     public static void main(String[] args) {
          var elfFactory = new ElfFactory();

          var elfCharacter =
          elfFactory.createCharacter();

          var elfWeapon = elfFactory.createWeapon();

          System.out.println(
           elfCharacter.getDescription() +
           elfWeapon.getDescription()
          );

          var orcFactory = new OrcFactory();
          var orcCharacter =
          orcFactory.createCharacter();
          var orcWeapon = orcFactory.createWeapon();

          System.out.println(
           orcCharacter.getDescription() +
           orcWeapon.getDescription()
          );

          System.out.println("Personajes listos para
```

```
            el combate...");
        }

}
```

Ahora el ejemplo en TypeScript:

Primero, definiremos las interfaces que representarán a las familias de objetos: `Character` y `Weapon`.

```
// Interfaces para personajes y armas
public interface Character {
    getDescription(): string;
}

public interface Weapon {
    getDescription(): string;
}
```

Luego, crearemos implementaciones concretas de esas interfaces para dos diferentes tipos de personajes: elfos y orcos, y dos tipos de armas: espadas y arcos.

```
// Implementaciones concretas de personajes
public class Elf implements Character {
    getDescription() {
        return "Este es un elfo.";
    }
}
public class Orc implements Character {
    getDescription() {
        return "Este es un orco.";
    }
}
```

```
// Implementaciones concretas de armas
public class Sword implements Weapon {
    getDescription() {
        return "Esta es una espada.";
    }
}


public class Bow implements Weapon {
    getDescription() {
        return "Este es un arco.";
    }
}
```

Ahora, definiremos `AbstractFactory` que tendrá métodos para crear personajes y armas.

```
// Fábrica abstracta
public interface AbstractFactory {
    createCharacter(): Character;
    createWeapon(): Weapon;
}
```

Finalmente podemos implementar nuestro `AbstractFactory` para crear diferentes personajes y asignarles armas.

```
// Implementaciones concretas de la fábrica abstracta
public class ElfFactory implements AbstractFactory {
    createCharacter() {
        return new Elf();
    }
    createWeapon() {
        return new Bow();
    }
}
public class OrcFactory implements AbstractFactory {
```

```
    createCharacter() {
        return new Orc();
    }

    createWeapon() {
        return new Sword();
    }
}
// Crear una fábrica de elfos y crear un personaje y un
arma
const elfFactory = new ElfFactory();
const elfCharacter = elfFactory.createCharacter();
const elfWeapon = elfFactory.createWeapon();
console.log(elfCharacter.getDescription());
```

```
console.log(elfWeapon.getDescription());

// Crear una fábrica de orcos y crear un personaje y un
arma
const orcFactory = new OrcFactory();
const orcCharacter = orcFactory.createCharacter();
const orcWeapon = orcFactory.createWeapon();

console.log(orcCharacter.getDescription()); // Output:
Este es un orco.
console.log(orcWeapon.getDescription()); // Output: Esta
es una espada.
```

La primera gran noticia es que espero te hayas dado cuenta lo parecidos que son TypeScript y Java, si solo conoces uno de estos dos lenguajes, te ánimo a que pruebes el que no conozcas ya verás que la curva de aprendizaje no es tan grande como crees.

Ahora bien, más allá de que hayas entendido el código necesito que empieces a pensar de una forma genérica, este ejemplo se

puede trasladar a cualquier software de la vida real, en lugar de orcos y elfos podríamos ser una fábrica de usuarios o una fábrica de roles, ese es el principal objetivo de los patrones de diseño.

Ayudarnos a resolver problemas comunes, en lugar de tener esparcido en el código un sin número de clases que se instancian sin motivo aparente en este caso particular podríamos tener una fábrica que se encargue de crear instancias de lo que necesitemos.

Esto es gracias a que tenemos una serie de interfaces globales capaces de cubrir la particularidad de clases hijas, por ejemplo la interfaz `Character` existe como una generalidad de cualquier tipo de `Character` que podamos llegar a necesitar no importa si es un Orco o Elfo, lo importante es saber que más adelante necesitamos agregar otro `Character` por ejemplo Duende gracias a la interfaz `Character` lo podremos hacer sin cambiar mucho nuestro código.

Lo mismo podemos decir de la interfaz `AbstractFactory` al ser una interfaz general, puede cubrir todas las otras posibilidades de nuestra aplicación, justo eso programar de forma genérica, crear código capaz de soportar bien el cambio, se dice que soporta bien el cambio porque se puede volver a usar, es posible que requiera ajustes pero no una reconstrucción total.

Ya que desde su diseño ha sido concebido como código capaz de ser reutilizado en otros casos.

Factory Method

El Factory Method, es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una clase base, pero delega la responsabilidad de la creación a las subclases concretas. En otras palabras, el Factory Method define un método abstracto en una clase base que las subclases deben implementar para crear instancias de un objeto. Esto permite que las subclases elijan qué clase concreta de objeto crear, mientras que el código que utiliza estos objetos sigue siendo independiente de las clases concretas específicas, veamos un ejemplo:

Supongamos que estamos construyendo una aplicación de creación de contenido multimedia, y necesitamos un mecanismo para crear diferentes tipos de elementos multimedia, como imágenes y videos.

Primero, definiremos una interfaz `Media` que representará los elementos multimedia y tendrá un método para obtener una descripción. Luego, crearemos clases concretas que implementarán esta interfaz para representar imágenes y videos. A continuación, definiremos la clase abstracta `MediaFactory` que tendrá un método abstracto llamado `createMedia`() que será implementado por las subclases concretas. Estas subclases concretas, como `ImageFactory` y `VideoFactory`, se encargarán de crear instancias específicas de `Media`.

Primero veamos un ejemplo en Java:

```
// Interfaz para elementos multimedia
public interface Media {
    String getDescription();
}
```

```java
// Implementaciones concretas de elementos multimedia
public class Image implements Media {
    @Override
    public String getDescription() {
        return "Esto es una imagen.";
    }
}
```

```java
public class Video implements Media {
    @Override
    public String getDescription() {
        return "Esto es un video.";
    }
}
```

```java
// Clase abstracta de fábrica
public abstract class MediaFactory {
    public abstract Media createMedia();
}
// Implementaciones concretas de fábricas
public class ImageFactory extends MediaFactory {
    @Override
    public Media createMedia() {
        return new Image();
    }
}
public class VideoFactory extends MediaFactory {
    @Override
    public Media createMedia() {
        return new Video();
    }
}
```

```java
public class Main {
  public static void main(String[] args) {
   // Creamos una fábrica de imágenes
```

```
    // Y creamos un elemento multimedia
    MediaFactory imageFactory = new ImageFactory();
    Media image = imageFactory.createMedia();
    System.out.println(image.getDescription());
    // Creamos una fábrica de videos
    // Y creamos un elemento multimedia
    MediaFactory videoFactory = new VideoFactory();
    Media video = videoFactory.createMedia();
    System.out.println(video.getDescription());
  }
}
```

Ahora veamos el ejemplo en TypeScript:

```
// Interfaz para elementos multimedia
interface Media {
    getDescription(): string;
}

// Implementaciones concretas de elementos multimedia
class MyImage implements Media {
    getDescription() {
        return "Esto es una imagen.";
    }
}
```

```
class Video implements Media {
    getDescription() {
        return "Esto es un video.";
    }
}
// Clase abstracta de fábrica
abstract class MediaFactory {
    abstract createMedia(): Media;
}
```

```
class ImageFactory extends MediaFactory {
    createMedia() {
        return new MyImage();
    }
}

class VideoFactory extends MediaFactory {
    createMedia() {
        return new Video();
    }
}
```

```
// Creamos una fábrica de imágenes y creamos un elemento multimedia
const imageFactory: MediaFactory = new ImageFactory();
const image: Media = imageFactory.createMedia();
console.log(image.getDescription());

// Creamos una fábrica de videos y creamos un elemento multimedia
const videoFactory: MediaFactory = new VideoFactory();
const video: Media = videoFactory.createMedia();
console.log(video.getDescription());
```

Es posible que en este punto estés confundido con respecto a la diferencia entre Abstract Factory y Factory Method, te voy a explicar las diferencias pero en este momento quiero explicarte la principal.

Abstract Factory está pensado para crear objetos que se relacionan entre sí, permitiendo que estos trabajen juntos sin la necesidad de estar mezclados en un solo objeto, en nuestro ejemplo de Abstract Factory creamos una fábrica capaz de generar personajes y asignarles un arma todo esto sin la necesidad de crear una única clase llena de atributos que no tienen sentido.

Sin embargo con Factory Method crear objetos de un único tipo, solo objetos relacionados a la multimedia tienen la posibilidad de ser creados a través de nuestro ejemplo, por ende podríamos concluir que la principal diferencia es la razón de su uso, uno es para permitir crear objetos de diferentes tipos que están relacionados sin que se mezclen en un único mientras que el otro es para crear objetos de un mismo tipo que tendrá muchas variaciones.

Tal y como te dije los patrones de diseño tienen esta particularidad de ser pensados y diseñados sobre conceptos globales, en nuestro primer ejemplo lo global venía con clases como `Character` y `Weapon` ahora lo global viene con la clase Media y todas las posibilidades que estas pueden llegar a cubrir.

Diferencias entre Abstract Factory y Factory Method

Factory Method:

- El Factory Method se centra en la creación de un solo tipo de objeto, que se deriva de una clase base o interfaz común.

- Define un método abstracto en una clase base (la fábrica) que las subclases concretas deben implementar para crear instancias del objeto deseado.

- Permite que las subclases controlen el proceso de creación del objeto.

- Es útil cuando necesitas delegar la responsabilidad de crear un objeto específico a subclases.

- Puede ser útil cuando necesitas una forma de personalizar o extender la creación de objetos en subclases sin cambiar el código que utiliza el objeto.

Singleton

El patrón de diseño Singleton es un patrón creacional que garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Esto significa que, independientemente de cuántas veces se solicite crear una instancia de la clase, siempre se devolverá la misma instancia única.

En resumen, el Singleton se utiliza para asegurarse de que una clase tenga una sola instancia en toda la aplicación y proporcionar una forma de acceder a esa instancia desde cualquier punto del código. Esto es especialmente útil en situaciones en las que necesitas una única fuente de verdad o cuando deseas limitar el número de instancias de una clase por razones de control, rendimiento o coherencia.

Ahora vemos un ejemplo que nos puede ayudar a entender mejor.

Supongamos que estamos construyendo una aplicación de registro de eventos y necesitamos una clase para manejar la configuración del registro. Queremos asegurarnos de que solo haya una instancia de esta clase en todo el sistema.

Primero el código en Java:

```
public class EventLogger {
    private static EventLogger instance;
```

```
    private String logFileName;

    private EventLogger() {
        logFileName = "event_log.txt";
    }

    public static EventLogger getInstance() {
        if (instance == null) {
            instance = new EventLogger();
        }
        return instance;
    }


    public void logEvent(String eventMessage) {
        // Simulamos que escribimos el evento
       // en un archivo de log
       System.out.println(
         "Logging event: " +
         eventMessage + " to " + logFileName
       );
    }
    public void setLogFileName(String fileName) {
        logFileName = fileName;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        /*
```

```
            Obtenemos la instancia singleton de
            EventLogger
        */
        EventLogger logger =
        EventLogger.getInstance();

        // Escribimos en el log
        logger.logEvent(
         "Event 1: Application started"
        );
        logger.logEvent("Event 2: User logged in");

        // Asignamos el nombre del archivo de log
        logger.setLogFileName("custom_log.txt");

        // Escribimos otro evento
        logger.logEvent("Event 3: Data processed");
    }
}
```

Ahora veamos el ejemplo en TypeScript:

```
class EventLogger {
    private static instance: EventLogger;
```

```
    private logFileName: string;

    private constructor() {
        this.logFileName = "event_log.txt";
    }
    static getInstance(): EventLogger {
        if (!EventLogger.instance) {
          EventLogger.instance = new EventLogger();
        }
        return EventLogger.instance;
    }
    logEvent(eventMessage: string): void {
        // Simulate writing the event to a log file
        console.log(
         `Logging event: ${eventMessage} to
         ${this.logFileName}`
        );
    }

    setLogFileName(fileName: string): void {
        this.logFileName = fileName;
    }
}
```

```
// Main code
const logger = EventLogger.getInstance();

logger.logEvent("Event 1: Application started");
logger.logEvent("Event 2: User logged in");

logger.setLogFileName("custom_log.txt");

logger.logEvent("Event 3: Data processed");
```

En estos ejemplos, la clase `EventLogger` implementa el patrón Singleton. La clase tiene un constructor privado para evitar que se creen instancias directamente desde fuera de la clase. El método

`getInstance()` es estático y devuelve la única instancia existente de la clase. Si aún no existe una instancia, se crea una nueva; de lo contrario, se devuelve la instancia existente.

Después, se muestra cómo se obtiene la instancia única de `EventLogger` y cómo se utilizan sus métodos para registrar eventos. El Singleton asegura que solo haya una instancia de `EventLogger` en todo el programa, lo que es útil para mantener la coherencia en la configuración y el estado de registro en todo el sistema.

Builder

El patrón de diseño Builder es un patrón creacional que separa la construcción de un objeto complejo de su representación, permitiendo crear diferentes representaciones utilizando el mismo proceso de construcción. Este patrón se utiliza cuando el proceso de construcción de un objeto es complejo y puede involucrar múltiples pasos o configuraciones opcionales.

El patrón Builder define una interfaz abstracta (el Builder) que declara métodos para construir las partes individuales del objeto y un método para obtener el resultado final. Luego, se crean clases concretas (los ConcreteBuilders) que implementan esa interfaz y proporcionan implementaciones específicas para construir las partes. Además, se puede tener una clase Director opcional que controla el orden y el flujo de construcción utilizando un objeto Builder.

Este patrón es útil cuando se necesita crear objetos con configuraciones complejas y se desea evitar la proliferación de constructores con múltiples parámetros o combinaciones de métodos. El Builder facilita la construcción paso a paso, permitiendo

ajustar y personalizar cada parte del objeto antes de obtener el resultado final.

Veamos un ejemplo que nos ayude a entender mejor.

Supongamos que estamos construyendo un sistema de generación de informes y necesitamos construir informes complejos con diferentes secciones, como encabezado, cuerpo y pie de página.

Primero, definiremos una clase `Report` que representará el informe final. Luego, crearemos una interfaz `ReportBuilder` con métodos para construir las diferentes secciones del informe. Después, crearemos una clase concreta `ConcreteReportBuilder` que implementa la interfaz ReportBuilder y se encargará de construir cada sección del informe. Finalmente, definiremos una clase `ReportDirector` que utilizará el builder para construir el informe en un orden específico.

Primero el código en Java:

```java
// Clase que representa el informe
public class Report {
    private String header;
    private String body;
    private String footer;

    public void setHeader(String header) {
        this.header = header;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public void setFooter(String footer) {
```

```
        this.footer = footer;
    }

    public String getResult() {
        return header + "\n" + body +
        "\n" + footer;
    }
}
// Interfaz del Builder
public interface ReportBuilder {
    void buildHeader(String header);

    void buildBody(String body);

    void buildFooter(String footer);

    Report getResult();
}

// Implementación concreta del Builder
public class ConcreteReportBuilder
implements ReportBuilder {
    private Report report = new Report();

    public void buildHeader(String header) {
        report.setHeader(header);
    }

    public void buildBody(String body) {
        report.setBody(body);
    }

    public void buildFooter(String footer) {
        report.setFooter(footer);
    }

    public Report getResult() {
        return report;
```

```
    }
}

// Director que construye el informe en un orden
específico
public class ReportDirector {
    private ReportBuilder builder;

    public ReportDirector(ReportBuilder builder)
    {
       this.builder = builder;
    }
    public void build() {
      builder.buildHeader("Report Header");
      builder.buildBody("Report Body");
      builder.buildFooter("Report Footer");
    }
}

public class Main {
    public static void main(String[] args) {
        ReportBuilder builder =
        new ConcreteReportBuilder();

        ReportDirector director =
        new ReportDirector(builder);

        director.build();
        Report report = builder.getResult();

        System.out.println(report.getResult());
     }
}
```

Ahora veamos el código en TypeScript:

```
// Clase que representa el informe
class MyReport {
    private header: string = "";
    private body: string = "";
    private footer: string = "";


    setHeader(header: string) {
        this.header = header;
    }

    setBody(body: string) {
        this.body = body;
    }

    setFooter(footer: string) {
        this.footer = footer;
    }

    getResult(): string {
        return
       `${this.header}\n
        ${this.body}\n${this.footer}`;
    }
}
// Interfaz del Builder
interface ReportBuilder {
    buildHeader(header: string): void;
    buildBody(body: string): void;
    buildFooter(footer: string): void;
    getResult(): MyReport;
}




// Implementación concreta del Builder
```

```
class ConcreteReportBuilder implements ReportBuilder {
    private report: MyReport = new MyReport();

    buildHeader(header: string) {
        this.report.setHeader(header);
    }

    buildBody(body: string) {
        this.report.setBody(body);
    }

    buildFooter(footer: string) {
        this.report.setFooter(footer);
    }
    getResult(): MyReport {
        return this.report;
    }
}
// Director que construye el informe en un orden
específico
class ReportDirector {
    private builder: ReportBuilder;

    constructor(builder: ReportBuilder) {
        this.builder = builder;
    }
    build() {
        this.builder.buildHeader("Report Header");
        this.builder.buildBody("Report Body");
        this.builder.buildFooter("Report Footer");
    }
}




// Código principal
```

```
const builder: ReportBuilder = new
ConcreteReportBuilder();
const director: ReportDirector = new
ReportDirector(builder);
director.build();
const report: MyReport = builder.getResult();
console.log(report.getResult());
```

En estos ejemplos creamos la clase `Report` que representa el informe y tiene secciones para el encabezado, el cuerpo y el pie de página. Luego, definimos la interfaz `ReportBuilder` con métodos para construir cada sección del informe. La clase `ConcreteReportBuilder` implementa esta interfaz y construye cada sección del informe. El ReportDirector se encarga de utilizar el builder para construir el informe en un orden específico.

En el código principal, creamos una instancia del `ConcreteReportBuilder`, la pasamos al `ReportDirector` y construimos el informe. Finalmente, obtenemos el resultado del informe a través del builder y lo imprimimos. Al igual que en el ejemplo anterior, el patrón de diseño Builder separa la construcción de un objeto complejo de su representación y permite crear diferentes representaciones utilizando el mismo proceso de construcción.

De esta manera tenemos clases genéricas que nos permitirían crear cualquier tipo de reporte, estas clases toman elementos que están presentes en cualquier reporte sin importar su tipo, lo que debemos cambiar es el `ConcreteReportBuilder` de acuerdo a la lógica que necesitemos.

Prototype

El patrón de diseño Prototype es un patrón creacional que se utiliza para crear objetos duplicados (clones) de un objeto existente (prototipo) sin acoplar el código cliente a las clases concretas de esos objetos. Es decir, el patrón Prototype permite crear copias exactas de un objeto, lo que es útil cuando se desea crear nuevas instancias que comparten propiedades y comportamientos similares con una instancia original.

En lugar de utilizar constructores para crear objetos, el patrón Prototype utiliza un objeto prototipo como base para crear clones. Este enfoque es especialmente beneficioso cuando la creación de un objeto es costosa o compleja. El patrón Prototype define una interfaz común que debe ser implementada por las clases concretas de prototipos, y cada clase concreta proporciona su propia lógica de clonación.

En resumen, el patrón de diseño Prototype es útil cuando se necesita crear copias de objetos existentes de manera eficiente y flexible, permitiendo que las instancias nuevas compartan similitudes con el prototipo original mientras se mantienen independientes entre sí.

Veamos un ejemplo que nos ayude a entender mejor.

Supongamos que estamos construyendo un sistema de gestión de documentos y necesitamos crear copias exactas de documentos existentes.

Primero, definiremos una clase `Document` que tendrá una interfaz para clonar y métodos para establecer y obtener contenido. Luego, crearemos clases concretas que implementarán esta interfaz para diferentes tipos de documentos, como `TextDocument` e `ImageDocument`. Estas clases tendrán su propia implementación de clonación.

Primero el código en Java:

```java
// Interfaz del prototipo
interface Document {
    Document cloneDocument();
    void setContent(String content);
    String getContent();
}

// Clase concreta del prototipo para documentos de texto
class TextDocument implements Document {
    private String content;

    public Document cloneDocument() {
        TextDocument clonedDocument =
        new TextDocument();
        clonedDocument.setContent(this.content);
        return clonedDocument;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

```
// Clase concreta del prototipo para documentos de imagen
class ImageDocument implements Document {
    private String content;

    public Document cloneDocument() {
        ImageDocument clonedDocument =
        new ImageDocument();
        clonedDocument.setContent(this.content);
        return clonedDocument;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

public class Main {
    public static void main(String[] args) {
        /* Crear un documento de texto
        y establecer su contenido */
        Document textDocument = new TextDocument();
        textDocument.setContent(
        "Este es un documento de texto."
        );

        // Clonar el documento de texto
        Document clonedTextDocument =
        textDocument.cloneDocument();

        System.out.println("Contenido
        del documento original: " +
        textDocument.getContent()
        );
        System.out.println("Contenido
```

```
        del documento clonado: " +
        clonedTextDocument.getContent()
        );

        Document imageDocument =
        new ImageDocument();

        imageDocument.setContent(
        "Esto es una imagen."
        );

        // Clonar el documento de texto
        Document clonedImage =
        imageDocument.cloneDocument();

        System.out.println("Contenido
        de la imagen original: " +
        imageDocument.getContent()
        );


        System.out.println("Contenido
        de la imagen clonada: " +
        clonedImage.getContent()
        );
    }
}
```

Ahora el código en TypeScript:

```
// Interfaz del prototipo
interface MyDocument {
    cloneDocument(): Document;
    setContent(content: string): void;
    getContent(): string;
}

// Clase concreta del prototipo para documentos de texto
class TextDocument implements MyDocument {
    private content: string = "";

    cloneDocument(): MyDocument {
        const clonedDocument = new TextDocument();
        clonedDocument.setContent(this.content);
        return clonedDocument;
    }


    setContent(content: string): void {
        this.content = content;
    }


    getContent(): string {
        return this.content;
    }
}

// Clase concreta de prototipo para documentos de imagen
class ImageDocument implements MyDocument {
    private content: string = "";

    cloneDocument(): MyDocument {
        const clonedDocument = new ImageDocument();
        clonedDocument.setContent(this.content);
        return clonedDocument;
    }
```

```
    setContent(content: string): void {
        this.content = content;
    }

    getContent(): string {
        return this.content;
    }
}
// Código principal
const textDocument: MyDocument = new TextDocument();
textDocument.setContent("Este es un documento de
texto.");
const clonedTextDocument: MyDocument =
textDocument.cloneDocument();
console.log("Contenido del documento original:",
textDocument.getContent());
console.log("Contenido del documento clonado:",
clonedTextDocument.getContent());
```

Conclusión

Los patrones creacionales son un conjunto de patrones de diseño que se enfocan en la creación y configuración de objetos de manera eficiente, flexible y desacoplada. Estos patrones abordan los desafíos asociados con la instanciación de objetos y ofrecen soluciones elegantes para crear y administrar objetos en el diseño de software.

Ventajas:

- **Desacoplamiento:** Los patrones creacionales permiten que el código de cada patrón esté desacoplado de las clases concretas, lo que facilita los cambios en la creación de objetos sin afectar el resto del sistema.

- **Reutilización:** Los patrones creacionales promueven la reutilización de código al encapsular la lógica de creación en clases especializadas.
- **Abstracción:** Estos patrones fomentan la creación de interfaces y abstracciones que permiten trabajar con familias de objetos relacionados sin preocuparse por sus implementaciones concretas.
- **Control de Instancias:** Los patrones creacionales ofrecen control sobre la cantidad y el alcance de las instancias creadas, como en el caso del Singleton.

Desventajas:

- **Complejidad Adicional:** Al introducir capas adicionales de abstracción y estructuras, los patrones creacionales pueden aumentar la complejidad general del diseño.
- **Aumento de Código:** La implementación de patrones creacionales puede requerir la creación de más clases y estructuras, lo que a su vez puede aumentar la cantidad de código en el proyecto.
- **Mayor Tiempo de Desarrollo:** La adopción de patrones creacionales puede requerir un tiempo adicional para planificar y crear las clases adicionales y establecer las relaciones entre ellas.

La elección de un patrón creacional depende de las necesidades específicas del diseño y las características únicas del problema a resolver. Cada patrón aborda un conjunto particular de desafíos de creación de objetos, y es importante considerar las implicaciones a largo plazo en términos de mantenibilidad y flexibilidad.

En general, los patrones creacionales son herramientas poderosas para abordar problemas de creación de objetos en el diseño de software. Al utilizar estos patrones de manera efectiva, los

diseñadores pueden crear sistemas más flexibles, mantenibles y adaptativos al cambio.

Por tanto es importante tener en cuenta que aunque el tiempo de desarrollo puede aumentar, el tiempo en otras fases posterior al desarrollo de un sistema puede disminuir significativamente.

Patrones de estructura

Los patrones de estructura son soluciones reutilizables para organizar, relacionar y combinar componentes y clases en un programa de software. Estos patrones ayudan a establecer relaciones entre las partes del código de manera eficiente, lo que mejora la organización, la flexibilidad y la claridad del diseño. En esencia, los patrones de estructura proporcionan guías para construir la "estructura" general de un programa al definir cómo los componentes interactúan y se relacionan entre sí, sin necesidad de reinventar constantemente la rueda en cada proyecto.

El grupo de los GoF diseñó una serie de patrones que nos pueden ayudar de acuerdo a la necesidad por la que estemos atravesando en determinado momento, veamos.

Adapter

El patrón de diseño Adapter es un patrón estructural que permite que objetos con interfaces incompatibles trabajen juntos. Este patrón se utiliza para adaptar la interfaz de una clase existente (llamada "Adaptado") a otra interfaz esperada por el cliente, permitiendo que ambos colaboren sin necesidad de modificar el código fuente original.

En otras palabras, el patrón Adapter actúa como un intermediario que traduce las llamadas y las operaciones del cliente a una forma que el objeto Adaptado pueda entender y manejar. Esto es especialmente útil cuando se integran componentes heredados o de terceros en un nuevo sistema, donde no se pueden o no se desean cambiar las interfaces existentes.

El patrón Adapter se compone principalmente de tres componentes:

1. **Cliente:** Es el código que necesita interactuar con el Adaptado, pero su interfaz es incompatible con la del Adaptado.

2. **Adaptado:** Es la clase existente que necesita ser adaptada para trabajar con el Cliente. Su interfaz es la que el Cliente no puede utilizar directamente.

3. **Adaptador:** Es la clase que implementa la interfaz esperada por el Cliente y se comunica con el Adaptado para realizar las operaciones necesarias.

En resumen, el patrón Adapter permite que objetos con interfaces diferentes trabajen juntos sin que el Cliente ni el Adaptado sean conscientes de la adaptación que ocurre detrás de escena. Esto fomenta la reutilización de código y la integración de componentes de manera más flexible en el diseño de software.

Veamos un ejemplo donde dejemos claro cuál es cada uno de los componentes.

Imagina que tienes un sistema que interactúa con varios servicios de pago, pero deseas unificar la interfaz para todos los servicios de pago independientemente de su implementación subyacente.

Primero, definiremos una interfaz `PaymentService` que representa la interfaz deseada para todos los servicios de pago. Luego, crearemos una clase `CreditCardPayment` que es una implementación concreta de un servicio de pago específico. Después, crearemos una clase `PaymentAdapter` que adaptará la interfaz de `CreditCardPayment` a la interfaz `PaymentService`.

Veamos el código en Java:

```java
// Interfaz deseada para todos los servicios de pago
interface PaymentService {
    void processPayment(double amount);
}

// Implementación concreta de un servicio de pago
class CreditCardPayment {
    void makePayment(double amount) {
        System.out.println("Paid " + amount + "
        using credit card.");
    }
}

// Adaptador para la interfaz de PaymentService
class PaymentAdapter implements PaymentService {
    private CreditCardPayment creditCardPayment;

    public PaymentAdapter(CreditCardPayment
           creditCardPayment) {
        this.creditCardPayment = creditCardPayment;
    }

    public void processPayment(double amount) {
        creditCardPayment.makePayment(amount);
    }
}

public class Main {
    public static void main(String[] args) {
  // Utilizando el adaptador para realizar un pago
     CreditCardPayment creditCardPayment =
     new CreditCardPayment();
```

```
        PaymentService paymentService =
        new PaymentAdapter(creditCardPayment);

        paymentService.processPayment(100.00);
    }
}
```

Ahora el código en TypeScript:

```
// Interfaz deseada para todos los servicios de pago
interface PaymentService {
    processPayment(amount: number): void;
}

// Implementación concreta de un servicio de pago
class CreditCardPayment {
    makePayment(amount: number) {
        console.log(`Paid ${amount} using credit card.`);
    }
}

// Adaptador para la interfaz de PaymentService
class PaymentAdapter implements PaymentService {
    private creditCardPayment: CreditCardPayment;
    constructor(creditCardPayment:
                CreditCardPayment) {
        this.creditCardPayment = creditCardPayment;
    }
    processPayment(amount: number): void {
        this.creditCardPayment.makePayment(amount);
    }
}
// Código principal
const creditCardPayment = new CreditCardPayment();
const paymentService: PaymentService = new
PaymentAdapter(creditCardPayment);
paymentService.processPayment(100.00);
```

En este ejemplo, definimos la interfaz `PaymentService` que representa la interfaz deseada para todos los servicios de pago. La clase `CreditCardPayment` representa un servicio de pago específico con su propia implementación. Luego, creamos la clase `PaymentAdapter` que actúa como adaptador y permite que un objeto `CreditCardPayment` sea tratado como un `PaymentService`.

En el método main, creamos una instancia de `CreditCardPayment`, la envolvemos en el adaptador `PaymentAdapter` y luego utilizamos la interfaz `PaymentService` para realizar un pago. El patrón Adapter permite que objetos con interfaces incompatibles trabajen juntos, adaptando la interfaz de uno de ellos para que sea compatible con la otra. En este caso, el Adapter permite que la implementación específica de `CreditCardPayment` sea utilizada como si fuera un `PaymentService`.

¿Qué rol ocupa cada clase?

- **Cliente:** El cliente es el código principal que necesita realizar pagos utilizando la interfaz `PaymentService`. En el ejemplo, el cliente se representa en el método main de la clase `Main`, donde se crean instancias de los adaptadores y se utilizan para realizar pagos. El cliente es quien interactúa con la interfaz `PaymentService` para procesar pagos.

- **Adaptado:** En este caso, el adaptado es la clase `CreditCardPayment`. Esta es la clase existente que tiene una interfaz que es incompatible con la interfaz requerida por el cliente (`PaymentService`). El adaptado es el objeto con el que el cliente desea interactuar, pero necesita una interfaz diferente.

- **Adaptador:** El adaptador es la clase `CreditCardPaymentAdapter`. Esta clase actúa como intermediaria entre el cliente y el adaptado. El adaptador implementa la interfaz `PaymentService` que el cliente espera y se comunica con el adaptado (en este caso, `CreditCardPayment`) para realizar las operaciones necesarias. El adaptador "adapta" la interfaz del adaptado para que sea compatible con la interfaz requerida por el cliente.

En resumen, en este ejemplo:

El cliente es el código en el método main.
El adaptado es la clase `CreditCardPayment`.
El adaptador es `CreditCardPaymentAdapter.`

Bridge

El patrón Bridge es un patrón de diseño estructural que separa una abstracción de su implementación, de modo que ambas puedan variar de manera independiente. Esto significa que las clases abstractas (abstracciones) y las clases concretas (implementaciones) son independientes y pueden ser extendidas sin depender una de la otra.

En esencia, el patrón Bridge promueve la composición sobre la herencia, permitiendo que las abstracciones y las implementaciones evolucionen de manera independiente. Esto es especialmente útil cuando tienes múltiples dimensiones de variación en tu sistema y deseas evitar una explosión combinatoria de subclases.

En resumen, el patrón Bridge permite que una abstracción y su implementación varíen por separado, lo que facilita la creación de estructuras flexibles y mantenibles en un sistema de software.

Veamos un ejemplo del patrón Bridge, supongamos que estamos desarrollando un sistema de entretenimiento que puede mostrar películas y series de televisión en diferentes tipos de dispositivos: televisores y proyectores. Además, queremos admitir diferentes plataformas de contenido, como Netflix y Amazon Prime. El patrón Bridge nos ayudará a separar las abstracciones (contenido) de sus implementaciones (dispositivos y plataformas de contenido), lo que facilitará la extensión de ambas de manera independiente.

Veamos primero el ejemplo en Java.

Primero, definimos las interfaces para el contenido y las implementaciones:

```java
// Interfaz para el contenido
public interface Content {
    void play();
    void pause();
    void stop();
}

// Implementaciones concretas de contenido
public class Movie implements Content {
    private String title;

    public Movie(String title) {
        this.title = title;
    }

    public void play() {
        System.out.println("Reproduciendo la
        película: " + title);
    }

    public void pause() {
        System.out.println("Pausando la película: "
        + title);
```

```
    }

    public void stop() {
        System.out.println("Deteniendo la película:
        " + title);
    }
}

public class TVSeries implements Content {
    private String title;

    public TVSeries(String title) {
        this.title = title;
    }

    public void play() {
        System.out.println("Reproduciendo la serie
        de TV: " + title);
    }

    public void pause() {
        System.out.println("Pausando la serie de
        TV: " + title);
    }

    public void stop() {
        System.out.println("Deteniendo la serie de
        TV: " + title);
    }
}

// Interfaz para las implementaciones de dispositivos
public interface Device {
    void powerOn();
    void powerOff();
    void setChannel(int channel);
}
```

```java
// Implementaciones concretas de dispositivos
public class Television implements Device {
    public void powerOn() {
        System.out.println("Encendiendo el
        televisor");
    }

    public void powerOff() {
        System.out.println("Apagando el
        televisor");
    }

    public void setChannel(int channel) {
        System.out.println("Cambiando al canal " +
        channel);
    }
}

public class Projector implements Device {
    public void powerOn() {
        System.out.println("Encendiendo el
        proyector");
    }

    public void powerOff() {
        System.out.println("Apagando el
        proyector");
    }

    public void setChannel(int channel) {
        System.out.println("No se puede cambiar de
        canal en el proyector");
    }
}
```

Luego, creamos una abstracción `EntertainmentDevice` que utiliza tanto las implementaciones de contenido como las implementaciones de dispositivos:

```
/* Abstracción que une el contenido y los dispositivos*/
public abstract class EntertainmentDevice {
    protected Content content;
    protected Device device;

    public EntertainmentDevice(Content content,
        Device device) {
        this.content = content;
        this.device = device;
    }

    public abstract void playContent();
    public abstract void stopContent();
}
```

Ahora, implementamos las clases concretas que unen películas y series de TV con dispositivos específicos:

```
/* Implementación concreta que une películas con
dispositivos*/
public class MoviePlayer extends
    EntertainmentDevice {

    public MoviePlayer(Content content, Device
        device) {
        super(content, device);
    }
    public void playContent() {
        device.powerOn();
        content.play();
    }
```

```
    public void stopContent() {
        content.stop();
        device.powerOff();
    }
}

// Implementación concreta que une series de TV con
dispositivos
public class TVSeriesPlayer extends
    EntertainmentDevice {

    public TVSeriesPlayer(Content content,
        Device device) {
        super(content, device);
    }

    public void playContent() {
        device.powerOn();
        content.play();
    }

    public void stopContent() {
        content.stop();
        device.powerOff();
    }
}
```

Finalmente, en el código principal, podemos crear instancias de películas, dispositivos y reproductores y utilizarlos juntos:

```
public class Main {
    public static void main(String[] args) {
        Content movie = new Movie("The Matrix");
        Content tvSeries = new TVSeries("Breaking
        Bad");

        Device television = new Television();
        Device projector = new Projector();

        EntertainmentDevice moviePlayer = new
        MoviePlayer(movie, television);

        EntertainmentDevice tvSeriesPlayer = new
        TVSeriesPlayer(tvSeries, projector);

        moviePlayer.playContent();
        tvSeriesPlayer.playContent();

        moviePlayer.stopContent();
        tvSeriesPlayer.stopContent();
    }
}
```

Ahora veamos el ejemplo en TypeScript:

Primero, definimos las interfaces para el contenido y las implementaciones:

```
interface Content {
    play(): void;
    pause(): void;
    stop(): void;
}
// Implementaciones concretas de las formas
class Movie implements Content {
    private title: string;
```

```
    constructor(title: string) {
        this.title = title;
    }

    play(): void {
        console.log("Reproduciendo la película: " +
        this.title);
    }

    pause(): void {
        console.log("Pausando la película: " +
        this.title);
    }

    stop(): void {
        console.log("Deteniendo la película: " +
        this.title);
    }
}

class TVSeries implements Content {
    private title: string;


    constructor(title: string) {
        this.title = title;
    }

    play(): void {
        console.log("Reproduciendo la serie de TV:"
        + this.title);
    }

    pause(): void {
        console.log("Pausando la serie de TV: " +
        this.title);
    }
```

```
    stop(): void {
        console.log("Deteniendo la serie de TV: " +
        this.title);
    }
}

// Interfaz para las implementaciones de dispositivos
interface Device {
    powerOn(): void;
    powerOff(): void;
    setChannel(channel: number): void;
}
// Implementaciones concretas de dispositivos
class Television implements Device {
    powerOn(): void {
        console.log("Encendiendo el televisor");
    }

    powerOff(): void {
        console.log("Apagando el televisor");
    }

    setChannel(channel: number): void {
        console.log("Cambiando al canal " +
        channel);
    }
}

class Projector implements Device {
    powerOn(): void {
        console.log("Encendiendo el proyector");
    }

    powerOff(): void {
        console.log("Apagando el proyector");
    }
```

```
    setChannel(channel: number): void {
        console.log("No se puede cambiar de canal
        en el proyector");
    }
}
```

Luego, creamos una abstracción `EntertainmentDevice` que utiliza tanto las implementaciones de contenido como las implementaciones de dispositivos:

```
/* Abstracción que une el contenido y los dispositivos */
abstract class EntertainmentDevice {
    protected content: Content;
    protected device: Device;

    constructor(content: Content, device: Device) {
        this.content = content;
        this.device = device;
    }

    abstract playContent(): void;
    abstract stopContent(): void;
}
```

Ahora, implementamos las clases concretas que unen películas y series de TV con dispositivos específicos:

```
// Implementación concreta que une películas con
dispositivos
class MoviePlayer extends EntertainmentDevice {
    constructor(content: Content, device: Device) {
        super(content, device);
    }

    playContent(): void {
        this.device.powerOn();
```

```
        this.content.play();
    }

    stopContent(): void {
        this.content.stop();
        this.device.powerOff();
    }
}

// Implementación concreta que une series de TV con
dispositivos
class TVSeriesPlayer extends EntertainmentDevice {
    constructor(content: Content, device: Device) {
        super(content, device);
    }

    playContent(): void {
        this.device.powerOn();
        this.content.play();
    }

    stopContent(): void {
        this.content.stop();
        this.device.powerOff();
    }
}
```

Finalmente, en el código principal, podemos crear instancias de películas, dispositivos y reproductores y utilizarlos juntos:

```
const movie: Content = new Movie("The Matrix");
const tvSeries: Content = new TVSeries("Breaking Bad");

const television: Device = new Television();
const projector: Device = new Projector();

const moviePlayer: EntertainmentDevice = new
```

```
MoviePlayer(movie, television);
const tvSeriesPlayer: EntertainmentDevice = new
TVSeriesPlayer(tvSeries, projector);

moviePlayer.playContent();
tvSeriesPlayer.playContent();

moviePlayer.stopContent();
tvSeriesPlayer.stopContent();
```

En este ejemplo, el patrón Bridge separa las abstracciones (contenido) de las implementaciones (dispositivos), lo que permite combinar diferentes contenidos con diferentes dispositivos de manera flexible sin modificar el código existente. Cada reproductor (`MoviePlayer` y `TVSeriesPlayer`) combina un contenido específico con un dispositivo específico y puede reproducir y detener el contenido en el dispositivo correspondiente. Esto hace que el sistema sea más extensible y fácil de mantener a medida que se agregan nuevos contenidos o dispositivos.

Composite

El patrón Composite es un patrón de diseño estructural que permite componer objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite a los clientes tratar tanto a los objetos individuales como a las composiciones de objetos de manera uniforme.

En otras palabras, el patrón Composite se utiliza cuando tienes objetos que pueden ser componentes individuales o conjuntos de esos componentes. Los objetos individuales y los conjuntos se tratan de la misma manera, lo que permite construir estructuras jerárquicas complejas de manera eficiente y uniforme.

Este patrón es útil cuando necesitas representar estructuras jerárquicas como árboles, donde un nodo puede ser tanto una hoja (objeto individual) como un nodo interno (un conjunto de nodos). El patrón Composite facilita la creación y manipulación de estas estructuras, ya que los clientes no necesitan conocer la diferencia entre los objetos individuales y los compuestos.

Pasemos al ejemplo, supongamos que estamos construyendo una aplicación de manejo de documentos donde los documentos pueden contener tanto texto como elementos gráficos como imágenes.

Veamos primero el ejemplo en Java:

Primero, definimos la interfaz `Component` que representa tanto los elementos hoja (como el texto) como los elementos compuestos (como un documento que contiene texto e imágenes):

```java
// Interfaz Component que representa tanto elementos
simples como elementos compuestos
public interface Component {
    void render();
}
// Implementación concreta para elementos texto
public class Text implements Component {
    private String content;

    public Text(String content) {
        this.content = content;
    }

    public void render() {
        System.out.println("Texto: " + content);
    }
}
```

```
// Implementación concreta para elementos compuestos
public class Document implements Component {
    private List<Component> components =
    new ArrayList<>();

    public void addComponent(Component component) {
        components.add(component);
    }

    public void render() {
        System.out.println("Documento:");
        for (Component component : components) {
            component.render();
        }
    }
}
```

Ahora, podemos crear documentos que contienen texto y elementos gráficos:

```
public class Main {
    public static void main(String[] args) {
        // Crear elementos hoja (texto)
        Component heading =
        new Text("Título del documento");
        Component paragraph =
        new Text("Este es un párrafo de texto.");

        /* Crear un documento compuesto que
        contiene texto y elementos gráficos*/
        Document document = new Document();
        document.addComponent(heading);
        document.addComponent(paragraph);
        // Renderizar el documento
        document.render();
    }
}
```

Ahora veamos el ejemplo en TypeScript:

Primero, definimos la interfaz `Component` que representa tanto los elementos hoja (como el texto) como los elementos compuestos (como un documento que contiene texto e imágenes):

```
// Interfaz Component que representa tanto elementos hoja como elementos compuestos
interface Component {
    render(): void;
}

// Implementación concreta para elementos hoja (por ejemplo, texto)
class MyText implements Component {
    private content: string;

    constructor(content: string) {
        this.content = content;
    }

    render(): void {
        console.log("Texto: " + this.content);
    }
}
// Implementación concreta para elementos compuestos (por ejemplo, documentos)
class MyDocument implements Component {
    private components: Component[] = [];

    addComponent(component: Component): void {
        this.components.push(component);
    }

    render(): void {
        console.log("Documento:");
```

```
        for (const component of this.components) {
            component.render();
        }
    }
}
```

Ahora, podemos crear documentos que contienen texto y elementos gráficos:

```
// Crear elementos hoja (texto)
const heading: Component = new MyText("Título del
documento");
const paragraph: Component = new MyText("Este es un
párrafo de texto.");

// Crear un documento compuesto que contiene texto y
elementos gráficos
const document: MyDocument = new MyDocument();
document.addComponent(heading);
document.addComponent(paragraph);

// Renderizar el documento
document.render();
```

En ambos ejemplo, Text representa elementos simples y Document representa elementos compuestos (como documentos que pueden contener otros componentes). El método render se llama tanto en elementos texto como en elementos compuestos, lo que permite que puedan ser reutilizados de una manera sencilla, el crear un objeto global como `Component` deja la puerta abierta a la creación de componentes que podría necesitar un caso como el del ejemplo que realizamos.

Decorator

El patrón Decorator es un patrón de diseño estructural que permite agregar comportamiento adicional o responsabilidades a objetos individuales de manera dinámica, sin modificar su estructura. En otras palabras, el patrón Decorator se utiliza para extender la funcionalidad de una clase sin tener que crear subclases adicionales.

El patrón Decorator se basa en la composición en lugar de la herencia. Permite que los objetos sean decorados o envueltos por otros objetos que tienen la misma interfaz. Cada decorador agrega su propia funcionalidad al objeto original, y estos decoradores pueden apilarse en capas para proporcionar múltiples extensiones de comportamiento.

Este patrón es especialmente útil cuando tienes clases con un comportamiento base y deseas agregar opciones adicionales de manera flexible sin tener que crear una clase derivada para cada combinación posible. En resumen, el patrón Decorator permite la adición dinámica de responsabilidades a objetos, lo que lo hace versátil y fácil de extender.

Vamos con el ejemplo, Supongamos que estamos construyendo un sistema de cafetería donde tenemos diferentes tipos de café, y los clientes pueden personalizar su café agregando ingredientes adicionales como leche, caramelo, etc. Utilizaremos el patrón Decorator para modelar esto.

Primero, definimos una interfaz Coffee que representa la clase base de todos los tipos de café:

```java
// Interfaz que representa un café
public interface Coffee {
    double cost();
    String getDescription();
}
```

A continuación, creamos una implementación concreta de `Coffee` llamada `SimpleCoffee`:

```java
// Implementación concreta de un café simple
public class SimpleCoffee implements Coffee {
    public double cost() {
        return 2.0; // Un café simple cuesta $2.00
    }

    public String getDescription() {
        return "Café simple";
    }
}
```

Luego, creamos decoradores que extienden la funcionalidad de Coffee. Por ejemplo, aquí está un decorador para agregar leche al café:

```java
// Decorador para agregar leche al café
public class MilkDecorator implements Coffee {
    private final Coffee coffee;

    public MilkDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
```

```
    public double cost() {
       /* Agregamos el costo
       de la leche al costo base del café*/
      return coffee.cost() + 1.0; // La leche cuesta
      $1.00 extra
    }

    public String getDescription() {
        /*Agregamos "con leche" a
        la descripción del café*/
        return coffee.getDescription() + " con
        leche";
    }
}
```

A continuación, puedes crear instancias de cafés simples y decorarlos con ingredientes adicionales según el pedido del cliente:

```
public class Main {
    public static void main(String[] args) {
        // Pedimos un café simple
        Coffee simpleCoffee = new SimpleCoffee();
        System.out.println("Café simple cost: $" +
        simpleCoffee.cost());

        System.out.println("Description: " +
        simpleCoffee.getDescription());

        // Pedimos un café con leche
        Coffee coffeeWithMilk = new
        MilkDecorator(simpleCoffee);

        System.out.println("Café con leche cost:
        $" + coffeeWithMilk.cost());
```

```
        System.out.println("Description: " +
        coffeeWithMilk.getDescription());
    }
}
```

Antes de explicar el ejemplo, vamos a ver el código en TypeScript.

Primero, definimos una interfaz `Coffee` que representa la clase base de todos los tipos de café:

```
// Interfaz que representa un café
interface Coffee {
    cost(): number;
    getDescription(): string;
}
```

A continuación, creamos una implementación concreta de `Coffee` llamada `SimpleCoffee`:

```
// Implementación concreta de un café simple
class SimpleCoffee implements Coffee {
    cost(): number {
        return 2.0; // Un café simple cuesta $2.00
    }

    getDescription(): string {
        return "Café simple";
    }
}
```

Luego, creamos decoradores que extienden la funcionalidad de Coffee. Por ejemplo, aquí está un decorador para agregar leche al café:

```
// Decorador para agregar leche al café
class MilkDecorator implements Coffee {
    private coffee: Coffee;

    constructor(coffee: Coffee) {
        this.coffee = coffee;
    }

    cost(): number {
        /* Agregamos el costo de
        la leche al costo base del café*/
        return this.coffee.cost() + 1.0;
        // La leche cuesta $1.00 extra
    }


    getDescription(): string {
        /*Agregamos "con leche"
        a la descripción del café*/
        return this.coffee.getDescription() + " con
        leche";
    }
}
```

A continuación, puedes crear instancias de cafés simples y decorarlos con ingredientes adicionales según el pedido del cliente:

```
// Pedimos un café simple
const simpleCoffee: Coffee = new SimpleCoffee();
console.log("Café simple cost: $" + simpleCoffee.cost());
console.log("Description: " +
simpleCoffee.getDescription());

// Pedimos un café con leche
const coffeeWithMilk: Coffee = new
MilkDecorator(simpleCoffee);
```

```
console.log("Café con leche cost: $" +
coffeeWithMilk.cost());
console.log("Description: " +
coffeeWithMilk.getDescription());
```

En este ejemplo, hemos creado un café simple `(SimpleCoffee)` y luego lo hemos decorado con leche `(MilkDecorator)`. Cada decorador agrega su propio costo y descripción al café base. Esto demuestra cómo el patrón Decorator permite que los objetos sean flexibles y extensibles al agregar funcionalidad adicional sin modificar la estructura básica de la clase base, incluso en TypeScript.

Facade

El patrón Facade es un patrón de diseño estructural que proporciona una interfaz simplificada para un conjunto de interfaces más complejas en un subsistema. Actúa como un punto de entrada único para interactuar con el subsistema, ocultando la complejidad y detalles de implementación de las clases individuales que lo componen.

En otras palabras, el patrón Facade se utiliza para simplificar la interacción con un conjunto de clases o componentes que forman un subsistema más grande. Proporciona una capa de abstracción que facilita que los clientes utilicen el subsistema sin necesidad de conocer los detalles internos de cómo funciona.

El objetivo principal del patrón Facade es simplificar y desacoplar el código del cliente del código del subsistema, lo que mejora la mantenibilidad y la legibilidad del código. Además, proporciona un punto de entrada único y cohesivo para interactuar con el subsistema, lo que facilita el uso del subsistema en una aplicación más grande.

Continuemos con el ejemplo, supongamos que estamos desarrollando una aplicación de procesamiento de pedidos en línea para una tienda en línea. La aplicación tiene varios subsistemas y complejidades, como el manejo de inventario, el procesamiento de pagos y el envío de pedidos. El patrón Facade se utilizará para simplificar la interacción del cliente con estos subsistemas.

Primero veamos el ejemplo en Java.

Primero, definimos clases para los subsistemas de la tienda en línea, como el manejo de inventario, el procesamiento de pagos y el envío de pedidos:

```
import java.util.List;

// Subsistema de manejo de inventario
class InventorySystem {
   public void checkInventory(List<String> items) {
        /* Lógica para verificar el inventario de
        productos*/
        System.out.println("Verificando
        inventario...");
    }
}

// Subsistema de procesamiento de pagos
class PaymentSystem {
    public void processPayment(double amount) {
        // Lógica para procesar el pago
        System.out.println("Procesando el pago de
        $" + amount);
    }
}
```

```
class ShippingSystem {
    public void shipOrder(List<String> items) {
        // Lógica para enviar el pedido
        System.out.println("Enviando el
        pedido...");
    }
}
```

Luego, creamos una clase `OrderFacade` que actúa como una fachada para simplificar la creación y el procesamiento de pedidos:

```
class OrderFacade {
    private InventorySystem inventorySystem;
    private PaymentSystem paymentSystem;
    private ShippingSystem shippingSystem;

    public OrderFacade() {
        inventorySystem = new InventorySystem();
        paymentSystem = new PaymentSystem();
        shippingSystem = new ShippingSystem();
    }

    public void placeOrder(List<String> items,
    double amount) {
        // Verificar el inventario
        inventorySystem.checkInventory(items);

        // Procesar el pago
        paymentSystem.processPayment(amount);

        // Enviar el pedido
        shippingSystem.shipOrder(items);

        System.out.println("Pedido completado con
        éxito.");
    }
}
```

Ahora, en el código principal, los clientes pueden interactuar con la fachada `OrderFacade` en lugar de tratar directamente con los subsistemas:

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        OrderFacade orderFacade = new
        OrderFacade();
        List<String> items =
        Arrays.asList("Producto1", "Producto2");
        double amount = 100.0;

        orderFacade.placeOrder(items, amount);
    }
}
```

Antes de explicar el ejemplo, pasemos a TypeScript.

Primero, definimos clases para los subsistemas de la tienda en línea, como el manejo de inventario, el procesamiento de pagos y el envío de pedidos:

```
// Subsistema de manejo de inventario
class InventorySystem {
    checkInventory(items: string[]): void {
        /* Lógica para verificar
        el inventario de productos*/
        console.log("Verificando inventario...");
    }
}
```

```
// Subsistema de procesamiento de pagos
class PaymentSystem {
    processPayment(amount: number): void {
        // Lógica para procesar el pago
        console.log(`Procesando el pago de
        $${amount}`);
    }
}

// Subsistema de envío de pedidos
class ShippingSystem {
    shipOrder(items: string[]): void {
        // Lógica para enviar el pedido
        console.log("Enviando el pedido...");
    }
}
```

Luego, creamos una clase `OrderFacade` que actúa como una fachada para simplificar la creación y el procesamiento de pedidos:

```
class OrderFacade {
    private inventorySystem: InventorySystem;
    private paymentSystem: PaymentSystem;
    private shippingSystem: ShippingSystem;

    constructor() {
        this.inventorySystem = new
        InventorySystem();
        this.paymentSystem = new PaymentSystem();
        this.shippingSystem = new ShippingSystem();
    }

    placeOrder(items: string[], amount: number): void {
        // Verificar el inventario
        this.inventorySystem.checkInventory(items);

        // Procesar el pago
```

```
        this.paymentSystem.processPayment(amount);

        // Enviar el pedido
        this.shippingSystem.shipOrder(items);
        console.log("Pedido completado con
        éxito.");
    }
}
```

Ahora, en el código principal, los clientes pueden interactuar con la fachada `OrderFacade` en lugar de tratar directamente con los subsistemas:

```
const orderFacade = new OrderFacade();
const items = ["Producto1", "Producto2"];
const amount = 100.0;

orderFacade.placeOrder(items, amount);
```

En este ejemplo, el patrón Facade simplifica la interacción del cliente con los subsistemas (manejo de inventario, procesamiento de pagos y envío de pedidos) al proporcionar una interfaz unificada a través de la fachada `OrderFacade`. Los clientes solo necesitan llamar al método `placeOrder` de la fachada y no tienen que preocuparse por los detalles internos de los subsistemas.

Flyweight

Flyweight es un patrón de diseño estructural que se utiliza para minimizar el uso de memoria o recursos compartiendo de manera eficiente, instancias comunes y compartidas de objetos similares. En lugar de crear una instancia única para cada objeto similar, el patrón Flyweight permite compartir una sola instancia, lo que ahorra recursos y mejora el rendimiento.

Este patrón se basa en la idea de dividir un objeto en dos partes: la parte intrínseca, que contiene el estado compartido y no cambia entre objetos, y la parte extrínseca, que contiene el estado específico y puede variar entre objetos. Al compartir la parte intrínseca, múltiples objetos pueden referenciarla, lo que reduce significativamente la cantidad de memoria utilizada.

El patrón Flyweight es especialmente útil cuando se trabaja con una gran cantidad de objetos similares y se busca optimizar la eficiencia de memoria y rendimiento de una aplicación.

Vamos con el ejemplo, primero en Java:

Un ejemplo del patrón Flyweight en Java podría ser en el contexto de una aplicación de procesamiento de texto, donde se utilizan muchos caracteres individuales, pero queremos ahorrar memoria almacenando solo una instancia de cada carácter.

Primero, definimos una interfaz `TextCharacter` que representa los caracteres de texto individuales:

```
// Interfaz para representar caracteres de texto
public interface TextCharacter {
    void printCharacter();
}
```

A continuación, implementamos clases concretas para los caracteres, que pueden ser compartidos:

```
// Clase concreta para caracteres de texto
public class ConcreteTextCharacter implements
    TextCharacter {
    private char character;
    public ConcreteTextCharacter(char character) {
        this.character = character;
```

```
    }

    public void printCharacter() {
        System.out.print(character);
    }
}
```

Ahora, creamos una fábrica `(CharacterFactory)` que administra y almacena instancias compartidas de caracteres:

```
import java.util.HashMap;
import java.util.Map;
// Fábrica para administrar instancias compartidas de
caracteres
class CharacterFactory {
    private Map<Character, TextCharacter>
    characterMap = new HashMap<>();

    public TextCharacter getCharacter(char
    character) {
        if (!characterMap.containsKey(character)) {
            characterMap.put(character, new
            ConcreteTextCharacter(character));
        }
        return characterMap.get(character);
    }
}
```

Finalmente, en el código principal, podemos utilizar la fábrica para obtener instancias de caracteres y imprimir texto:

```
public class Main {
    public static void main(String[] args) {
        CharacterFactory characterFactory = new
        CharacterFactory();

        String text = "Hello, World!";
        for (char c : text.toCharArray()) {
            TextCharacter character =
            characterFactory.getCharacter(c);
            character.printCharacter();
        }
    }
}
```

Ahora, pasemos al ejemplo en TypeScript:

Primero, definimos una interfaz `TextCharacter` que representa los caracteres de texto individuales:

```
// Interfaz para representar caracteres de texto
interface TextCharacter {
    printCharacter(): void;
}
```

A continuación, implementamos clases concretas para los caracteres, que pueden ser compartidos:

```
// Clase concreta para caracteres de texto
class ConcreteTextCharacter implements TextCharacter {
    private character: string;

    constructor(character: string) {
        this.character = character;
    }

    printCharacter(): void {
```

```
        process.stdout.write(this.character);
    }
}
```

Ahora, creamos una fábrica (`CharacterFactory`) que administra y almacena instancias compartidas de caracteres:

```
// Fábrica para administrar instancias compartidas de
caracteres
class CharacterFactory {
    private characterMap: { [key: string]:
    TextCharacter } = {};

    getCharacter(character: string): TextCharacter
    {
        if (!this.characterMap[character]) {
            this.characterMap[character] = new
            ConcreteTextCharacter(character);
        }
        return this.characterMap[character];
    }
}
```

Finalmente, en el código principal, podemos utilizar la fábrica para obtener instancias de caracteres y imprimir texto:

```
const characterFactory = new CharacterFactory();

const text = "Hello, World!";
for (const char of text) {
    const character =
characterFactory.getCharacter(char);
    character.printCharacter();
}
```

En este ejemplo, el patrón Flyweight se aplica al almacenar y compartir instancias de caracteres de texto individuales (`TextCharacter`). En lugar de crear una nueva instancia para cada carácter en el texto, utilizamos la fábrica `CharacterFactory` para obtener instancias compartidas siempre que sea posible, lo que ahorra memoria.

Este patrón es útil en situaciones en las que hay una gran cantidad de objetos similares que pueden compartir parte de su estado, lo que resulta en un uso eficiente de la memoria. En aplicaciones del mundo real, puede aplicarse en representación de fuentes de texto, píxeles de imágenes, objetos geométricos y más, como en el ejemplo anterior.

Proxy

El patrón Proxy es un patrón de diseño estructural que proporciona un sustituto o representante de otro objeto para controlar el acceso a él. El proxy actúa como una capa intermedia entre el cliente y el objeto real, permitiendo realizar tareas adicionales, como la carga bajo demanda, la verificación de permisos, el registro de acceso, entre otros, sin que el cliente sea consciente de ello.

En resumen, el patrón Proxy se utiliza para agregar una capa de control y funcionalidad adicional a un objeto sin modificar su estructura ni afectar su uso normal. Esto es particularmente útil en situaciones en las que se necesita un control adicional sobre el acceso o el comportamiento de un objeto, como la gestión de recursos costosos, la seguridad, la auditoría o la carga bajo demanda de recursos. El proxy permite al cliente interactuar con el objeto de la misma manera que lo haría con el objeto real, pero con funcionalidades adicionales proporcionadas por el proxy.

Pasemos al ejemplo, supongamos que estamos construyendo una aplicación de gestión de documentos en una empresa, y

necesitamos implementar un sistema que permita a los empleados acceder a documentos almacenados en un servidor remoto de manera eficiente. Cada documento puede ser bastante grande, y no queremos descargarlos todos de inmediato debido a la limitación de ancho de banda y el espacio de almacenamiento en las máquinas locales de los empleados. En cambio, deseamos descargar documentos sólo cuando un empleado solicita un documento en particular.

Veamos el código en Java.

Primero definimos la interfaz `File` que representa un archivo en un servidor remoto:

```
// Interfaz para representar un archivo en un servidor remoto
public interface File {
    void download();
}
```

A continuación, implementamos una clase `RealFile` que representa un documento real en el servidor:

```
// Implementación concreta de un documento real en el servidor
public class RealFile implements File {
    private String filename;
    private boolean downloaded = false;

    public RealFile(String filename) {
        this.filename = filename;
    }
    public void download() {
        if (!downloaded) {
            System.out.println("Descargando archivo
            " + filename + " desde el servidor
```

```
            remoto.");
            downloaded = true;
        } else {
            System.out.println("El archivo " +
            filename + " ya ha sido descargado.");
        }
    }
}
```

A continuación, creamos una clase FileProxy que actúa como un proxy para los archivos. El proxy verifica si el archivo real ya ha sido descargado antes de permitir la descarga:

```
// Proxy que controla el acceso a los documentos y
descarga bajo demanda
public class FileProxy implements File {
    private RealFile realFile;
    private String filename;

    public FileProxy(String filename) {
        this.filename = filename;
    }

    public void download() {
        if (realFile == null) {
            realFile = new RealFile(filename);
        }
        realFile.download();
    }
}
```

En el código principal, podemos usar el proxy FileProxy para descargar archivos. Si el archivo aún no se ha descargado, el proxy se encarga de descargarlo; de lo contrario, reutiliza el archivo ya descargado:

```
public class Main {
    public static void main(String[] args) {
        File file1 = new
        FileProxy("archivo1.txt");
        File file2 = new
        FileProxy("archivo2.txt");

        /* Descargamos el primer archivo (se
        descargará desde el servidor)*/
        file1.download();
        file1.download();

        /* Descargamos el segundo
        archivo (ya se descargó antes)*/
        file2.download();
        file2.download();
    }
}
```

Ahora vemos el ejemplo con TypeScript.

Primero, definimos una interfaz File que representa un documento en el servidor remoto:

```
// Interfaz para representar un documento en el servidor
remoto
interface MyFile {
    download(): void;
}
```

A continuación, implementamos una clase `RealFile` que representa un documento real en el servidor:

```
// Implementación concreta de un documento real en el
servidor
class RealFile implements MyFile {
    private filename: string;
    private downloaded: boolean = false;

    constructor(filename: string) {
        this.filename = filename;
    }

    download(): void {
        if (!this.downloaded) {
            console.log(`Descargando archivo
            ${this.filename} desde el servidor
            remoto.`);
            this.downloaded = true;
        } else {
            console.log(`El archivo
            ${this.filename} ya ha sido
            descargado.`);
        }
    }
}
```

La clase `FileProxy` actúa como un proxy para los documentos y verifica si un documento ya ha sido descargado antes de permitir la descarga:

```
// Proxy que controla el acceso a los documentos y
descarga bajo demanda
class FileProxy implements MyFile {
    private realFile: RealFile | null = null;
    private filename: string;
    constructor(filename: string) {
        this.filename = filename;
    }
```

```
    download(): void {
        if (!this.realFile) {
            this.realFile = new
            RealFile(this.filename);
        }
        this.realFile.download();
    }
}
```

En el código principal, podemos utilizar el proxy `FileProxy` para descargar documentos. El proxy verifica si el documento ya se ha descargado, y si no, se encarga de la descarga; de lo contrario, reutiliza el documento ya descargado:

```
const file1: File = new FileProxy("documento1.pdf");
const file2: File = new FileProxy("documento2.doc");
// Descargamos el primer documento (se descargará desde
el servidor)
file1.download();
file1.download();
// Descargamos el segundo documento (ya se descargó
antes)
file2.download();
file2.download();
```

En este ejemplo, el patrón `Proxy` se utiliza para controlar el acceso a los archivos en el servidor remoto. Cuando se descarga un archivo a través del proxy, el proxy verifica si el archivo ya ha sido descargado. Si no, se descarga desde el servidor remoto; de lo contrario, se reutiliza la versión ya descargada. Esto puede ser útil en situaciones en las que deseas cargar recursos costosos bajo demanda y evitar descargas innecesarias.

Conclusión

Hemos explorado una variedad de patrones de diseño de estructura que son esenciales para la creación de software sólido y mantenible. Cada uno de estos patrones tiene un propósito y una utilidad específica en la organización de componentes y clases en un sistema, lo que mejora su flexibilidad y extensibilidad. Aquí hay una recapitulación de algunas ventajas y desventajas clave de trabajar con patrones de diseño de estructura.

Ventajas:

- **Organización y Claridad:** Los patrones de estructura ayudan a organizar las clases y componentes en el código, lo que facilita la comprensión y el mantenimiento.

- **Reutilización de Código:** Facilitan la reutilización de código al encapsular comportamientos comunes y componentes en clases separadas, lo que reduce la duplicación de código.

- **Flexibilidad:** Los patrones de estructura permiten que el código sea más flexible y adaptable a cambios futuros, ya que se pueden agregar, quitar o modificar componentes sin afectar otras partes del sistema.

- **Escalabilidad:** Ayudan a crear sistemas escalables, lo que significa que se pueden agregar nuevos componentes o funcionalidades sin afectar negativamente el rendimiento o la estabilidad.

Desventajas:

- **Complejidad Adicional:** A veces, la implementación de patrones de estructura puede introducir una capa adicional de complejidad en el código, lo que puede dificultar la comprensión para desarrolladores menos experimentados.

- **Sobrecarga de Abstracción:** En algunos casos, la introducción excesiva de abstracciones y patrones puede aumentar la sobrecarga y dificultar el rendimiento del sistema.

- **Tiempo de Desarrollo:** Implementar patrones de diseño de estructura puede requerir más tiempo de desarrollo inicial en comparación con enfoques más simples.

- **Sobrediseño:** Existe el riesgo de sobrediseñar un sistema, aplicando patrones innecesarios que complican en lugar de simplificar.

En conclusión los patrones de diseño de estructura son herramientas valiosas para los desarrolladores, pero deben aplicarse con discernimiento. La elección de un patrón de estructura debe basarse en las necesidades del proyecto y en un equilibrio entre claridad y flexibilidad. Como desarrollador, la comprensión de estos patrones te brinda una valiosa caja de herramientas para abordar desafíos de diseño en tus proyectos de software.

Patrones de Comportamiento

Los patrones de comportamiento son un conjunto de patrones de diseño en ingeniería de software que se enfocan en cómo los objetos y clases interactúan y se comunican entre sí para lograr un comportamiento coherente y flexible en una aplicación. Estos patrones proporcionan soluciones probadas y comunes para problemas relacionados con la gestión de algoritmos, la coordinación de objetos, la captura de cambios de estado y la definición de flujos de trabajo.

Chain of Responsibility

El patrón Chain of Responsibility es un patrón de diseño de comportamiento que se utiliza para crear una cadena de objetos (manejadores) en la que cada objeto en la cadena tiene la capacidad de procesar una solicitud o pasarla al siguiente objeto en la cadena.

En otras palabras, cuando llega una solicitud, cada objeto en la cadena decide si puede manejarla o si debe pasar al siguiente objeto en la cadena. Esto permite que múltiples objetos tengan la oportunidad de procesar la solicitud, y el proceso de manejo puede ser flexible y escalable. Si un objeto en la cadena puede manejar la solicitud, lo hace; de lo contrario, la pasa al siguiente objeto.

El patrón Chain of Responsibility se utiliza comúnmente en situaciones donde se requiere un procesamiento secuencial de solicitudes y donde se necesita desacoplar el remitente de la solicitud de sus receptores. Cada manejador en la cadena puede tener reglas específicas para procesar la solicitud y decidir si la maneja o la delega. Este patrón promueve la reutilización y permite

agregar o eliminar fácilmente manejadores sin afectar el flujo general de procesamiento de solicitudes.

Vamos al ejemplo, supongamos que tienes una serie de empleados en una empresa, y cada empleado tiene un límite de autorización para aprobar gastos. La solicitud de compra debe pasar por una cadena de empleados hasta que alguien con la autoridad adecuada la apruebe.

Veamos el ejemplo en Java.

Primero, definimos una interfaz `Handler` que representa a los manejadores en la cadena:

```
/* Interfaz Handler que define el manejo de
solicitudes*/
public interface Handler {
    void setNextHandler(Handler nextHandler);
    void processRequest(PurchaseRequest request);
}
```

Luego, implementamos varias clases concretas que representan diferentes niveles de autoridad:

```java
// Clase concreta que maneja solicitudes de bajo valor
public class JuniorManager implements Handler {
    private Handler nextHandler;

    public void setNextHandler(Handler nextHandler)
    {
        this.nextHandler = nextHandler;
    }

    public void processRequest(PurchaseRequest
    request) {
        if (request.getAmount() <= 1000) {
            System.out.println("Solicitud de compra
            aprobada por el Junior Manager.");
        } else if (nextHandler != null) {
            nextHandler.processRequest(request);
        } else {
            System.out.println("La solicitud de
            compra requiere aprobación
            adicional.");
        }
    }
}
```

```java
/*Clase concreta que maneja solicitudes de valor medio*/
public class SeniorManager implements Handler {
    private Handler nextHandler;

    public void setNextHandler(Handler nextHandler)
    {
        this.nextHandler = nextHandler;
    }

    public void processRequest(PurchaseRequest
    request) {
        if (request.getAmount() <= 5000) {
            System.out.println("Solicitud de compra
            aprobada por el Senior Manager.");
        } else if (nextHandler != null) {
            nextHandler.processRequest(request);
        } else {
            System.out.println("La solicitud de
            compra requiere aprobación
            adicional.");
        }
    }
}
```

```java
// Clase que representa una solicitud de compra
public class PurchaseRequest {
    private double amount;

    public PurchaseRequest(double amount) {
        this.amount = amount;
    }
    public double getAmount() {
        return amount;
    }
}
```

```
// Clase concreta que maneja solicitudes de alto valor
public class Director implements Handler {
    public void processRequest(PurchaseRequest
    request) {
        if (request.getAmount() <= 10000) {
            System.out.println("Solicitud de compra
            aprobada por el Director.");
        } else {
            System.out.println("La solicitud de
            compra fue rechazada.");
        }
    }
    public void setNextHandler(Handler nextHandler)
    {
        /* El Director es el último en la cadena,
           no se establece un próximo manejador.*/
    }
}
```

En el código principal, configuramos la cadena de responsabilidad y procesamos una solicitud de compra:

```
public class Main {
    public static void main(String[] args) {
        Handler jrManager = new JuniorManager();
        Handler seniorManager = new
        SeniorManager();
        Handler director = new Director();
        jrManager.setNextHandler(seniorManager);
        seniorManager.setNextHandler(director);

        // Simulamos una solicitud de compra
        PurchaseRequest request1 = new
        PurchaseRequest(2800);
```

```
        // Procesamos las solicitudes
        jrManager.processRequest(request1);

    }
}
```

Ahora vamos al ejemplo en TypeScript:

Primero, definimos una interfaz Handler que representa a los manejadores en la cadena:

```
/* Interfaz Handler que define el manejo de solicitudes
*/
interface Handler {
    setNextHandler(nextHandler: Handler): void;
    processRequest(request: PurchaseRequest): void;
}
```

Luego, implementamos varias clases concretas que representan diferentes niveles de autoridad:

```
// Clase concreta que maneja solicitudes de bajo valor
class JuniorManager implements Handler {
    private nextHandler: Handler | null = null;

    setNextHandler(nextHandler: Handler): void {
        this.nextHandler = nextHandler;
    }

    processRequest(request: PurchaseRequest): void
    {
        if (request.getAmount() <= 1000) {
            console.log("Solicitud de compra
            aprobada por el Junior Manager.");
        } else if (this.nextHandler) {
```

```
            this.nextHandler.processRequest(request);
        } else {
            console.log("La solicitud de compra
            requiere aprobación adicional.");
        }
    }
}
```

```
/* Clase concreta que maneja solicitudes de valor medio */
class SeniorManager implements Handler {
    private nextHandler: Handler | null = null;

    setNextHandler(nextHandler: Handler): void {
        this.nextHandler = nextHandler;
    }

    processRequest(request: PurchaseRequest): void
    {
        if (request.getAmount() <= 5000) {
            console.log("Solicitud de compra
            aprobada por el Senior Manager.");
        } else if (this.nextHandler) {
            this.nextHandler.processRequest(request);
        } else {
            console.log("La solicitud de compra
            requiere aprobación adicional.");
        }
    }
}
```

```
// Clase concreta que maneja solicitudes de alto valor
class Director implements Handler {
    setNextHandler(nextHandler: Handler): void {
        /* El Director es el último en la cadena,
        no se establece un próximo manejador.*/
    }

    processRequest(request: PurchaseRequest): void
    {
        if (request.getAmount() <= 10000) {
            console.log("Solicitud de compra
            aprobada por el Director.");
        } else {
            console.log("La solicitud de compra fue
            rechazada.");
        }
    }
}
```

A continuación, definimos una clase `PurchaseRequest` que representa una solicitud de compra:

```
// Clase que representa una solicitud de compra
class PurchaseRequest {
    constructor(private amount: number) {}

    getAmount(): number {
        return this.amount;
    }
}
```

Finalmente, en el código principal, configuramos la cadena de responsabilidad y procesamos las solicitudes de compra:

```
const juniorManager = new JuniorManager();
const seniorManager = new SeniorManager();
```

```
const director = new Director();

juniorManager.setNextHandler(seniorManager);
seniorManager.setNextHandler(director);

// Simulamos solicitudes de compra
const request1 = new PurchaseRequest(800);
const request2 = new PurchaseRequest(4500);
const request3 = new PurchaseRequest(12000);

// Procesamos las solicitudes
juniorManager.processRequest(request1);
juniorManager.processRequest(request2);
juniorManager.processRequest(request3);
```

En este ejemplo, las solicitudes de compra pasan a través de la cadena de responsabilidad comenzando desde el `JuniorManager`, pasando al `SeniorManager` y finalmente al `Director`. Cada manejador decide si aprueba la solicitud o la pasa al siguiente nivel en la cadena. Si ningún manejador puede aprobar la solicitud, se informa que se requiere aprobación adicional.

Command

El patrón Command es un patrón de diseño de comportamiento que se utiliza para encapsular una solicitud como un objeto. Esto permite que los clientes emitan solicitudes sin conocer los detalles de la operación que se realizará ni quién la llevará a cabo.

Es decir el patrón Command convierte una solicitud o request en un objeto independiente que contiene toda la información necesaria para ejecutar esa solicitud en un momento posterior. Esto promueve la separación de la invocación de una acción de la implementación real de esa acción, lo que resulta en un diseño más flexible y extensible. Este patrón es especialmente útil cuando se necesita la

capacidad de deshacer o rehacer acciones, o cuando se requiere el registro de solicitudes para su posterior ejecución o manipulación.

Hablando del ejemplo, supongamos que tienes un control remoto que debe manejar diferentes dispositivos electrónicos, como una televisión y una luz. Cada dispositivo tiene operaciones específicas, como encender, apagar y ajustar el volumen (en el caso de la televisión). Veamos cómo sería el código en Java para lograr esto utilizando el patrón Command.

Primero definimos la interfaz base para los comandos.

```
// Interfaz Command que define las operaciones comunes
para los comandos
interface Command {
    void execute();
}
```

Ahora la clase base para los dispositivos:

```
public class Device {

     private String name;

     public Device(String name) {
          this.name = name;
     }

     public void turnOn() {
          System.out.println("Encendiendo " + name);
     }
     public void turnOff() {
          System.out.println("Apagando " + name);
     }
}
```

Ahora realizamos las implementaciones, de acuerdo a los comandos que necesitamos ejecutar:

```
// Clase concreta que representa un comando para encender un dispositivo
public class TurnOnCommand implements Command {
    private Device device;
    public TurnOnCommand(Device device) {
        this.device = device;
    }
    public void execute() {
        device.turnOn();
    }
}
```

```
// Clase concreta que representa un comando para apagar un dispositivo
public class TurnOffCommand implements Command {
    private Device device;

    public TurnOffCommand(Device device) {
        this.device = device;
    }

    public void execute() {
        device.turnOff();
    }
}
```

```
// Clase concreta que representa un comando para ajustar el volumen de la televisión
public class AdjustVolumeCommand implements Command {
    private Television television;
    private int volume;
    public AdjustVolumeCommand(Television
    television, int volume) {
```

```
        this.television = television;
        this.volume = volume;
    }
    public void execute() {
        television.adjustVolume(volume);
    }
}
```

Ahora definimos las clases sobre las cuales serán ejecutados los comandos, en este caso los dispositivos:

```
// Clase que representa una televisión (receptor
específico)
public class Television extends Device {
    public Television(String name) {
        super(name);
    }

    public void adjustVolume(int volume) {
        System.out.println("Ajustando el volumen de
        la televisión a " + volume);
    }
}
```

```
// Clase que representa el control remoto (invocador)
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Ahora creamos la clase principal donde todo esto interactúe:

```
public class Main {
    public static void main(String[] args) {
        // Crear dispositivos
        var tv = new Television("Televisión");
        var light = new Device("Luz");

        // Crear comandos
        Command turnOnTV = new TurnOnCommand(tv);
        Command turnOffTV = new TurnOffCommand(tv);
        Command adjustVolume = new
        AdjustVolumeCommand(tv, 20);
        Command turnOnLight = new
        TurnOnCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.setCommand(turnOnTV);

        // Presionar el botón del control remoto
        remote.pressButton();

        // Usar otro comando
        remote.setCommand(adjustVolume);
        remote.pressButton();

        // Cambiar el comando del control remoto
        remote.setCommand(turnOffTV);
        remote.pressButton();


        // Controlar otro dispositivo
        remote.setCommand(turnOnLight);
        remote.pressButton();
    }
}
```

Ahora veamos el código en TypeScript y después pasamos a la explicación:

Primero definimos la interfaz base para los comandos:

```
// Interfaz Command que define las operaciones comunes
para los comandos
interface Command {
    execute(): void;
}
```

También definimos la clase base para los dispositivos:

```
class Device {
    private name: string;

    constructor(name: string) {
        this.name = name;
    }

    turnOn(): void {
        console.log("Encendiendo " + this.name);
    }

    turnOff(): void {
        console.log("Apagando " + this.name);
    }
}
```

Ahora realizamos las implementaciones, de acuerdo a los comandos que necesitamos ejecutar:

```
// Clase concreta que representa un comando para encender un dispositivo
class TurnOnCommand implements Command {
    private device: Device;

    constructor(device: Device) {
        this.device = device;
    }

    execute(): void {
        this.device.turnOn();
    }
}
```

```
// Clase concreta que representa un comando para apagar un dispositivo
class TurnOffCommand implements Command {
    private device: Device;

    constructor(device: Device) {
        this.device = device;
    }

    execute(): void {
        this.device.turnOff();
    }
}
```

```
// Clase concreta que representa un comando para ajustar el volumen de la televisión
class AdjustVolumeCommand implements Command {
    private television: Television;
    private volume: number;
```

```
    constructor(television: Television, volume:
    number) {
        this.television = television;
        this.volume = volume;
    }

    execute(): void {
        this.television.adjustVolume(this.volume);
    }
}
```

```
// Clase que representa una televisión (receptor
específico)
class Television extends Device {
    constructor(name: string) {
        super(name);
    }

    adjustVolume(volume: number): void {
        console.log("Ajustando el volumen de la
        televisión a " + volume);
    }
}
```

Ahora creamos la clase que se encarga de ejecutar los comandos a través de los dispositivos:

```
// Clase que representa el control remoto (invocador)
class RemoteControl {
    private command: Command | null = null;

    setCommand(command: Command): void {
        this.command = command;
    }

    pressButton(): void {
        if (this.command) {
            this.command.execute();
        }
    }
}
```

Por último creamos un fragmento de código que interactúa con todas las clases:

```
const tv = new Television("Televisión");
const light = new Device("Luz");

const turnOnTV = new TurnOnCommand(tv);
const turnOffTV = new TurnOffCommand(tv);
const adjustVolume = new AdjustVolumeCommand(tv, 20);
const turnOnLight = new TurnOnCommand(light);

const remote = new RemoteControl();
remote.setCommand(turnOnTV);
remote.pressButton();

remote.setCommand(adjustVolume);
remote.pressButton();

remote.setCommand(turnOffTV);
remote.pressButton();
```

```
remote.setCommand(turnOnLight);
remote.pressButton();
```

Expliquemos el código que acabamos de crear:

- Interfaz `Command`: Definimos una interfaz llamada `Command` que declara un método `execute()`. Esta interfaz representa todos los comandos posibles que pueden ser ejecutados. Cualquier comando concreto debe implementar esta interfaz y proporcionar su propia implementación del método `execute()`.

- Comandos concretos: Creamos clases concretas que implementan la interfaz `Command` para representar comandos específicos. Por ejemplo, `TurnOnCommand` y `TurnOffCommand` son comandos que encienden y apagan un dispositivo respectivamente. `AdjustVolumeCommand` es un comando que ajusta el volumen de una televisión a un valor específico.

- Dispositivos (receptores): Creamos una clase llamada Device que representa un dispositivo electrónico genérico. Esta clase contiene métodos para encender y apagar el dispositivo. `Television` es una subclase de `Device` que también tiene un método para ajustar el volumen. Los comandos actuarán sobre estos dispositivos.

- Control remoto (invocador): La clase `RemoteControl` actúa como el control remoto que puede configurarse con un comando y ejecutar ese comando. Cuando el método `pressButton()` del control remoto se llama, se ejecuta el comando actual. Esto permite que el control remoto ejecute comandos sin conocer los detalles de cómo funcionan o qué dispositivos controlan.

- Ejecución del ejemplo: En ambos casos (TypeScript y Java), creamos instancias de dispositivos y comandos concretos. Luego, configuramos el control remoto con diferentes comandos y presionamos el botón del control remoto. Cada vez que presionamos el botón, el control remoto ejecuta el comando configurado en ese momento, lo que resulta en acciones como encender, apagar o ajustar el volumen de la televisión.

En resumen, el patrón `Command` encapsula comandos como objetos, permitiendo que las solicitudes sean independientes de los receptores y de cómo se ejecutan las operaciones. Esto proporciona flexibilidad, extensibilidad y la capacidad de realizar acciones como deshacer o rehacer fácilmente.

Interpreter

El patrón Interpreter es un patrón de diseño de comportamiento que se utiliza para definir una gramática para un lenguaje y proporcionar un intérprete que interpreta sentencias escritas en ese lenguaje. Este patrón se usa para convertir una expresión dada en un objeto que puede ser evaluado y produce un resultado.

- **Gramática:** Define las reglas y la estructura del lenguaje que se desea interpretar. Puede ser una gramática formal o una estructura más flexible.

- **Terminal y No Terminal:** En el contexto del patrón Interpreter, los elementos de la gramática se dividen en "terminales" (expresiones que no pueden ser divididas en partes más pequeñas) y "no terminales" (expresiones que pueden dividirse en partes más pequeñas).

- **Expresión:** Cada regla gramatical se representa como una clase o interfaz de expresión. Las clases concretas implementan estas expresiones.

- **Contexto:** Proporciona información de contexto que puede ser necesaria para la interpretación de las expresiones.

El patrón Interpreter es útil cuando se necesita evaluar expresiones complejas definidas por una gramática y es particularmente útil en lenguajes de consulta, sistemas de procesamiento de lenguaje natural, análisis sintáctico y otros escenarios donde se requiere interpretar o evaluar estructuras de datos complejas basadas en reglas específicas.

En resumen, el patrón Interpreter facilita la interpretación de lenguajes y gramáticas mediante la representación de reglas gramaticales como objetos y proporcionando un mecanismo para evaluar sentencias basadas en esas reglas.

En este ejemplo, crearemos un intérprete que puede evaluar expresiones aritméticas básicas, como sumas y restas. La gramática será bastante limitada, pero ilustrará cómo se puede implementar el patrón Interpreter.

Primero, definiremos una interfaz Expression que representará todas las expresiones que pueden ser interpretadas:

```
public interface Expression {
    int interpret();
}
```

Luego, implementaremos clases concretas para representar diferentes tipos de expresiones, como números, sumas y restas:

```java
// Clase concreta que representa un número
public class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    public int interpret() {
        return number;
    }
}
```

```java
// Clase concreta que representa una suma
public class AddExpression implements Expression {
    private Expression left;
    private Expression right;
    public AddExpression(Expression left,
    Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret() {
        return left.interpret() +
        right.interpret();
    }
}
```

```java
// Clase concreta que representa una resta
public class SubtractExpression implements
Expression {
    private Expression left;
    private Expression right;
```

```
    public SubtractExpression(Expression left,
    Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret() {
        return left.interpret() -
        right.interpret();
    }
}
```

Ahora, creemos un cliente que utilice estas expresiones para evaluar una expresión aritmética:

```
public class InterpreterClient {
    public static void main(String[] args) {
        // Construir una expresión aritmética: 1 +
        Expression expression = new
        SubtractExpression(
            new AddExpression(
                new NumberExpression(1),
                new NumberExpression(2)
            ),
            new NumberExpression(3)
        );

        // Evaluar la expresión
        int result = expression.interpret();

        System.out.println("Resultado de la
        expresión: " + result);
    }
}
```

Ahora, veamos el código en TypeScript.

Primero, definiremos una interfaz Expression que representará todas las expresiones que pueden ser interpretadas:

```
// Interfaz Expression que representa todas las
expresiones que pueden ser interpretadas
interface Expression {
    interpret(): number;
}
```

Luego, implementaremos clases concretas para representar diferentes tipos de expresiones, como números, sumas y restas:

```
// Clase concreta que representa un número
class NumberExpression implements Expression {
    private number: number;

    constructor(number: number) {
        this.number = number;
    }

    interpret(): number {
        return this.number;
    }
}
```

```
// Clase concreta que representa una suma
class AddExpression implements Expression {
    private left: Expression;
    private right: Expression;

    constructor(left: Expression, right:
    Expression) {
        this.left = left;
        this.right = right;
    }
```

```
    interpret(): number {
        return this.left.interpret() +
        this.right.interpret();
    }
}
```

```
// Clase concreta que representa una resta
class SubtractExpression implements Expression {
    private left: Expression;
    private right: Expression;

    constructor(left: Expression, right:
    Expression) {
        this.left = left;
        this.right = right;
    }

    interpret(): number {
        return this.left.interpret() -
        this.right.interpret();
    }
}
```

Por último creamos un código principal que nos permita interactuar con todo el código que creamos:

```
// Cliente que utiliza las expresiones para evaluar una
expresión aritmética
const main = () => {
    /* Construir una expresión aritmética: 1 + 2 -
       3*/
    const expression: Expression =
    new SubtractExpression(
        new AddExpression(
            new NumberExpression(1),
```

```
            new NumberExpression(2)
        ),
        new NumberExpression(3)
    );
    // Evaluar la expresión
    const result: number = expression.interpret();

    console.log("Resultado de la expresión:",
    result);
};
main();
```

En este ejemplo, hemos creado una expresión aritmética "1 + 2 - 3". Utilizamos las clases `NumberExpression`, `AddExpression` y `SubtractExpression` para construir la estructura de la expresión. Luego, llamamos al método `interpret()` en la expresión principal para obtener el resultado, que en este caso sería -1.

Este es un ejemplo simplificado del patrón Interpreter en acción, donde hemos creado un intérprete de expresiones aritméticas básicas. En aplicaciones del mundo real, el patrón Interpreter se utiliza para construir intérpretes para lenguajes de dominio específico más complejos, como lenguajes de consulta SQL o lenguajes de programación especializados.

Iterator

El patrón Iterator es un patrón de diseño de comportamiento que se utiliza para proporcionar una forma uniforme de acceder secuencialmente a los elementos de una colección de objetos sin exponer los detalles de su implementación subyacente. En otras palabras, el patrón Iterator permite que los clientes recorran los elementos de una colección sin necesidad de conocer la estructura interna de dicha colección.

Características clave del patrón Iterator:

- **Separación de preocupaciones:** Permite separar la lógica de acceso a los elementos de la colección de la lógica de la propia colección. Esto mejora la modularidad y el mantenimiento del código.

- **Interfaz común:** Define una interfaz común para todos los iteradores, lo que facilita su uso y permite a los clientes recorrer diferentes tipos de colecciones de manera uniforme.

- **Encapsulación de la colección:** Permite encapsular los detalles de implementación de la colección, como si es un arreglo, una lista enlazada u otra estructura, ocultándolos al cliente.

- **Recorrido secuencial:** Proporciona un mecanismo para recorrer los elementos de la colección secuencialmente, avanzando de uno en uno.

Un ejemplo común de uso del patrón Iterator en Java es a través de la clase Iterator que forma parte del Framework de Colecciones de Java (java.util). Esta clase abstracta define métodos como `hasNext()` y `next()` que permiten a los clientes recorrer colecciones como listas, conjuntos y mapas de manera uniforme.

En resumen, el patrón Iterator es ampliamente utilizado en lenguajes de programación como Java para facilitar la navegación a través de colecciones de objetos, promoviendo la encapsulación y la reutilización del código de recorrido de elementos.

Pasemos al ejemplo, en este ejemplo, crearemos una colección de libros y utilizaremos un iterador para recorrerlos.

Primero, definiremos una interfaz `Iterator` que declare los métodos comunes que debe proporcionar cualquier iterador:

```
// Interfaz Iterator
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

Ahora, crearemos la clase concreta que implementará nuestro iterador.

```
// Clase concreta que implementa Iterator para libros
public class BookIterator implements Iterator<String> {
    private String[] books;
    private int position = 0;

    public BookIterator(String[] books) {
        this.books = books;
    }

    public boolean hasNext() {
        return position < books.length;
    }

    public String next() {
        if (hasNext()) {
            return books[position++];
        }
        return null;
    }
}
```

Luego, crearemos una interfaz Iterable que declara un método para obtener un iterador:

```java
// Interfaz Iterable
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

A continuación, implementaremos una clase concreta que representa una colección de libros y que implementa la interfaz Iterable:

```java
// Clase concreta que implementa Iterable para una colección de libros
public class BookCollection implements Iterable<String> {
    private String[] books;

    public BookCollection(String[] books) {
        this.books = books;
    }

    public Iterator<String> iterator() {
        return new BookIterator(books);
    }
}
```

Finalmente, utilizamos estas clases en el cliente:

```java
public class IteratorClient {
    public static void main(String[] args) {
        // Crear una colección de libros
        String[] booksArray = {
            "Libro 1",
            "Libro 2",
            "Libro 3",
            "Libro 4",
            "Libro 5"
        };
```

```
        BookCollection bookCollection = new
        BookCollection(booksArray);

        /* Utilizar el iterador para recorrer la
          colección */
        Iterator<String> iterator =
        bookCollection.iterator();
        while (iterator.hasNext()) {
            String book = iterator.next();
            System.out.println("Libro: " + book);
        }
    }
}
```

Ahora pasemos al código en TypeScript:

```
// Interfaz Iterator
interface MyIterator<T> {
    hasNext(): boolean;
    next(): T | null;
}
```

Ahora, crearemos la clase concreta que implementará nuestro iterador.

```
// Clase concreta que implementa Iterator para libros
class BookIterator implements MyIterator<string> {
    private books: string[];
    private position: number = 0;

    constructor(books: string[]) {
        this.books = books;
    }
    hasNext(): boolean {
        return this.position < this.books.length;
    }
```

```
    next(): string | null {
        if (this.hasNext()) {
            return this.books[this.position++];
        }
        return null;
    }
}
```

Luego, crearemos una interfaz Iterable que declara un método para obtener un iterador:

```
// Interfaz Iterable
interface MyIterable<T> {
    iterator(): MyIterator<T>;
}
```

A continuación, implementaremos una clase concreta que representa una colección de libros y que implementa la interfaz Iterable:

```
/* Clase concreta que implementa Iterable para una
colección de libros*/
class BookCollection implements MyIterable<string> {
    private books: string[];

    constructor(books: string[]) {
        this.books = books;
    }

    iterator(): MyIterator<string> {
        return new BookIterator(this.books);
    }
}
```

Finalmente, utilizamos estas clases en el cliente:

```
// Cliente
function main() {
    // Crear una colección de libros
    const booksArray: string[] = [
        "Libro 1",
        "Libro 2",
        "Libro 3",
        "Libro 4",
        "Libro 5",
    ];
    const bookCollection: MyIterable<string> = new
    BookCollection(booksArray);

    /* Utilizar el iterador para recorrer la
       colección */
    const iterator: MyIterator<string> =
    bookCollection.iterator();
    while (iterator.hasNext()) {
        const book: string = iterator.next()!;
        console.log("Libro:", book);
    }
}

// Ejecutar el cliente
main();
```

Explicación del código:

- Creamos una interfaz Iterator que define los métodos `hasNext()` para verificar si hay más elementos y `next()` para obtener el siguiente elemento.

- Implementamos una clase concreta `BookIterator` que implementa la interfaz Iterator para recorrer una matriz de libros.

- Definimos una interfaz `Iterable` que declara un método `iterator()` para obtener un iterador.

- Implementamos una clase concreta `BookCollection` que representa una colección de libros y proporciona una implementación del método `iterator()`. Esta clase devuelve una instancia de `BookIterator` para recorrer la matriz de libros.

- En el cliente `(IteratorClient)`, creamos una instancia de `BookCollection` y utilizamos su iterador para recorrer y mostrar los libros en la colección.

El patrón Iterator nos permite recorrer una colección sin conocer los detalles de su implementación interna, lo que hace que el código sea más flexible y fácil de mantener.

Mediator

El patrón Mediator es un patrón de diseño de comportamiento que se utiliza para reducir las dependencias complejas y no deseadas entre objetos en un sistema. Proporciona un punto centralizado (el mediador) a través del cual los objetos pueden comunicarse entre sí sin conocerse directamente. En otras palabras, el mediador actúa como un intermediario que coordina las interacciones entre objetos, evitando que los objetos se comuniquen directamente entre ellos.

El patrón Mediator es especialmente útil en sistemas donde un conjunto de objetos interactúa de manera compleja y se desea evitar un alto nivel de acoplamiento entre ellos. Ejemplos comunes de uso del patrón Mediator incluyen sistemas de chat, sistemas de control de tráfico aéreo, sistemas de gestión de eventos y sistemas de componentes de interfaz de usuario.

En resumen, el patrón Mediator proporciona una forma de gestionar las comunicaciones entre objetos de manera centralizada y desacoplada, lo que contribuye a la modularidad y la facilidad de mantenimiento del código.

Esta vez en el ejemplo, crearemos un sistema de chat en el que los usuarios se comunican a través de un mediador.

Veamos el código en Java.

Primero, definimos una interfaz `ChatMediator` que declara métodos para que los usuarios se registren y envíen mensajes:

```
// Interfaz Mediator
public interface ChatMediator {
    void sendMessage(String message, User user);
    void addUser(User user);
}
```

Luego, implementamos una clase concreta `ChatMediatorImpl` que implementa esta interfaz y maneja la comunicación entre usuarios:

```
// Clase concreta que implementa el Mediator
public class ChatMediatorImpl implements ChatMediator {
    private List<User> users;

    public ChatMediatorImpl() {
        this.users = new ArrayList<>();
    }

    public void addUser(User user) {
        users.add(user);
    }

    public void sendMessage(String message,
    User sender)
```

```
    {
        for (User user : users) {
            /* Evitar enviar el mensaje al
            remitente */
            if (user != sender) {
                user.receiveMessage(message);
            }
        }
    }
}
```

A continuación, definimos una clase `User` que representa un usuario del sistema de chat y que se comunica a través del mediador:

```
// Clase que representa un usuario
public class User {
    private String name;
    private ChatMediator mediator;

    public User(String name, ChatMediator mediator)
    {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }
    public void sendMessage(String message) {
        System.out.println(name + " envía: " +
        message);
        mediator.sendMessage(message, this);
    }

    public void receiveMessage(String message) {
        System.out.println(name + " recibe: " +
        message);
    }
}
```

Finalmente, en el cliente, creamos un mediador, varios usuarios y les permitimos comunicarse a través del mediador:

```
public class MediatorClient {
    public static void main(String[] args) {
        // Crear un mediador
        ChatMediator mediator =
        new ChatMediatorImpl();

        // Crear usuarios
        User user1 = new User("Usuario 1",
        mediator);

        User user2 = new User("Usuario 2",
        mediator);

        User user3 = new User("Usuario 3",
        mediator);

        /* Los usuarios envían mensajes a través
           del mediador */
        user1.sendMessage("Hola a todos");
        user2.sendMessage("Hola, Usuario 1");
        user3.sendMessage("Hola, ¿cómo están?");
    }
}
```

Ahora veamos el código en TypeScript.

Primero, definimos una interfaz `ChatMediator` que declara métodos para que los usuarios se registren y envíen mensajes:

```
// Interfaz Mediator
interface ChatMediator {
    sendMessage(message: string, user: User): void;
    addUser(user: User): void;
}
```

Luego, implementamos una clase concreta `ChatMediatorImpl` que implementa esta interfaz y maneja la comunicación entre usuarios:

```
// Clase concreta que implementa el Mediator
class ChatMediatorImpl implements ChatMediator {
    private users: User[] = [];

    addUser(user: User): void {
        this.users.push(user);
    }
    sendMessage(message: string, sender: User):
    void {
        for (const user of this.users) {
            /* Evitar enviar el mensaje al
               remitente*/
            if (user !== sender) {
                user.receiveMessage(message);
            }
        }
    }
}
```

A continuación, definimos una clase `User` que representa un usuario del sistema de chat y que se comunica a través del mediador:

```
// Clase que representa un usuario
class User {
    private name: string;
    private mediator: ChatMediator;

    constructor(name: string, mediator:
    ChatMediator) {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }

    sendMessage(message: string): void {
        console.log(this.name + " envía: " +
        message);
        this.mediator.sendMessage(message, this);
    }

    receiveMessage(message: string): void {
        console.log(this.name + " recibe: " +
        message);
    }
}
```

Finalmente, en el cliente, creamos un mediador, varios usuarios y les permitimos comunicarse a través del mediador:

```
// Crear un mediador
const mediator: ChatMediator = new ChatMediatorImpl();

// Crear usuarios
const user1: User = new User("Usuario 1", mediator);
const user2: User = new User("Usuario 2", mediator);
const user3: User = new User("Usuario 3", mediator);
```

```
// Los usuarios envían mensajes a través del mediador
user1.sendMessage("Hola a todos");
user2.sendMessage("Hola, Usuario 1");
user3.sendMessage("Hola, ¿cómo están?");
```

Explicación del código:

- Creamos una interfaz `ChatMediator` que define los métodos para enviar mensajes y agregar usuarios al sistema.
- Implementamos una clase `ChatMediatorImpl` que implementa esta interfaz y gestiona la lista de usuarios y la comunicación entre ellos.
- Creamos una clase `User` que representa un usuario del chat. Cada usuario se registra en el mediador cuando se crea y puede enviar y recibir mensajes a través de él.
- En el cliente `(MediatorClient)`, creamos un mediador, varios usuarios y les permitimos comunicarse a través del mediador. Los mensajes enviados por un usuario se distribuyen a todos los demás usuarios a través del mediador, lo que permite la comunicación indirecta entre ellos.

Este ejemplo ilustra cómo el patrón Mediator facilita la comunicación indirecta entre objetos, lo que puede ser útil en sistemas en los que los objetos deben interactuar pero se desea reducir la dependencia directa entre ellos.

Memento

El patrón Memento es un patrón de diseño de comportamiento que se utiliza para capturar y externalizar el estado interno de un objeto de manera que el objeto pueda ser restaurado a ese estado en un momento posterior sin revelar los detalles de su implementación interna.

Este patrón se suele usar cuando necesitas mantener un historial de cambios en un objeto o proporcionar la funcionalidad "deshacer" en una aplicación. El Memento permite que un objeto capture su estado actual y lo almacene en un objeto llamado "memento", que puede ser almacenado en una colección o incluso enviado a otro lugar. En un momento posterior, el objeto puede restaurar su estado anterior a partir del memento.

En el ejemplo vamos a simular un un sistema de editor de texto que permite al usuario escribir y guardar instantáneas (mementos) de su trabajo para restaurarlas más tarde.

Primero el código en Java.

Primero, definiremos la clase TextEditor que representa el editor de texto y el estado que queremos capturar:

```
// Clase que representa el editor de texto
public class TextEditor {
    private StringBuilder text = new
    StringBuilder();

    public void write(String content) {
        text.append(content);
```

```
    }

    public String getContent() {
        return text.toString();
    }

    public Memento save() {
        return new Memento(text.toString());
    }

    public void restore(Memento memento) {
        text = new
        StringBuilder(memento.getSavedState());
    }

    // Clase interna Memento
    public static class Memento {
        private final String state;

        private Memento(String state) {
            this.state = state;
        }

        private String getSavedState() {
            return state;
        }
    }
}
```

Luego, en el cliente, utilizaremos el editor de texto y los mementos:

```
public class MementoClient {
    public static void main(String[] args) {
        // Crear el editor de texto
        TextEditor editor = new TextEditor();
        // Escribir contenido
        editor.write("Primera línea\n");
```

```
        editor.write("Segunda línea\n");
        // Guardar el estado actual
        TextEditor.Memento memento = editor.save();
        // Escribir más contenido
        editor.write("Tercera línea\n");
        // Restaurar al estado anterior
        editor.restore(memento);

        /* Obtener y mostrar el contenido
        restaurado*/
        System.out.println("Contenido
        restaurado:\n" + editor.getContent());
    }
}
```

Ahora veamos el código en TypeScript y después veremos la explicación del código para ambos casos:

Primero, definiremos la clase `TextEditor` y la clase `Memento` que representa el editor de texto y el estado que queremos capturar:

```
// Clase que representa el editor de texto
class TextEditor {
    private text: string = "";

    write(content: string): void {
        this.text += content;
    }

    getContent(): string {
        return this.text;
    }

    save(): Memento {
        return new Memento(this.text);
    }
```

```
    restore(memento: Memento): void {
        this.text = memento.getSavedState();
    }
}
```

```
// Clase Memento que almacena el estado del editor de texto
class Memento {
    private state: string;

    constructor(state: string) {
        this.state = state;
    }

    getSavedState(): string {
        return this.state;
    }
}
```

Luego, en el cliente, utilizaremos el editor de texto y los mementos:

```
// Cliente
function main() {
    // Crear el editor de texto
    const editor: TextEditor = new TextEditor();

    // Escribir contenido
    editor.write("Primera línea\n");
    editor.write("Segunda línea\n");

    // Guardar el estado actual
    const memento: Memento = editor.save();

    // Escribir más contenido
    editor.write("Tercera línea\n");
```

```
    // Restaurar al estado anterior
    editor.restore(memento);

    // Obtener y mostrar el contenido restaurado
    console.log("Contenido restaurado:\n" +
    editor.getContent());
}

// Ejecutar el cliente
main();
```

Explicación del código:

- Creamos la clase `TextEditor` que representa el editor de texto. Permite escribir contenido, obtener el contenido actual, guardar el estado en un memento y restaurar el estado desde un memento.

- La clase interna `Memento` almacena el estado capturado del editor.

- En el cliente `(MementoClient)`, creamos un editor de texto y escribimos algunas líneas de contenido. Luego, guardamos el estado actual en un memento.

- Continuamos escribiendo más contenido en el editor y luego restauramos el estado anterior desde el memento. Esto nos permite volver al estado anterior y mostrar el contenido restaurado.

Este ejemplo ilustra cómo el patrón Memento se utiliza para capturar y restaurar el estado de un objeto, en este caso, un editor de texto. Permite que el editor de texto vuelva a un estado anterior sin exponer su estructura interna.

Observer

El patrón Observador, también conocido como el patrón de Publicador-Suscriptor o el patrón Sujeto-Observador, es un patrón de diseño de comportamiento que define una relación de uno a muchos entre objetos. En este patrón, un objeto, llamado sujeto, mantiene una lista de observadores que dependen de él para ser notificados y actualizados automáticamente cuando cambia su estado. Cuando el sujeto cambia su estado, notifica a todos sus observadores, lo que les permite reaccionar y tomar acciones específicas.

Las características clave del patrón Observador son:

- **Desacoplamiento:** Permite que el sujeto y los observadores funcionen de manera independiente, sin necesidad de conocer los detalles internos del otro. Esto promueve la modularidad y el bajo acoplamiento en el diseño de software.

- **Notificación automática:** Los observadores se actualizan automáticamente cuando el sujeto cambia su estado, eliminando la necesidad de que los observadores verifiquen activamente el estado del sujeto.

- **Extensibilidad:** Puedes agregar o quitar observadores fácilmente sin modificar el sujeto, lo que hace que el patrón sea flexible y escalable.

El patrón Observador se usa comúnmente en situaciones en las que varios objetos necesitan estar informados y actuar en función de los cambios en el estado de otro objeto. Ejemplos típicos de uso incluyen sistemas de eventos en interfaces de usuario, sistemas de

notificación en aplicaciones de mensajería, y sistemas de seguimiento de cambios en datos en tiempo real.

Esta vez en el ejemplo crearemos un sistema simple de seguimiento de stock, donde los observadores (inversores) recibirán actualizaciones cuando el precio de una acción cambie.

Primero el código en Java.

Primero, definiremos una interfaz `Observer` que declara un método update que los observadores implementarán para recibir actualizaciones:

```java
// Interfaz Observer
public interface Observer {
    void update(String message);
}
```

Luego, crearemos una clase concreta `Stock` que representa una acción y mantendrá una lista de observadores interesados:

```java
import java.util.ArrayList;
import java.util.List;

// Clase concreta Observable (Sujeto)
public class Stock {
    private String symbol;
    private double price;
    private List<Observer> observers = new
    ArrayList<>();
    public Stock(String symbol, double price) {
        this.symbol = symbol;
        this.price = price;
    }

    public void addObserver(Observer observer) {
```

```
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }
    private void notifyObservers() {
        String message = "Precio de " + symbol + "
        ha cambiado a " + price;
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

A continuación, implementamos una clase Investor que representa un inversor y que implementa la interfaz `Observer` para recibir actualizaciones:

```
// Clase concreta Observer (Observador)
public class Investor implements Observer {
    private String name;

    public Investor(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " recibió una
        actualización: " + message);
    }
}
```

En el cliente, utilizamos estas clases para crear inversores y acciones, y luego simulamos cambios en los precios de las acciones:

```
public class ObserverClient {
    public static void main(String[] args) {
        // Crear acciones
        Stock appleStock = new Stock("AAPL",
        150.0);
        Stock googleStock = new Stock("GOOGL",
        2500.0);

        // Crear inversores
        Investor investor1 = new Investor(
        "Inversor 1");
        Investor investor2 = new Investor("
        Inversor 2");

        // Registrar inversores como observadores
        appleStock.addObserver(investor1);
        appleStock.addObserver(investor2);
        googleStock.addObserver(investor2);

        /* Simular cambios en los precios de las
           acciones */
        appleStock.setPrice(155.0);
        googleStock.setPrice(2600.0);
    }
}
```

Ahora veamos el código TypeScript.

Primero, definiremos una interfaz `Observer` que declara un método update que los observadores implementarán para recibir actualizaciones:

```
// Interfaz Observer
interface Observer {
    update(message: string): void;
}
```

Luego, crearemos una clase concreta `Stock` que representa una acción y mantendrá una lista de observadores interesados:

```
// Clase concreta Observable (Sujeto)
class Stock {
    private symbol: string;
    private price: number;
    private observers: Observer[] = [];

    constructor(symbol: string, price: number) {
        this.symbol = symbol;
        this.price = price;
    }

    addObserver(observer: Observer): void {
        this.observers.push(observer);
    }

    removeObserver(observer: Observer): void {
        this.observers = this.observers.filter(
        o => o !== observer);
    }


    setPrice(price: number): void {
        this.price = price;
        this.notifyObservers();
    }

    private notifyObservers(): void {
        const message = `Precio de ${this.symbol}
```

```
        ha cambiado a ${this.price}`;
        this.observers.forEach(observer =>
        observer.update(message));
    }
}
```

A continuación, implementamos una clase `Investor` que representa un inversor y que implementa la interfaz `Observer` para recibir actualizaciones:

```
// Clase concreta Observer (Observador)
class Investor implements Observer {
    private name: string;

    constructor(name: string) {
        this.name = name;
    }

    update(message: string): void {
        console.log(`${this.name} recibió una
        actualización: ${message}`);
    }
}
```

En el cliente, utilizamos estas clases para crear inversores y acciones, y luego simulamos cambios en los precios de las acciones:

```
// Cliente
function main() {
    // Crear acciones
    const appleStock: Stock = new Stock("AAPL",
    150.0);
    const googleStock: Stock = new Stock("GOOGL",
    2500.0);
```

```
    // Crear inversores
    const investor1: Investor = new
    Investor("Inversor 1");

    const investor2: Investor = new
    Investor("Inversor 2");


    // Registrar inversores como observadores
    appleStock.addObserver(investor1);
    appleStock.addObserver(investor2);
    googleStock.addObserver(investor2);

    /* Simular cambios en los precios de las
       acciones*/
    appleStock.setPrice(155.0);
    googleStock.setPrice(2600.0);
}

// Ejecutar el cliente
main();
```

Explicación del código:

- Creamos la interfaz Observer para declarar el método update, que será implementado por los observadores para recibir actualizaciones.

- La clase Stock representa una acción y permite la gestión de observadores. Cuando el precio de una acción cambia, notifica a sus observadores.

- La clase Investor implementa Observer y muestra las actualizaciones que recibe.

- En el cliente (ObserverClient), creamos acciones y observadores, registramos los observadores para recibir actualizaciones de acciones y luego simulamos cambios en los precios de las acciones.

Cuando se ejecuta este código, los inversores recibirán notificaciones cuando los precios de las acciones cambien, lo que ilustra cómo el patrón Observer permite una comunicación efectiva entre objetos sin que tengan que conocerse directamente.

State

El patrón State es un patrón de diseño de comportamiento que permite que un objeto altere su comportamiento cuando su estado interno cambia. Esto se logra representando cada estado posible como un objeto y delegando el comportamiento relacionado con el estado a estos objetos. El objeto "contexto" (que puede cambiar de estado) contiene una referencia a uno de los objetos de estado, lo que determina su comportamiento actual.

En resumen, el patrón State se utiliza para modelar comportamientos que dependen del estado interno de un objeto de manera que sea más fácil de entender y mantener, ya que cada estado es una clase separada que encapsula su propio comportamiento. Este patrón es especialmente útil cuando un objeto debe cambiar su comportamiento en función de múltiples estados diferentes, y permite que el objeto cambie de estado en tiempo de ejecución sin cambiar su interfaz.

Algunos beneficios clave del patrón State incluyen el aumento de la cohesión (cada estado se encuentra en su propia clase), la reducción del acoplamiento (el contexto no necesita conocer

detalles de implementación de los estados) y la facilidad de agregar nuevos estados sin modificar el código existente.

Para el ejemplo crearemos un simulador simple de una máquina expendedora de bebidas que puede tener diferentes estados, como "Sin dinero", "Seleccionando bebida" y "Entregando bebida". Dependiendo del estado actual, la máquina responderá de manera diferente a las acciones del usuario.

Primero, definiremos una interfaz `VendingMachineState` que declarará los métodos que deben ser implementados por los estados concretos:

```
// Interfaz que representa un estado de la máquina expendedora
public interface VendingMachineState {
    void insertMoney();
    void ejectMoney();
    void selectBeverage();
    void dispense();
}
```

Luego, crearemos clases concretas para cada estado de la máquina expendedora:

```java
// Clase concreta para el estado "Sin dinero"
public class NoMoneyState implements VendingMachineState
{
    @Override
    public void insertMoney() {
        System.out.println("Dinero insertado");
    }

    @Override
    public void ejectMoney() {
        System.out.println("No se puede expulsar
        dinero, no se ha insertado dinero");
    }

    @Override
    public void selectBeverage() {
        System.out.println("No se puede seleccionar
        una bebida, inserta dinero primero");
    }

    @Override
    public void dispense() {
        System.out.println("No se puede dispensar
        una bebida, inserta dinero primero");
    }
}
```

```java
// Clase concreta para el estado "Seleccionando bebida"
public class SelectingBeverageState implements
VendingMachineState {
    @Override
    public void insertMoney() {
        System.out.println("Dinero ya insertado,
        espere...");
    }

    @Override
    public void ejectMoney() {
        System.out.println("Dinero devuelto");
    }

    @Override
    public void selectBeverage() {
        System.out.println("Bebida seleccionada,
        espere...");
    }

    @Override
    public void dispense() {
        System.out.println("Bebida dispensada");
    }
}
```

```java
// Clase concreta para el estado "Entregando bebida"
public class DispensingState implements
VendingMachineState {
    @Override
    public void insertMoney() {
        System.out.println("Dinero ya insertado,
        espere...");
    }

    @Override
    public void ejectMoney() {
        System.out.println("Dinero devuelto");
    }

    @Override
    public void selectBeverage() {
        System.out.println("Bebida seleccionada,
        espere...");
    }

    @Override
    public void dispense() {
        System.out.println("Bebida dispensada");
    }
}
```

Cada clase concreta implementa la interfaz `VendingMachineState` y proporciona su propia lógica para las acciones relacionadas con su estado.

Luego, creamos la clase `VendingMachine` que mantiene el estado actual y realiza transiciones de estado:

```java
// Clase que representa la máquina expendedora
public class VendingMachine {
    private VendingMachineState state;

    public VendingMachine() {
        state = new NoMoneyState();
        /* Inicialmente, la máquina está en el
        estado "Sin dinero"*/
    }

    public void insertMoney() {
        state.insertMoney();
        if (state instanceof NoMoneyState) {
            state = new SelectingBeverageState();
        }
    }

    public void ejectMoney() {
        state.ejectMoney();
        if (state instanceof
            SelectingBeverageState) {
            state = new NoMoneyState();
        }
    }

    public void selectBeverage() {
        state.selectBeverage();
        if (state instanceof
            SelectingBeverageState) {
            state = new DispensingState();
        }
    }

    public void dispense() {
        state.dispense();
        if (state instanceof DispensingState) {
            state = new NoMoneyState();
```

```
        }
    }
}
```

Finalmente, en el cliente, podemos simular las interacciones del usuario con la máquina expendedora:

```
public class StateClient {
    public static void main(String[] args) {
        var vendingMachine = new VendingMachine();

        vendingMachine.insertMoney();
        vendingMachine.selectBeverage();
        vendingMachine.dispense();

        vendingMachine.insertMoney();
        vendingMachine.ejectMoney();

        vendingMachine.selectBeverage();
        vendingMachine.insertMoney();
        vendingMachine.selectBeverage();
        vendingMachine.dispense();
    }
}
```

Continuemos con el código en TypeScript:

Primero, definiremos una interfaz `VendingMachineState` que declarará los métodos que deben ser implementados por los estados concretos:

```
// Interfaz que representa un estado de la máquina expendedora
interface VendingMachineState {
    insertMoney(): void;
    ejectMoney(): void;
```

```
    selectBeverage(): void;
    dispense(): void;
}
```

Luego, crearemos clases concretas para cada estado de la máquina expendedora:

```
// Clase concreta para el estado "Sin dinero"
class NoMoneyState implements VendingMachineState {
    insertMoney(): void {
        console.log("Dinero insertado");
    }

    ejectMoney(): void {
        console.log("No se puede expulsar dinero,
        no se ha insertado dinero");
    }

    selectBeverage(): void {
        console.log("No se puede seleccionar una
        bebida, inserta dinero primero");
    }


    dispense(): void {
        console.log("No se puede dispensar una
        bebida, inserta dinero primero");
    }
}
```

```
/* Clase concreta para el estado "Seleccionando bebida"*/
class SelectingBeverageState implements
VendingMachineState {

    insertMoney(): void {
        console.log("Dinero ya insertado,
        espere...");
    }

    ejectMoney(): void {
        console.log("Dinero devuelto");
    }

    selectBeverage(): void {
        console.log("Bebida seleccionada,
        espere...");
    }

    dispense(): void {
        console.log("Bebida dispensada");
    }
}
```

```
// Clase concreta para el estado "Entregando bebida"
class DispensingState implements VendingMachineState {
    insertMoney(): void {
        console.log("Dinero ya insertado,
        espere...");
    }

    ejectMoney(): void {
        console.log("Dinero devuelto");
    }
```

```
    selectBeverage(): void {
        console.log("Bebida seleccionada,
        espere...");
    }

    dispense(): void {
        console.log("Bebida dispensada");
    }
}
```

Cada clase concreta implementa la interfaz `VendingMachineState` y proporciona su propia lógica para las acciones relacionadas con su estado.

Luego, creamos la clase `VendingMachine` que mantiene el estado actual y realiza transiciones de estado:

```
// Clase que representa la máquina expendedora
class VendingMachine {
    private state: VendingMachineState;

    constructor() {
        this.state = new NoMoneyState();
        /* Inicialmente, la máquina está en el
           estado "Sin dinero" */
    }

    insertMoney(): void {
        this.state.insertMoney();
        if (this.state instanceof NoMoneyState) {
            this.state = new
            SelectingBeverageState();
        }
    }
```

```
    ejectMoney(): void {
        this.state.ejectMoney();
        if (this.state instanceof
            SelectingBeverageState) {
            this.state = new NoMoneyState();
        }
    }

    selectBeverage(): void {
        this.state.selectBeverage();
        if (this.state instanceof
            SelectingBeverageState) {
            this.state = new DispensingState();
        }
    }


    dispense(): void {
        this.state.dispense();
        if (this.state instanceof DispensingState)
        {
            this.state = new NoMoneyState();
        }
    }
}
```

Finalmente, en el cliente, podemos simular las interacciones del usuario con la máquina expendedora:

```
// Cliente
function main() {
    const vendingMachine: VendingMachine = new
VendingMachine();

    vendingMachine.insertMoney();
    vendingMachine.selectBeverage();
    vendingMachine.dispense();
```

```
    vendingMachine.insertMoney();
    vendingMachine.ejectMoney();

    vendingMachine.selectBeverage();
    vendingMachine.insertMoney();
    vendingMachine.selectBeverage();
    vendingMachine.dispense();
}

// Ejecutar el cliente
main();
```

Strategy

El patrón Strategy, que también se conoce como el patrón de estrategia, es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Este patrón permite que el cliente elija el algoritmo concreto que debe utilizarse en tiempo de ejecución sin cambiar la estructura del código.

En otras palabras, el patrón Strategy desacopla el comportamiento de un objeto del objeto en sí mismo. Esto se logra mediante la definición de una interfaz o una clase abstracta que representa una estrategia (un conjunto de acciones) y la implementación de varias clases concretas que representan diferentes versiones o variantes de esa estrategia. El objeto que utiliza la estrategia (el contexto) tiene un miembro que apunta a una instancia de la estrategia y delega la ejecución de las acciones a esa estrategia.

Es decir, el patrón Strategy permite que el comportamiento de un objeto sea modular, extensible y configurable, lo que facilita la incorporación de nuevas estrategias sin afectar al código existente y permite que el cliente seleccione dinámicamente la estrategia

adecuada en tiempo de ejecución. Este patrón es útil en situaciones donde varias alternativas de algoritmos o comportamientos son aplicables y se deben intercambiar o seleccionar en función de la situación.

Para este ejemplo, crearemos un simulador simple de una aplicación de procesamiento de imágenes que permite aplicar diferentes filtros a una imagen.

Primero, definiremos una interfaz `ImageFilter` que declare un método apply para aplicar un filtro a una imagen:

```
// Interfaz para representar un filtro de imagen
public interface ImageFilter {
    void apply(String fileName);
}
```

A continuación, implementaremos algunas clases concretas que representan diferentes filtros de imagen. Por ejemplo, aquí está la clase `BlackAndWhiteFilter` que convierte una imagen en blanco y negro:

```
// Clase concreta que implementa un filtro de imagen en
blanco y negro
public class BlackAndWhiteFilter implements ImageFilter {
    @Override
    public void apply(String fileName) {
        System.out.println("Aplicando filtro blanco
        y negro a " + fileName);
    }
}
```

Luego, crearemos otra clase concreta, `SepiaFilter`, que aplica un filtro sepia a una imagen:

```
public class SepiaFilter implements ImageFilter {
    @Override
    public void apply(String fileName) {
        System.out.println("Aplicando filtro sepia
        a " + fileName);
    }
}
```

Ahora, vamos a crear la clase `ImageProcessor` que utiliza un filtro para procesar una imagen:

```
// Clase Contexto que utiliza un filtro para procesar una
imagen
public class ImageProcessor {
    private ImageFilter filter;

    public void setFilter(ImageFilter filter) {
        this.filter = filter;
    }

    public void processImage(String fileName) {
        filter.apply(fileName);
    }
}
```

Finalmente, en el cliente, podemos crear una instancia de `ImageProcessor` y cambiar el filtro en tiempo de ejecución para aplicar diferentes filtros a una imagen:

```
public class StrategyClient {
    public static void main(String[] args) {
        var imageProcessor = new ImageProcessor();

        // Aplicar el filtro en blanco y negro
        imageProcessor.setFilter(new
        BlackAndWhiteFilter());
        imageProcessor.processImage("imagen1.jpg");

        // Cambiar al filtro sepia y aplicarlo
        imageProcessor.setFilter(new
        SepiaFilter());
        imageProcessor.processImage("imagen2.jpg");
    }
}
```

Ahora creamos el ejemplo con TypeScript.

Primero, definiremos una interfaz `ImageFilter` que declare un método apply para aplicar un filtro a una imagen:

```
// Interfaz para representar un filtro de imagen
interface ImageFilter {
    apply(fileName: string): void;
}
```

A continuación, implementaremos algunas clases concretas que representan diferentes filtros de imagen. Por ejemplo, aquí está la clase `BlackAndWhiteFilter` que convierte una imagen en blanco y negro:

```
// Clase concreta que implementa un filtro de imagen en
blanco y negro
class BlackAndWhiteFilter implements ImageFilter {
    apply(fileName: string): void {
        console.log(`Aplicando filtro blanco y
        negro a ${fileName}`);
    }
}
```

Luego, crearemos otra clase concreta, `SepiaFilter`, que aplica un filtro sepia a una imagen:

```
// Clase concreta que implementa un filtro de imagen
sepia
class SepiaFilter implements ImageFilter {
    apply(fileName: string): void {
        console.log(`Aplicando filtro sepia a
        ${fileName}`);
    }
}
```

Ahora, vamos a crear la clase `ImageProcessor` que utiliza un filtro para procesar una imagen:

```
// Clase Contexto que utiliza un filtro para procesar una
imagen
class ImageProcessor {
    private filter: ImageFilter;

    setFilter(filter: ImageFilter): void {
        this.filter = filter;
    }
    processImage(fileName: string): void {
        this.filter.apply(fileName);
    }
}
```

Finalmente, en el cliente, podemos crear una instancia de `ImageProcessor` y cambiar el filtro en tiempo de ejecución para aplicar diferentes filtros a una imagen:

```
// Cliente
function main() {
    const imageProcessor = new ImageProcessor();

    // Aplicar el filtro en blanco y negro
    imageProcessor.setFilter(new
    BlackAndWhiteFilter());
    imageProcessor.processImage("imagen1.jpg");

    // Cambiar al filtro sepia y aplicarlo
    imageProcessor.setFilter(new SepiaFilter());
    imageProcessor.processImage("imagen2.jpg");
}

// Ejecutar el cliente
main();
```

En ambos ejemplos hemos definido una interfaz `ImageFilter`, implementado dos clases concretas para filtros de blanco y negro (`BlackAndWhiteFilter`) y sepia (`SepiaFilter`), y creado la clase ImageProcessor como contexto que utiliza un filtro para procesar una imagen.

Luego, en el cliente (`main`), creamos una instancia de `ImageProcessor` y cambiamos el filtro en tiempo de ejecución para aplicar diferentes filtros a las imágenes. El resultado se muestra en la consola.

Este ejemplo ilustra cómo el patrón Strategy permite cambiar dinámicamente el comportamiento de un objeto en tiempo de ejecución al encapsular algoritmos intercambiables en objetos

separados, lo que facilita la extensibilidad y la personalización de la lógica sin modificar el código existente.

Template Method

El método Template Method, es un patrón de diseño de comportamiento, que proporciona una estructura para definir un algoritmo en una superclase, pero permite que las subclases reemplacen o extiendan ciertos pasos del algoritmo sin cambiar su estructura general. En otras palabras, el patrón Template Method define un esqueleto de algoritmo en una clase base, pero permite que las subclases personalicen partes específicas del algoritmo según sea necesario.

En este caso para el ejemplo vamos a crear un pequeño framework de juegos que tiene un método de plantilla para el ciclo de vida de un juego, permitiendo a los desarrolladores crear juegos específicos que heredan de este framework y personalizar ciertos aspectos del juego según sus necesidades.

Veamos primero el código Java.

Primero, definimos una clase abstracta `Game` que contiene el método de plantilla play, que define el ciclo de vida básico de un juego:

```java
// Clase abstracta que define el ciclo de vida de un
juego
public abstract class Game {
    public abstract void initialize();
    public abstract void startPlay();
    public abstract void endPlay();

    // Método de plantilla que define el ciclo de vida de
un juego
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }
}
```

A continuación, creamos una subclase concreta `Chess` que hereda de `Game` y proporciona implementaciones específicas para los pasos abstractos:

```java
// Subclase que implementa un juego de ajedrez
public class Chess extends Game {
    @Override
    public void initialize() {
        System.out.println("Inicializando el juego
        de ajedrez");
    }

    @Override
    public void startPlay() {
        System.out.println("Comenzando el juego de
        ajedrez");
    }
```

```
    @Override
    public void endPlay() {
        System.out.println("Terminando el juego de
        ajedrez");
    }
}
```

Ahora, podemos crear una subclase diferente, por ejemplo, `Monopoly`, que también hereda de `Game` y proporciona sus propias implementaciones para los pasos abstractos:

```
// Subclase que implementa un juego de Monopoly
class Monopoly extends Game {
    @Override
    public void initialize() {
        System.out.println("Inicializando el juego
        de Monopoly");
    }

    @Override
    public void startPlay() {
        System.out.println("Comenzando el juego de
        Monopoly");
    }

    @Override
    public void endPlay() {
        System.out.println("Terminando el juego de
        Monopoly");
    }
}
```

En el cliente, podemos crear instancias de `Chess` o `Monopoly` y llamar al método play para iniciar el juego. Cada juego seguirá el ciclo de vida definido en la clase base `Game`, pero personalizará sus propios pasos específicos.

```
public class TemplateMethodClient {
    public static void main(String[] args) {
        System.out.println("Jugando al ajedrez:");
        Game chessGame = new Chess();
        chessGame.play();

        System.out.println("Jugando al
        Monopoly:");
        Game monopolyGame = new Monopoly();
        monopolyGame.play();
    }
}
```

Ahora veamos el código en TypeScript.

Primero, definimos una clase abstracta `Game` que contiene el método de plantilla play, que define el ciclo de vida básico de un juego:

```
abstract class Game {
    abstract initialize(): void;
    abstract startPlay(): void;
    abstract endPlay(): void;
    play(): void {
        this.initialize();
        this.startPlay();
        this.endPlay();
    }
}
```

A continuación, creamos una subclase concreta `Chess` que hereda de `Game` y proporciona implementaciones específicas para los pasos abstractos:

```
// Subclase que implementa un juego de ajedrez
class Chess extends Game {
    initialize(): void {
        console.log("Inicializando el juego de
        ajedrez");
    }

    startPlay(): void {
        console.log("Comenzando el juego de
        ajedrez");
    }

    endPlay(): void {
        console.log("Terminando el juego de
        ajedrez");
    }
}
```

Ahora, podemos crear una subclase diferente, por ejemplo, `Monopoly`, que también hereda de `Game` y proporciona sus propias implementaciones para los pasos abstractos:

```
// Subclase que implementa un juego de Monopoly
class Monopoly extends Game {
    initialize(): void {
        console.log("Inicializando el juego de
        Monopoly");
    }

    startPlay(): void {
        console.log("Comenzando el juego de
        Monopoly");
    }
```

```
    endPlay(): void {
        console.log("Terminando el juego de
        Monopoly");
    }
}
```

En el cliente, podemos crear instancias de `Chess` o `Monopoly` y llamar al método play para iniciar el juego. Cada juego seguirá el ciclo de vida definido en la clase base `Game`, pero personalizará sus propios pasos específicos.

```
function main() {
    console.log("Jugando al ajedrez:");
    const chessGame = new Chess();
    chessGame.play();

    console.log("Jugando al Monopoly:");
    const monopolyGame = new Monopoly();
    monopolyGame.play();
}
main();
```

Explicación del código:

- Hemos definido la clase abstracta `Game`, que contiene el método de plantilla play que define el ciclo de vida de un juego.

- Creamos dos subclases concretas, `Chess` y `Monopoly`, que heredan de `Game` y proporcionan sus propias implementaciones para los pasos abstractos.

- En el cliente `(TemplateMethodClient)`, creamos instancias de `Chess` y `Monopoly` y llamamos al método play para iniciar los juegos. Cada juego sigue el ciclo de vida general definido en la clase base, pero personaliza sus propios pasos específicos.

Este ejemplo ilustra cómo el patrón Template Method permite definir una estructura común para diferentes juegos, donde cada juego puede personalizar su propio comportamiento sin cambiar la estructura general del ciclo de vida del juego.

Visitor

El patrón de diseño Visitor es un patrón de diseño de comportamiento que permite agregar nuevas operaciones o funcionalidades a un conjunto de objetos sin modificar su estructura. Permite que un objeto visite (recorra) los objetos de otro conjunto de objetos y realice una operación específica en cada uno de ellos, de manera independiente de la estructura de la clase del objeto visitado.

Es decir, el patrón Visitor permite extender las funcionalidades de un conjunto de objetos sin modificar su estructura, promoviendo así el principio de Open/Closed (abierto/cerrado) de diseño, donde una clase debe estar abierta para su extensión pero cerrada para su modificación. Esto lo convierte en una herramienta valiosa para la construcción de sistemas flexibles y mantenibles.

Para tratar de crear un ejemplo claro y realista consideremos un sistema de procesamiento de documentos que necesita realizar operaciones específicas en diferentes tipos de elementos de documento, como párrafos y encabezados.

Primero, definimos las interfaces que representan los elementos del documento y las operaciones del visitante:

```java
// Interfaz que define un elemento de documento
public interface DocumentElement {
    void accept(DocumentVisitor visitor);
}

/* Implementación concreta de un elemento de párrafo*/
public class ParagraphElement implements DocumentElement
{
    @Override
    public void accept(DocumentVisitor visitor) {
        visitor.visitParagraph(this);
    }
}
```

```java
// Implementación concreta de un elemento de encabezado
public class HeaderElement implements DocumentElement {
    @Override
    public void accept(DocumentVisitor visitor) {
        visitor.visitHeader(this);
    }
}

// Interfaz que define las operaciones del visitante
public interface DocumentVisitor {
    void visitParagraph(ParagraphElement
    paragraph);

    void visitHeader(HeaderElement header);
}
```

Luego, creamos una implementación concreta del visitante que realiza operaciones específicas en los elementos del documento:

```java
// Implementación concreta de un visitante que realiza operaciones en el documento
public class DocumentProcessor implements DocumentVisitor
{
    @Override
    public void visitParagraph(ParagraphElement
    paragraph) {
        System.out.println("Procesando un
        párrafo");
    }


    @Override
    public void visitHeader(HeaderElement header) {
        System.out.println("Procesando un
        encabezado");
    }
}
```

Finalmente, podemos crear un documento con elementos y utilizar el visitante para procesar cada elemento sin tener que modificar la estructura del documento:

```java
public class VisitorClient {
    public static void main(String[] args) {
        DocumentElement[] elements = {
          new ParagraphElement(),
          new HeaderElement()
        };
        var visitor = new DocumentProcessor();

        for (DocumentElement element : elements) {
            element.accept(visitor);
        }
    }
}
```

Ahora veamos el código en TypeScript.

Primero, definimos las interfaces que representan los elementos del documento y las operaciones del visitante:

```
// Interfaz que define un elemento de documento
interface DocumentElement {
    accept(visitor: DocumentVisitor): void;
}

// Implementación concreta de un elemento de párrafo
class ParagraphElement implements DocumentElement {
    accept(visitor: DocumentVisitor): void {
        visitor.visitParagraph(this);
    }
}

// Implementación concreta de un elemento de encabezado
class HeaderElement implements DocumentElement {
    accept(visitor: DocumentVisitor): void {
        visitor.visitHeader(this);
    }
}

// Interfaz que define las operaciones del visitante
interface DocumentVisitor {
    visitParagraph(paragraph: ParagraphElement):
    void;
    visitHeader(header: HeaderElement): void;
}
```

Luego, creamos una implementación concreta del visitante que realiza operaciones específicas en los elementos del documento:

```
/* Implementación concreta de un visitante que realiza
operaciones en el documento*/
class DocumentProcessor implements DocumentVisitor {
    visitParagraph(paragraph: ParagraphElement):
    void {
        console.log("Procesando un párrafo");
    }

    visitHeader(header: HeaderElement): void {
        console.log("Procesando un encabezado");
    }
}
```

Finalmente, podemos crear un documento con elementos y utilizar el visitante para procesar cada elemento sin tener que modificar la estructura del documento:

```
// Cliente
function main() {
    const elements: DocumentElement[] = [
      new ParagraphElement(),
      new HeaderElement()
    ];
    const visitor: DocumentVisitor =
    new DocumentProcessor();

    for (const element of elements) {
        element.accept(visitor);
    }
}
// Ejecutar el cliente
main();
```

En este ejemplo:

- Hemos definido las interfaces `DocumentElement` y `DocumentVisitor`. `DocumentElement` representa los elementos del documento que pueden aceptar visitantes, y `DocumentVisitor` define las operaciones que el visitante puede realizar en esos elementos.

- Creamos dos clases concretas que implementan `DocumentElement`: `ParagraphElement` y `HeaderElement`. Cada una de ellas implementa el método accept, que permite que el visitante realice operaciones específicas en ese tipo de elemento.

- Implementamos `DocumentProcessor`, una clase concreta de visitante que implementa las operaciones del visitante para procesar párrafos y encabezados.

- En el cliente (`VisitorClient`), creamos un array de elementos de documento y un visitante `DocumentProcessor`. Iteramos a través de los elementos y llamamos a `accept` en cada elemento para que el visitante realice la operación correspondiente.

Este ejemplo ilustra cómo el patrón Visitor permite agregar operaciones a objetos individuales (en este caso, elementos de documento) sin cambiar su estructura. Esto es especialmente útil en situaciones donde se tienen múltiples tipos de elementos y se necesitan realizar diferentes operaciones en ellos sin modificar su código.

Conclusión

Hemos explorado una amplia gama de patrones de comportamiento que son fundamentales para el diseño y la implementación de sistemas de software eficientes y flexibles.

Cada uno de estos patrones aborda desafíos específicos relacionados con la interacción y la comunicación entre objetos en una aplicación. Aquí tienes una recapitulación de algunas ventajas y desventajas clave de trabajar con patrones de comportamiento:

Ventajas:

- **Separación de Responsabilidades:** Los patrones de comportamiento promueven la separación de responsabilidades entre objetos, lo que facilita la comprensión y el mantenimiento del código.

- **Reutilización de Código:** Facilitan la reutilización de comportamientos y algoritmos comunes en diferentes partes de una aplicación, lo que reduce la duplicación de código.

- **Flexibilidad y Adaptabilidad:** Los patrones de comportamiento hacen que las aplicaciones sean más flexibles y adaptables al permitir cambios en el comportamiento sin afectar la estructura del código.

- **Extensibilidad:** Ayudan a crear sistemas que pueden crecer y evolucionar con facilidad a medida que cambian los requisitos, sin requerir cambios importantes en el código existente.

Desventajas:

- **Complejidad Añadida:** La implementación de patrones de comportamiento puede introducir una capa adicional de complejidad en el código, lo que puede dificultar su comprensión y mantenimiento.

- **Sobrecarga de Abstracción:** En algunos casos, la introducción excesiva de abstracciones y patrones puede aumentar la sobrecarga y afectar negativamente el rendimiento del sistema.

- **Tiempo de Desarrollo:** Implementar patrones de comportamiento puede requerir más tiempo de desarrollo inicial en comparación con enfoques más simples.

- **Sobrediseño:** Existe el riesgo de sobrediseñar un sistema, aplicando patrones innecesarios que complican en lugar de simplificar.

En resumen, los patrones de comportamiento son herramientas poderosas para los desarrolladores de software, pero su aplicación debe ser cuidadosa y basada en las necesidades del proyecto. Los beneficios de una mayor claridad, reutilización y extensibilidad son evidentes, pero es importante equilibrar estos beneficios con la complejidad potencial introducida por los patrones. Como desarrollador, comprender y aplicar estos patrones te brinda una valiosa caja de herramientas para abordar desafíos de diseño y comunicación en tus proyectos de software.

Conclusiones generales

En este ebook, hemos explorado una amplia gama de patrones de diseño en los lenguajes TypeScript y Java. Estos patrones, que se basan en principios de ingeniería de software probados y sólidos, brindan soluciones efectivas para desafíos comunes en el diseño y desarrollo de software. Han demostrado ser herramientas poderosas para crear sistemas más flexibles, mantenibles y escalables.

La importancia de los patrones de diseño

Los patrones de diseño no solo son técnicas avanzadas para resolver problemas de diseño, sino que también representan un conjunto de mejores prácticas basadas en la experiencia acumulada por la comunidad de desarrolladores a lo largo de los años. Algunos de los beneficios clave que proporcionan incluyen:

- **Reutilización de Código:** Los patrones permiten la reutilización de soluciones probadas en múltiples contextos, reduciendo la duplicación de código y acelerando el desarrollo.

- **Flexibilidad y Adaptabilidad:** Facilitan la creación de sistemas flexibles que pueden evolucionar con facilidad a medida que cambian los requisitos.

- **Mantenimiento Simplificado:** Mejoran la comprensión del código y reducen la complejidad, lo que facilita la corrección de errores y la incorporación de nuevas características.

- **Escalabilidad:** Ayudan a diseñar sistemas que pueden crecer y escalar sin introducir problemas adicionales.

Herramientas para la Creatividad:

Los patrones de diseño son como un conjunto de herramientas en manos de un diseñador. Al comprender estos patrones y cuándo aplicarlos, los desarrolladores pueden abordar desafíos de diseño de manera más eficiente y, al mismo tiempo, desatar su creatividad para crear soluciones únicas y efectivas.

Aprender y Aplicar:

Aprender patrones de diseño lleva tiempo y práctica. No todos los patrones son aplicables en todas las situaciones, y la elección del patrón adecuado depende de los requisitos específicos de cada proyecto. La experiencia y el buen juicio desempeñan un papel importante en esta elección.

En resumen, los patrones de diseño son herramientas valiosas para cualquier desarrollador. Al comprender estos patrones y cómo aplicarlos en TypeScript y Java, tienes una base sólida para diseñar sistemas de software robustos y eficientes. A medida que continúas tu viaje en el desarrollo de software, te animamos a seguir explorando, aprendiendo y aplicando estos patrones para perfeccionar tu habilidad como diseñador y desarrollador de software.