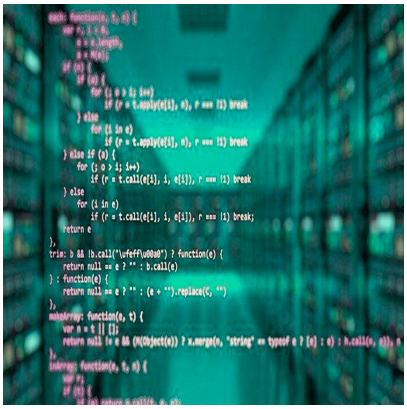

¿Qué son los patrones de diseño en programación y porque debes aprenderlos?

Descripción

Bienvenido a este emocionante viaje en el que exploraremos un elemento fundamental en el universo de la programación: los **patrones de diseño**. Si eres nuevo en este apasionante mundo o simplemente estás buscando ampliar tus conocimientos, has llegado al lugar adecuado.

¡Explora el fascinante mundo de los patrones de diseño y Struts para elevar tus habilidades en desarrollo web! Únete a nuestro **curso gratis de Patrones de Diseño y Struts** que desmitifica estos conceptos. Además, descubre nuestra amplia oferta de **cursos online gratis de programación**, desde **curso gratis de Desarrollo de Aplicaciones Web con ASP.NET** , **curso gratis de Programacion.net**, **curso gratis de Análisis En Código Bdd Y Tdd** **,** **curso gratis de Programación en Visual C++**, hasta **curso gratis de Python y Django**. ¡Inicia tu viaje de aprendizaje hoy en nuestra plataforma de educación de calidad!

Imagina la programación como un lenguaje, y los patrones de diseño como las reglas gramaticales que le dan estructura y coherencia. En esencia, son soluciones probadas y comprobadas para problemas recurrentes en el desarrollo de software. Pero, ¿por qué deberías sumergirte en este fascinante tema?

Antes de sumergirnos en el porqué, es esencial entender el qué. Los **patrones de diseño** son como recetas maestras en el mundo de la programación, proporcionando soluciones elegantes y eficientes para situaciones comunes.

Ahora, hablemos del meollo del asunto. ¿Por qué invertir tu tiempo en aprender patrones de diseño? La respuesta es simple, pero poderosa: **te hacen un programador más eficiente y habilidoso**.

Imagina que estás resolviendo un rompecabezas. Si conoces los patrones de diseño, no solo encontrarás la pieza correcta más rápido,

¿Qué son los Patrones de Diseño?

Los **patrones de diseño** son soluciones probadas y comprobadas para problemas recurrentes en el desarrollo de software. En otras palabras, son como recetas maestras que los desarrolladores utilizan

para resolver desafíos usuales de manera eficiente y elegante.

Imagina que estás construyendo una casa. En lugar de diseñar cada elemento desde cero, utilizarías planos arquitectónicos estándar que han sido refinados a lo largo del tiempo para garantizar la funcionalidad y la estabilidad. De manera similar, los patrones de diseño ofrecen un enfoque estructurado y optimizado para resolver problemas de programación.

Patrones de diseño y su relevancia en el desarrollo de software.

Los **patrones de diseño** son elementos fundamentales en el desarrollo de software, ya que ofrecen soluciones probadas y comprobadas para problemas recurrentes en el diseño y la implementación de sistemas informáticos. Su relevancia radica en varios aspectos clave:

Estandarización y Consistencia: Los patrones de diseño proporcionan un vocabulario común y una forma estructurada de abordar problemas de diseño. Esto promueve la estandarización y la coherencia en el desarrollo de software, lo que facilita la comprensión del código y la colaboración entre desarrolladores.

Eficiencia en el Desarrollo: Al utilizar patrones de diseño, los desarrolladores pueden aprovechar soluciones predefinidas y optimizadas para problemas comunes. Esto permite acelerar el proceso de desarrollo, ya que no es necesario reinventar la rueda cada vez que se enfrentan a un problema similar.

Mantenibilidad y Escalabilidad: Los patrones de diseño promueven prácticas de programación que conducen a un código más modular, flexible y fácil de mantener. Esto facilita la evolución del software a lo largo del tiempo y su adaptación a nuevos requisitos o cambios en el entorno.

Reutilización de Código: Al encapsular soluciones a problemas comunes en forma de patrones de diseño, se fomenta la reutilización de código. Esto permite aprovechar la experiencia acumulada en la industria y reduce la duplicación de esfuerzos en el desarrollo de software.

Abstracción y Separación de Responsabilidades: Los patrones de diseño promueven la abstracción y la separación de responsabilidades en el diseño de sistemas informáticos. Esto facilita la gestión de la complejidad del software y permite construir sistemas más flexibles y adaptables.

Tipos de Patrones de Diseño

Los patrones de diseño se pueden clasificar en tres categorías principales: creacionales, estructurales y de comportamiento. Cada tipo aborda diferentes aspectos del diseño de software y proporciona soluciones específicas para distintos tipos de problemas.

patrones de diseño Creacionales

Los **patrones creacionales** se centran en la creación de objetos de manera eficiente y flexible. Estos patrones ayudan a ocultar los detalles de implementación de la creación de objetos, permitiendo queel código sea más mantenible y escalable. Algunos ejemplos de patrones creacionales incluyen:

- **Singleton:** Garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a dicha instancia.
- **Factory Method:** Define una interfaz para crear un objeto, pero permite a las subclases alterar el tipo de objetos que se crearán.
- **Abstract Factory:** Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

patrones de diseño Estructurales

Los **patrones estructurales** se enfocan en la composición de clases y objetos para formar estructuras más grandes y complejas. Estos patrones ayudan a garantizar que los cambios en la estructura no afecten a la funcionalidad del sistema. Algunos ejemplos de patrones estructurales son:

- **Adapter:** Permite que interfaces incompatibles trabajen juntas.
- **Decorator:** Añade funcionalidades a un objeto dinámicamente.
- **Composite:** Permite tratar objetos individuales y composiciones de objetos de manera uniforme.

patrones de diseño de Comportamiento

Los **patrones de comportamiento** se centran en la comunicación entre objetos, definiendo cómo interactúan y cómo se distribuyen las responsabilidades entre ellos. Estos patrones ayudan a mejorar la flexibilidad en la comunicación y el control de las acciones. Algunos ejemplos de patrones de comportamiento incluyen:

- **Observer:** Define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
- **Strategy:** Permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables.
- **Command:** Encapsula una solicitud como un objeto, lo que le permite parametrizar clientes con operaciones, encolar solicitudes, y soportar operaciones que pueden ser deshechas.

Estos son solo algunos ejemplos de patrones de diseño, pero existen muchos más, cada uno con su propio propósito y aplicación específica en el desarrollo de software.

Beneficios de Aprender Patrones de Diseño

El conocimiento y aplicación de **patrones de diseño** en proyectos de programación conlleva una serie de ventajas significativas que mejoran tanto la calidad del código como la eficiencia del desarrollo. A continuación, exploraremos algunos de los principales beneficios:

Mejora de la Eficiencia y Mantenibilidad del Código

Los patrones de diseño desempeñan un papel fundamental en la mejora de la eficiencia y mantenibilidad del código. Veamos cómo:

Mejora de la Legibilidad

Los patrones de diseño promueven la claridad y la coherencia en el código. Al seguir patrones establecidos, como el Singleton o el Factory Method, los desarrolladores pueden estructurar su código de manera más comprensible y consistente. Esto facilita la lectura y comprensión del código tanto para el desarrollador que lo escribe como para otros miembros del equipo que puedan necesitar trabajar en él en el futuro.

Aumento de la Escalabilidad

Los patrones de diseño están diseñados para adaptarse a diferentes situaciones y requisitos. Al utilizar patrones creacionales como Abstract Factory o Builder, los desarrolladores pueden crear componentes de software que sean fácilmente escalables. Esto significa que el código puede crecer y evolucionar con el tiempo sin necesidad de realizar cambios drásticos en su estructura. Además, los patrones estructurales, como el Composite, pueden ayudar a manejar la complejidad de los sistemas al permitir la composición de objetos en estructuras jerárquicas.

Facilitación de la Mantenibilidad

Los patrones de diseño promueven la modularidad y el bajo acoplamiento, lo que hace que el código sea más fácil de mantener. Al dividir el sistema en componentes independientes y bien definidos, los desarrolladores pueden realizar cambios en una parte del código sin afectar a otras partes. Esto reduce el riesgo de introducir errores inesperados y facilita la actualización y el mantenimiento continuo del software a lo largo del tiempo.

En resumen, los patrones de diseño son herramientas poderosas que pueden mejorar significativamente la eficiencia y mantenibilidad del código al promover la legibilidad, escalabilidad y modularidad. Al comprender y aplicar estos patrones de manera adecuada, los desarrolladores pueden escribir software más robusto, flexible y fácil de mantener.

Promoción de Buenas Prácticas de Programación

Los **patrones de diseño** no solo proporcionan soluciones a problemas comunes en el desarrollo de software, sino que también fomentan la adopción de buenas prácticas de programación. Veamos cómo los patrones de diseño contribuyen a la promoción de estas prácticas:

Modularidad

Los patrones de diseño promueven la descomposición del sistema en componentes más pequeños y

autónomos. Esto fomenta la modularidad, que es una de las principales prácticas en programación. Al tener componentes independientes y bien definidos, el código se vuelve más legible, mantenible y reutilizable.

Bajo Acoplamiento

El bajo acoplamiento es esencial para escribir código flexible y fácil de mantener. Los patrones de diseño ayudan a reducir el acoplamiento entre clases y módulos, lo que significa que los cambios en una parte del sistema tienen un impacto mínimo en otras partes. Esto facilita la evolución del software y hace que sea menos propenso a errores.

Alta Cohesión

Los patrones de diseño también favorecen la alta cohesión, que se refiere a la relación lógica y funcional entre los elementos de un módulo. Al utilizar patrones como el Singleton o el Strategy, los desarrolladores pueden agrupar funcionalidades relacionadas en clases cohesivas, lo que facilita su comprensión y mantenimiento.

Flexibilidad y Adaptabilidad

Los patrones de diseño están diseñados para ser flexibles y adaptables a diferentes situaciones y requisitos. Esto fomenta la escritura de código que pueda evolucionar con el tiempo y satisfacer las necesidades cambiantes del usuario. Al utilizar patrones como el Factory Method o el Observer, los desarrolladores pueden diseñar sistemas que sean fácilmente extensibles y modificables.

Documentación y Comunicación

Los patrones de diseño proporcionan un lenguaje común y una forma estructurada de describir soluciones a problemas de programación. Esto facilita la comunicación entre miembros del equipo y ayuda a documentar el diseño del software de manera clara y concisa. La utilización de patrones de diseño también ayuda a transmitir conocimientos y experiencias entre desarrolladores, promoviendo así un mejor entendimiento del código.

En resumen, los patrones de diseño no solo son herramientas para resolver problemas técnicos, sino que también promueven prácticas de programación sólidas, como la modularidad, el bajo acoplamiento, la alta cohesión y la flexibilidad. Al adoptar y aplicar estos patrones de manera adecuada, los desarrolladores pueden escribir código más limpio, mantenible y robusto.

Facilitación de la Colaboración y Comprensión del Código

Los **patrones de diseño** no solo son herramientas para resolver problemas técnicos, sino que también desempeñan un papel crucial en la facilitación de la colaboración y la comprensión del código entre equipos de desarrollo. Veamos cómo los patrones de diseño contribuyen a este aspecto:

Lenguaje Común:

Los patrones de diseño proporcionan un lenguaje común que los desarrolladores pueden utilizar para comunicarse entre ellos. Al tener un conjunto estándar de términos y conceptos, los equipos pueden discutir y entender mejor las soluciones propuestas, evitando malentendidos y confusiones.

Conocimiento Compartido:

Al aplicar patrones de diseño reconocidos y bien documentados, los equipos de desarrollo pueden aprovechar el conocimiento compartido en la industria. Esto significa que los desarrolladores pueden beneficiarse de las mejores prácticas y experiencias acumuladas, lo que resulta en una mejor calidad de código y una mayor eficiencia en el desarrollo.

Separación de Responsabilidades:

Los patrones de diseño promueven la separación de responsabilidades entre diferentes componentes del sistema. Al seguir patrones como el MVC (Modelo-Vista-Controlador), los desarrolladores pueden dividir claramente el código en capas funcionales, lo que facilita la comprensión de cada parte del sistema y permite que diferentes equipos se centren en áreas específicas.

Documentación Implícita:

La aplicación de patrones de diseño actúa como una forma de documentación implícita del código. Los patrones establecidos tienen una descripción clara de su propósito y funcionamiento, lo que ayuda a los desarrolladores a comprender rápidamente la estructura y el flujo del código sin necesidad de una documentación detallada adicional.

Mejora de la Legibilidad:

Los patrones de diseño promueven prácticas de programación que conducen a un código más limpio y legible. Al seguir patrones bien conocidos y establecidos, los desarrolladores pueden escribir código que sea fácil de entender para otros miembros del equipo, lo que facilita la colaboración y la revisión del código.

En resumen, los patrones de diseño son herramientas poderosas que no solo ayudan a resolver problemas técnicos, sino que también fomentan la colaboración y la comprensión del código entre equipos de desarrollo. Al adoptar y aplicar estos patrones de manera adecuada, los equipos pueden trabajar de manera más eficiente y producir software de mayor calidad.

Fomento de la Reutilización de Código

Los **patrones de diseño** son una herramienta fundamental para promover la reutilización de componentes y módulos de código en el desarrollo de software. Aquí hay algunos ejemplos de cómo los patrones de diseño facilitan esta reutilización:

Patrones Creacionales:

Singleton: Este patrón garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. Al utilizar el patrón Singleton, los desarrolladores pueden reutilizar la misma instancia de una clase en múltiples partes del código, en lugar de crear instancias nuevas cada vez que se necesite.

Factory Method y Abstract Factory: Estos patrones permiten la creación de objetos de manera flexible y encapsulada. Al utilizar estos patrones, los desarrolladores pueden definir una interfaz común para la creación de objetos y proporcionar implementaciones específicas según sea necesario. Esto facilita la reutilización de código al permitir que diferentes partes del sistema utilicen la misma lógica de creación de objetos.

Patrones Estructurales:

Adapter: Este patrón permite que objetos con interfaces incompatibles trabajen juntos. Al utilizar un adaptador, los desarrolladores pueden reutilizar componentes existentes en lugar de escribir nuevos desde cero para integrar diferentes sistemas o bibliotecas.

Decorator: Este patrón permite agregar funcionalidades adicionales a objetos existentes de manera dinámica. Al utilizar el patrón Decorator, los desarrolladores pueden envolver objetos con diferentes capas de funcionalidad sin modificar su estructura básica, lo que facilita la reutilización de código y la composición de comportamientos complejos.

Patrones de Comportamiento:

Observer: Este patrón define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Al utilizar el patrón Observer, los desarrolladores pueden reutilizar el código para gestionar eventos y notificaciones en diferentes partes del sistema de manera consistente.

Strategy: Este patrón permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Al utilizar el patrón Strategy, los desarrolladores pueden reutilizar la lógica de algoritmos en diferentes contextos sin modificar el código existente, lo que facilita la adaptación y la extensibilidad del sistema.

Ejemplos prácticos de patrones de diseño

Aquí tienes algunos ejemplos prácticos de cómo se pueden aplicar diferentes patrones de diseño en diversos escenarios de programación:

Ejemplo de uso de Singleton

Caso de Uso: Gestión de conexiones a una base de datos.

Descripción: Si se necesita una única instancia de un objeto que maneje las conexiones a una base de datos en toda la aplicación, se puede utilizar el patrón Singleton para garantizar que solo exista una instancia de la clase encargada de gestionar las conexiones.

Ejemplo de código usando el framework web Struts

En el contexto de un framework web como Struts, el patrón Singleton puede ser útil para la gestión de conexiones a la base de datos. Supongamos que estamos desarrollando una aplicación web que utiliza Struts y necesitamos una única instancia de un objeto que maneje todas las conexiones a la base de datos en la aplicación.

Para implementar este escenario utilizando el patrón Singleton en Struts, podríamos crear una clase llamada `DatabaseManager` que se encargue de establecer y administrar las conexiones a la base de datos. Aquí hay un ejemplo de cómo podría ser la implementación:


```
public class DatabaseManager {
private static DatabaseManager instance;
private Connection connection;

private DatabaseManager() {
// Constructor privado para evitar instanciación directa
// Inicializar la conexión a la base de datos aquí
// (Código de inicialización de conexión a la base de datos)
}

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

public Connection getConnection() {
return connection;
}

// Otros métodos para realizar operaciones en la base de datos
}
```

En este ejemplo, la clase `DatabaseManager` sigue el patrón Singleton. El constructor de la clase es privado, lo que significa que no se puede instanciar directamente desde fuera de la clase. En su lugar, la única manera de obtener una instancia de `DatabaseManager` es a través del método estático `getInstance()`.

Este método garantiza que solo exista una única instancia de `DatabaseManager` en toda la aplicación. Si se llama al método `getInstance()` varias veces, siempre se devolverá la misma

instancia.

En el contexto de una aplicación web desarrollada con Struts, podríamos utilizar esta clase `DatabaseManager` en nuestros action classes para obtener la conexión a la base de datos y realizar operaciones de lectura y escritura.

Ejemplo de uso de Factory Method con Struts

Caso de Uso: Creación de diferentes tipos de controladores en una aplicación web utilizando el framework Struts.

Descripción: En una aplicación web desarrollada con Struts, es común tener diferentes tipos de controladores para manejar las solicitudes del usuario, como controladores para la autenticación, el perfil del usuario, o para operaciones CRUD en diferentes entidades. El patrón Factory Method nos permite definir una interfaz común para la creación de estos controladores y delegar la creación de instancias específicas a clases concretas según el tipo de solicitud.

Supongamos que estamos desarrollando una aplicación web con Struts para gestionar una biblioteca virtual. Tenemos diferentes tipos de solicitudes de usuario, como la búsqueda de libros, la gestión de usuarios, y la administración de libros. Para cada tipo de solicitud, necesitamos un controlador correspondiente.

Ejemplo de código usando el framework web Struts

Aquí hay un ejemplo de cómo podríamos implementar el patrón Factory Method en Struts para crear los diferentes controladores:

```
public interface ControladorFactory {
Controlador crearControlador();
}

public class BusquedaLibrosFactory implements ControladorFactory {
@Override
public Controlador crearControlador() {
return new BusquedaLibrosControlador();
}
}

public class GestionUsuariosFactory implements ControladorFactory {
@Override
public Controlador crearControlador() {
return new GestionUsuariosControlador();
}
}

public class AdministracionLibrosFactory implements ControladorFactory {
@Override
public Controlador crearControlador() {
return new AdministracionLibrosControlador();
}
}
```

```
public class Controlador {
public void manejarSolicitud() {
// Lógica para manejar la solicitud del usuario
}
}

public class BusquedaLibrosControlador extends Controlador {
@Override
public void manejarSolicitud() {
// Lógica específica para manejar la búsqueda de libros
}
}

public class GestionUsuariosControlador extends Controlador {
@Override
public void manejarSolicitud() {
// Lógica específica para manejar la gestión de usuarios
}
}

public class AdministracionLibrosControlador extends Controlador {
@Override
public void manejarSolicitud() {
// Lógica específica para manejar la administración de libros
}
}
```


En este ejemplo, hemos definido una interfaz `ControladorFactory` que declara un método `crearControlador()`, el cual devuelve un objeto de tipo `Controlador`. Luego, hemos creado implementaciones concretas de esta interfaz para cada tipo de solicitud de usuario, como `BusquedaLibrosFactory`, `GestionUsuariosFactory` y `AdministracionLibrosFactory`, cada una de las cuales devuelve una instancia correspondiente de un controlador concreto.

Al utilizar este enfoque, podemos crear fácilmente nuevos controladores para diferentes tipos de solicitudes simplemente implementando la interfaz `ControladorFactory` y proporcionando la lógica necesaria para manejar la solicitud específica del usuario. Esto nos permite mantener nuestro código modular y fácil de mantener, siguiendo así los principios del patrón Factory Method.

Ejemplo de uso de Decorator con Struts

Caso de Uso: Modificación dinámica del contenido de una página web.

Descripción: Imagina que necesitas agregar funcionalidades adicionales, como un botón de compartir en redes sociales o un contador de visitas, a diferentes páginas web. El patrón Decorator permite envolver cada página web con diferentes capas de funcionalidad adicional sin modificar su estructura original, lo que facilita la incorporación de nuevas características sin afectar a las existentes.

Ejemplo de código usando el framework web Struts

```
// Interfaz que define el comportamiento de una página web
```

```
public interface PaginaWeb {
    String mostrarContenido();
}

// Implementación concreta de una página web
public class PaginaWebSimple implements PaginaWeb {
    @Override
    public String mostrarContenido() {
        return "Contenido básico de la página web.";
    }
}

// Clase abstracta que actúa como decorador
public abstract class DecoradorPaginaWeb implements PaginaWeb {
    protected PaginaWeb paginaWeb;

    public DecoradorPaginaWeb(PaginaWeb paginaWeb) {
        this.paginaWeb = paginaWeb;
    }

    @Override
    public String mostrarContenido() {
        return paginaWeb.mostrarContenido();
    }
}

// Decorador concreto que agrega un botón de compartir en redes sociales
public class BotonCompartirDecorator extends DecoradorPaginaWeb {
    public BotonCompartirDecorator(PaginaWeb paginaWeb) {
        super(paginaWeb);
    }

    @Override
    public String mostrarContenido() {
        return super.mostrarContenido() + "<br/><button>Compartir en redes soc
    }
}

// Decorador concreto que agrega un contador de visitas
public class ContadorVisitasDecorator extends DecoradorPaginaWeb {
    public ContadorVisitasDecorator(PaginaWeb paginaWeb) {
        super(paginaWeb);
    }

    @Override
    public String mostrarContenido() {
        return super.mostrarContenido() + "<br/>Número de visitas: <span>1000<
    }
}

// Ejemplo de uso
public class Main {
    public static void main(String[] args) {
        // Crear una página web básica
        PaginaWeb paginaWeb = new PaginaWebSimple();
```

```
        // Agregar un botón de compartir
        paginaWeb = new BotonCompartirDecorator(paginaWeb);

        // Agregar un contador de visitas
        paginaWeb = new ContadorVisitasDecorator(paginaWeb);

        // Mostrar el contenido de la página web
        System.out.println(paginaWeb.mostrarContenido());
    }
}
```

En este ejemplo, hemos creado una interfaz `PaginaWeb` que define el comportamiento de una página web y una implementación concreta `PaginaWebSimple`. Luego, hemos definido una clase abstracta `DecoradorPaginaWeb` que actúa como base para los decoradores concretos. Finalmente, hemos creado dos decoradores concretos (`BotonCompartirDecorator` y `ContadorVisitasDecorator`) que agregan funcionalidades adicionales a la página web básica.

Al utilizar estos decoradores, podemos envolver dinámicamente una página web con las funcionalidades adicionales que necesitemos, sin modificar su estructura original. Esto nos permite agregar nuevas características a la página web de manera flexible y extensible, siguiendo el principio de diseño de software abierto a la extensión pero cerrado a la modificación (Open/Closed Principle).

Ejemplo de uso de Observer con Struts

Caso de Uso: Notificación de cambios en el estado de un archivo.

Descripción: Si se necesita que múltiples componentes de una aplicación sean notificados cuando un archivo cambie de estado (por ejemplo, cuando se guarda un archivo), se puede utilizar el patrón Observer. Los componentes interesados se registran como observadores y son notificados automáticamente cuando se produce el cambio de estado en el archivo.

Ejemplo de código usando el framework web Struts

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

// Interfaz del sujeto observable
interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Sujeto concreto que representa el archivo observado
class File implements Observable {
    private List<Observer> observers = new ArrayList<>();
    private String fileName;
    private boolean modified;

    public File(String fileName) {
```

```
        this.fileName = fileName;
        this.modified = false;
    }

    public void modify() {
        this.modified = true;
        notifyObservers();
    }

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }

    public String getFileName() {
        return fileName;
    }

    public boolean isModified() {
        return modified;
    }
}

// Interfaz del observador
interface Observer {
    void update(Observable observable);
}

// Observador concreto que representa un componente de la aplicación
class FileObserver implements Observer {
    private String componentName;

    public FileObserver(String componentName) {
        this.componentName = componentName;
    }

    @Override
    public void update(Observable observable) {
        File file = (File) observable;
        System.out.println("El archivo '" + file.getFileName() + "' ha cambiad
    }
}
```

```
// Clase de ejemplo que simula el uso de Observer con Struts
public class Main {
    public static void main(String[] args) {
        // Crear el archivo observado
        File file = new File("documento.txt");

        // Crear los observadores
        Observer observer1 = new FileObserver("Componente1");
        Observer observer2 = new FileObserver("Componente2");

        // Registrar los observadores en el archivo
        file.addObserver(observer1);
        file.addObserver(observer2);

        // Modificar el archivo (simula un cambio de estado)
        file.modify();
    }
}
```

En este ejemplo, la clase `File` actúa como el sujeto observable, que representa el archivo que se está observando. Los observadores interesados en el estado del archivo se registran con el sujeto observable mediante el método `addObserver()`. Cuando el archivo experimenta un cambio de estado (por ejemplo, se modifica), se invoca el método `notifyObservers()` para notificar a todos los observadores registrados. Cada observador, implementado por la clase `FileObserver`, recibe la notificación y realiza la acción correspondiente, como imprimir un mensaje en la consola.

Este ejemplo ilustra cómo se puede implementar el patrón Observer en una aplicación desarrollada con Struts para notificar cambios en el estado de un archivo a múltiples componentes de la aplicación.

Ejemplo de uso de Strategy con Struts

Caso de Uso: Selección dinámica de algoritmos de encriptación en una aplicación de seguridad.

Descripción: En una aplicación de seguridad, puede ser necesario utilizar diferentes algoritmos de encriptación dependiendo del tipo de datos que se esté manipulando. El patrón Strategy permite encapsular cada algoritmo de encriptación en una clase separada y seleccionar dinámicamente el algoritmo a utilizar en tiempo de ejecución, proporcionando así flexibilidad y extensibilidad al sistema.

En el contexto de una aplicación web construida con el framework Struts, podemos implementar el patrón Strategy para manejar la encriptación de contraseñas de usuarios. Supongamos que tenemos tres algoritmos de encriptación: SHA-256, MD5 y bcrypt. Dependiendo de los requisitos de seguridad y la sensibilidad de los datos, queremos poder cambiar el algoritmo de encriptación de manera dinámica sin afectar el resto de la aplicación.

Ejemplo de código usando el framework web Struts

A continuación, mostraremos cómo se puede implementar este ejemplo utilizando el patrón Strategy en una aplicación Struts:

```
// Interfaz para el algoritmo de encriptación
```

```
public interface EncryptorStrategy {
    String encrypt(String password);
}

// Implementación concreta del algoritmo SHA-256
public class SHA256Encryptor implements EncryptorStrategy {
    @Override
    public String encrypt(String password) {
        // Implementación del algoritmo SHA-256
        // Código para encriptar la contraseña con SHA-256
        return encryptedPassword;
    }
}

// Implementación concreta del algoritmo MD5
public class MD5Encryptor implements EncryptorStrategy {
    @Override
    public String encrypt(String password) {
        // Implementación del algoritmo MD5
        // Código para encriptar la contraseña con MD5
        return encryptedPassword;
    }
}

// Implementación concreta del algoritmo bcrypt
public class BcryptEncryptor implements EncryptorStrategy {
    @Override
    public String encrypt(String password) {
        // Implementación del algoritmo bcrypt
        // Código para encriptar la contraseña con bcrypt
        return encryptedPassword;
    }
}

// Clase Contexto que utiliza la estrategia seleccionada
public class PasswordEncryptorContext {
    private EncryptorStrategy strategy;

    public void setStrategy(EncryptorStrategy strategy) {
        this.strategy = strategy;
    }

    public String encryptPassword(String password) {
        return strategy.encrypt(password);
    }
}
```


En el ejemplo anterior, hemos definido una interfaz `EncryptorStrategy` que define el contrato para los diferentes algoritmos de encriptación. Luego, hemos implementado tres clases concretas (`SHA256Encryptor`, `MD5Encryptor` y `BcryptEncryptor`) que representan los diferentes algoritmos de encriptación.

La clase `PasswordEncryptorContext` actúa como el contexto que utiliza la estrategia seleccionada para encriptar contraseñas. Dependiendo de los requisitos de seguridad de la aplicación, podemos

configurar dinámicamente el algoritmo de encriptación deseado en tiempo de ejecución.

Esta implementación nos proporciona la flexibilidad y extensibilidad necesarias para cambiar el algoritmo de encriptación sin tener que modificar el código fuente de la aplicación. Con Struts, podemos integrar fácilmente este patrón en la lógica de negocio de nuestra aplicación web para manejar la encriptación de contraseñas de manera eficiente y segura.

Conclusiones ¿qué son los patrones de diseño en programación y porque debes aprenderlos?

En conclusión, los patrones de diseño representan un conjunto invaluable de soluciones probadas y comprobadas para problemas recurrentes en el desarrollo de software. A lo largo de este artículo, hemos explorado su definición, tipos principales y una variedad de beneficios que ofrecen a los desarrolladores.

Desde mejorar la eficiencia y mantenibilidad del código hasta promover buenas prácticas de programación, los patrones de diseño desempeñan un papel crucial en la creación de software robusto y escalable. Además, facilitan la colaboración entre equipos de desarrollo y promueven la reutilización de código, lo que resulta en una mayor productividad y calidad del producto final.

A través de ejemplos prácticos, hemos visto cómo los patrones de diseño se aplican en diferentes escenarios de programación, proporcionando soluciones elegantes y flexibles a una variedad de problemas. Ya sea creando una única instancia de un objeto con Singleton, seleccionando dinámicamente algoritmos con Strategy, o notificando cambios con Observer, los patrones de diseño ofrecen herramientas poderosas para enfrentar los desafíos del desarrollo de software.