UNIVERSIDAD DEL VALLE DE GUATEMALA
Ingeniería de Software 1, sección 30

TAREA INVESTIGATIVA: PATRONES DE DISEÑO

VENTAS E INVENTARIO DIVINO SEAS

Sofia Garcia - 22210
Julio Garcia Salas - 22076
Joaquin Campos - 22155
Juan Fernando Menéndez - 17444
Juan Pablo Solis - 22102

GUATEMALA, marzo de 2024

Patrones de diseño elegidos

Encargados	Patrón
Juan Pablo y Juan Fer	Factory
Julio y Sofi	Decorator
Campos y Juan Fer	Visitor

Desglose y asignación de tareas

Patrón Factory (Juan Pablo y Juan Fer)

Sábado 2 de marzo

- Intención
    - Encargado: Juan Pablo
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:00 AM
- Conocido como
    - Encargado: Juan Fer
    - Hora de inicio: 09:00 AM
    - Hora de fin: 09:30 AM
- Motivo
    - Encargado: Juan Pablo
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM

Domingo 3 de marzo

- Aplicaciones
    - Encargado: Juan Fer
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM
- Estructura
    - Encargado: Juan Pablo
    - Hora de inicio: 11:00 AM
    - Hora de fin: 12:00 PM
- Participantes
    - Encargado: Juan Fer
    - Hora de inicio: 12:00 PM
    - Hora de fin: 01:00 PM

Lunes 4 de marzo

- Colaboraciones
    - Encargado: Juan Pablo
    - Hora de inicio: 08:00 AM

        - Hora de fin: 09:00 AM
- Consecuencias
    - Encargado: Juan Fer
    - Hora de inicio: 09:00 AM
    - Hora de fin: 10:00 AM
- Implementación
    - Encargado: Juan Pablo
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM

Martes 5 de marzo

- Código de ejemplo
    - Encargado: Juan Fer
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:30 AM
- Usos conocidos
    - Encargado: Juan Pablo
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM
- Patrones relacionados
    - Encargado: Juan Fer
    - Hora de inicio: 10:30 AM
    - Hora de fin: 11:30 AM

Patrón Decorator (Julio y Sofi)

Sábado 2 de marzo

- Intención
    - Encargado: Julio
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:00 AM
- Conocido como
    - Encargado: Sofi
    - Hora de inicio: 09:00 AM
    - Hora de fin: 09:30 AM
- Motivo
    - Encargado: Julio
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM

Domingo 3 de marzo

- Aplicaciones
    - Encargado: Sofi
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM

- Estructura
    - Encargado: Julio
    - Hora de inicio: 11:00 AM
    - Hora de fin: 12:00 PM
- Participantes
    - Encargado: Sofi
    - Hora de inicio: 12:00 PM
    - Hora de fin: 01:00 PM

Lunes 4 de marzo

- Colaboraciones
    - Encargado: Julio
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:00 AM
- Consecuencias
    - Encargado: Sofi
    - Hora de inicio: 09:00 AM
    - Hora de fin: 10:00 AM
- Implementación
    - Encargado: Julio
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM

Martes 5 de marzo

- Código de ejemplo
    - Encargado: Sofi
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:30 AM
- Usos conocidos
    - Encargado: Julio
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM
- Patrones relacionados
    - Encargado: Sofi
    - Hora de inicio: 10:30 AM
    - Hora de fin: 11:30 AM

Patrón Visitor (Campos y Juan Fer)

Sábado 2 de marzo

- Intención
    - Encargado: Campos
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:00 AM
- Conocido como

        - Encargado: Juan Fer
        - Hora de inicio: 09:00 AM
        - Hora de fin: 09:30 AM
- Motivo
    - Encargado: Campos
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM

Domingo 3 de marzo

- Aplicaciones
    - Encargado: Juan Fer
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM
- Estructura
    - Encargado: Campos
    - Hora de inicio: 11:00 AM
    - Hora de fin: 12:00 PM
- Participantes
    - Encargado: Juan Fer
    - Hora de inicio: 12:00 PM
    - Hora de fin: 01:00 PM

Lunes 4 de marzo

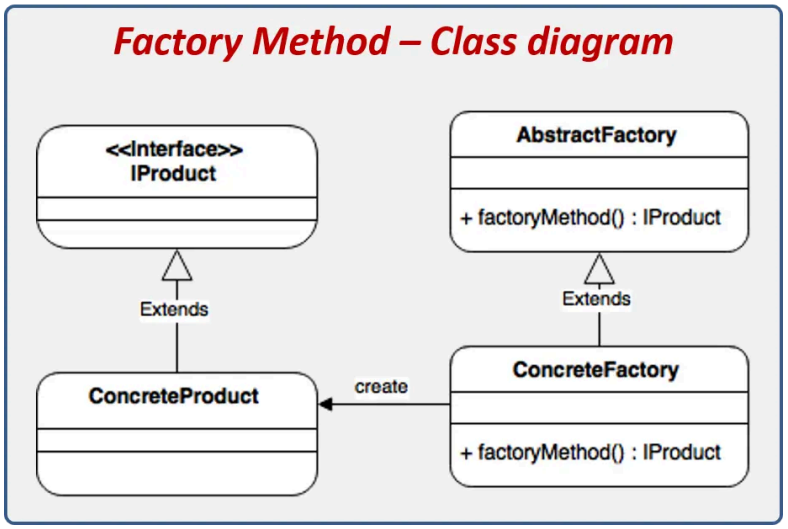
- Colaboraciones
    - Encargado: Campos
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:00 AM
- Consecuencias
    - Encargado: Juan Fer
    - Hora de inicio: 09:00 AM
    - Hora de fin: 10:00 AM
- Implementación
    - Encargado: Campos
    - Hora de inicio: 10:00 AM
    - Hora de fin: 11:00 AM

Martes 5 de marzo

- Código de ejemplo
    - Encargado: Juan Fer
    - Hora de inicio: 08:00 AM
    - Hora de fin: 09:30 AM
- Usos conocidos
    - Encargado: Campos
    - Hora de inicio: 09:30 AM
    - Hora de fin: 10:30 AM
- Patrones relacionados

- Encargado: Juan Fer
- Hora de inicio: 10:30 AM
- Hora de fin: 11:30 AM

Patrones de diseño

Factory
Intención
Proporcionar una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán.
Conocido como
Factory Method
Motivo
• Manejar y ocultar los detalles de creación de objetos. • Desacoplar el código de la aplicación de las clases concretas. • Proporcionar flexibilidad y extensibilidad al introducir nuevas clases concretas, ya que el código trabaja con productos a través de interfaces comunes.
Aplicaciones
• Cuando una clase no puede anticipar el tipo de objetos que debe crear. • Cuando una clase quiere que sus subclases especifiquen los objetos que crea. • Cuando las clases delegan responsabilidades a una de varias posibles subclases, y quieres localizar el conocimiento de a cuál delegar.
Estructura

Participantes

- Product: La interfaz o clase abstracta que define las operaciones que los objetos creados por el método de fábrica soportarán.
- ConcreteProduct: La clase que implementa o extiende Product, representando los objetos que son creados por el método de fábrica.
- Creator: La clase que declara el método de fábrica para crear objetos Product. Puede contener código común para manipular productos.
- ConcreteCreator: La clase que implementa el método de fábrica para devolver instancias de ConcreteProduct.

Colaboraciones

El cliente invoca el método de fábrica del Creator para crear un objeto Product. El ConcreteCreator crea el ConcreteProduct correspondiente y lo devuelve al cliente a través del método de fábrica.

Consecuencias

- Promueve el principio de encapsulamiento, ya que oculta la creación del objeto del cliente.
- Facilita la introducción de nuevas clases de productos sin modificar el código existente.
- Puede resultar en la proliferación de subclases si se necesitan crear muchos tipos diferentes de productos.

Implementación

1. Product es una interfaz o una clase abstracta con las operaciones que deben ser soportadas por los objetos creados.
2. Implementa un método de fábrica en Creator que devuelva instancias de Product. Puede ser un método abstracto o con una implementación por defecto que puede ser anulada por subclases.
3. Crea una o más subclases ConcreteCreator que implementen el método de fábrica para crear instancias de ConcreteProduct.

Código de ejemplo

```java
// Product
public interface Pizza {
    void prepare();
    void bake();
    void cut();
    void box();
}

// ConcreteProduct
public class CheesePizza implements Pizza {
    // Implementación de los métodos
}

// Creator
public abstract class PizzaStore {
    public Pizza orderPizza() {
        Pizza pizza = createPizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    protected abstract Pizza createPizza();
}

// ConcreteCreator
public class NYStylePizzaStore extends PizzaStore {
    protected Pizza createPizza() {
        return new NYStyleCheesePizza();
    }
}
```

Usos conocidos

Frameworks de desarrollo de juegos, donde se crean instancias de objetos según las necesidades del juego.

Patrones relacionados

Prototype: se utiliza para crear nuevos objetos duplicándolos de un prototipo existente. En algunos casos, el Factory Method puede utilizarse en combinación con el Prototype para crear instancias de objetos prototipo.

Decorator

Intención

Agregar dinámicamente responsabilidades adicionales a un objeto. Los decoradores proporcionan una alternativa flexible a la derivación de subclases para extender la funcionalidad

Conocido como

Wrapper

Motivo

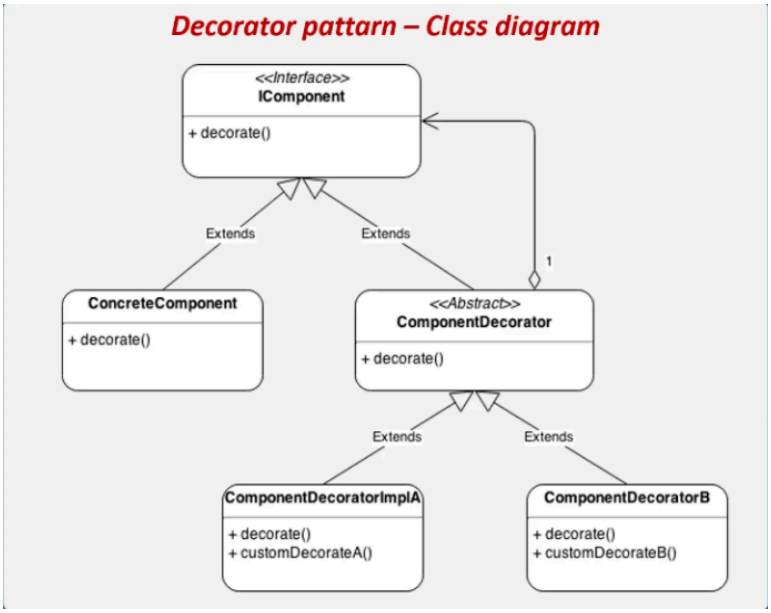
La necesidad de extender las funcionalidades de una clase en tiempo de ejecución sin modificar el código existente, manteniendo la funcionalidad separada para promover la

cohesión y cumplir con el principio de responsabilidad única.

Aplicaciones

- Cuando se quiere añadir responsabilidades a objetos individuales dinámicamente y de manera transparente, es decir, sin afectar a otros objetos.
- Para extensiones que pueden ser retiradas.
- Cuando la extensión mediante herencia es impracticable debido a la gran cantidad de extensiones posibles, o porque la extensión debe ser posible en tiempo de ejecución.

Estructura



Participantes

- Componente (Interfaz o clase abstracta): el objeto original a ser decorado.
- ComponenteConcreto: implementación del Componente, el objeto que se decorará.
- Decorador: mantiene una referencia al Componente y conforma la interfaz del Componente.
- DecoradorConcreto: añade responsabilidades al componente.

Colaboraciones

El flujo de trabajo general en un diseño Decorator es que el usuario utiliza el Componente Concreto, y puede decorar este componente con uno o más Decoradores Concretos para añadir dinámicamente funcionalidades. Los decoradores "envuelven" el componente, cada uno agregando su propia funcionalidad antes o después de pasar las llamadas al componente o decorador "envuelto" dentro de sí.

Consecuencias

Flexibilidad Mejorada

Al permitir la adición dinámica de responsabilidades a objetos sin modificar las clases

existentes, el patrón Decorator fomenta una mayor flexibilidad en el desarrollo de software. Esta característica es particularmente valiosa en sistemas donde la extensibilidad en tiempo de ejecución es esencial.

Adherencia al Principio de Responsabilidad Única

Facilita la conformidad con el principio de responsabilidad única, ya que cada decorador encapsula su propia funcionalidad, separando claramente las preocupaciones. Esto resulta en un diseño más limpio y fácil de mantener.

Aumento de la Complejidad

Mientras que el patrón promueve la extensibilidad, también puede llevar a una proliferación de pequeños objetos decoradores, lo que potencialmente complica la arquitectura del sistema. La gestión de esta complejidad requiere una planificación cuidadosa y puede aumentar la curva de aprendizaje para los nuevos desarrolladores del proyecto.

Riesgo de Romper el Encapsulamiento

Aunque los decoradores pueden añadir funcionalidades de manera eficiente, también existe el riesgo de romper el encapsulamiento de objetos, ya que los decoradores necesitan acceso a la interfaz interna del objeto decorado. Es crucial diseñar cuidadosamente las interfaces y las interacciones entre decoradores y componentes para mitigar este riesgo.

Implementación

La implementación efectiva del patrón Decorator implica varias consideraciones clave para asegurar su eficiencia en sistemas de software complejos:

Diseño de Interfaz de Componente

La interfaz del componente debe ser cuidadosamente diseñada para garantizar que sea lo suficientemente general para permitir la decoración pero específica para proporcionar funcionalidad útil. Esto incluye la definición de operaciones comunes que los decoradores pueden extender o modificar.

Compatibilidad entre Componentes y Decoradores

Asegurar la compatibilidad entre componentes y decoradores es esencial para la flexibilidad del patrón. Esto se logra mediante la implementación de una interfaz común que permita a los decoradores envolver cualquier instancia de componente concreto, ofreciendo así extensibilidad sin modificar las clases existentes.

Implementación de Decoradores Concretos

Los decoradores concretos deben implementarse con atención a los detalles, asegurando que cada uno añada o modifique la funcionalidad del componente de manera coherente y predecible. La claridad en la implementación y la documentación es crucial para mantener la integridad del diseño.

Gestión de Memoria

En entornos de programación donde la recolección de basura no es automática, como C++, la gestión de memoria se convierte en una consideración crítica. Los desarrolladores deben asegurarse de que todos los objetos decoradores gestionen adecuadamente la memoria para evitar fugas, lo que puede requerir estrategias explícitas de liberación de memoria.

Transparencia de Uso

Aunque los decoradores añaden funcionalidades a los componentes, su uso debe permanecer transparente para el cliente del código, manteniendo la simplicidad en la interacción con el objeto compuesto. Esto se logra manteniendo una interfaz común entre decoradores y componentes.

Código de ejemplo

Se define la interfaz "Texto":

```
public interface Texto {
    String darFormato();
}
```

Se crea una implementación concreta a "Texto", que es un texto plano:

```
public class TextoPlano implements Texto {
    private String texto;

    public TextoPlano(String texto) {
        this.texto = texto;
    }

    @Override
    public String darFormato() {
        return texto;
    }
}
```

El decorador abstracto implementa la interfaz "Texto" y contiene una referencia al objeto "Texto" que está decorando:

```
public abstract class DecoradorTexto implements Texto {
    protected Texto textoDecorado;

    public DecoradorTexto(Texto textoDecorado) {
        this.textoDecorado = textoDecorado;
    }

    public String darFormato() {
        return textoDecorado.darFormato();
    }
}
```

Se crean decoradores concretos para añadir funcionalidades como negrita y cursiva al texto:

```
public class NegritaDecorator extends DecoradorTexto {
    public NegritaDecorator(Texto textoDecorado) {
```

```
        super(textoDecorado);
    }

    @Override
    public String darFormato() {
        return "<b>" + super.darFormato() + "</b>";
    }
}

public class CursivaDecorator extends DecoradorTexto {
    public CursivaDecorator(Texto textoDecorado) {
        super(textoDecorado);
    }

    @Override
    public String darFormato() {
        return "<i>" + super.darFormato() + "</i>";
    }
}
```

Usos conocidos

- Interfaces gráficas de usuario donde los componentes pueden ser decorados con propiedades como bordes, sombras, etc.
- Aplicaciones de streaming de datos, donde los datos pueden ser "decorados" con encabezados o metadatos adicionales antes de ser enviados.

Patrones relacionados

- Adapter: Mientras que Decorator añade responsabilidades a objetos, Adapter convierte la interfaz de un objeto para que sea lo que otro espera.
- Composite: Se puede usar Decorator para añadir responsabilidades a objetos individuales en una estructura de Composite.
- Strategy: Decorator permite cambiar el aspecto de un objeto, mientras Strategy permite cambiar el núcleo de su comportamiento.

Visitor

Intención

El permitir separar tareas o algoritmos de los objetos sobre los que operan.

Conocido como

Visitante

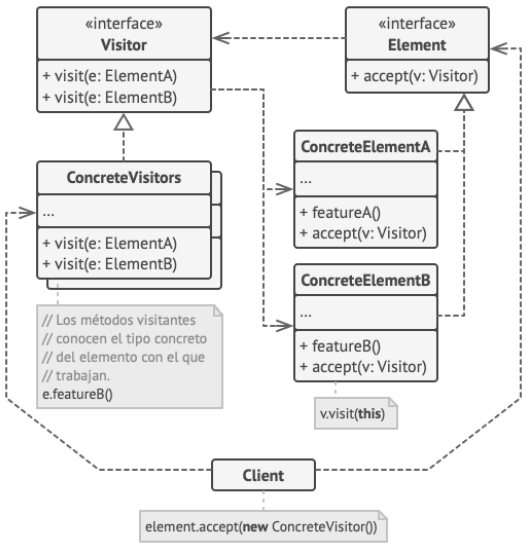
Motivo

Evitar potenciales errores al momento de exportar a XML

Aplicaciones

- Cuando se necesite realizar una operación sobre todos los elementos de una compleja estructura de objetos.
- Para limpiar la lógica de negocio de comportamientos auxiliares.

Estructura



Participantes

1. Se declara un grupo de métodos visitantes que pueden tomar elementos concretos de una estructura de objetos.
2. Cada Visitante Concreto implementa varias versiones de los mismos comportamientos
3. La interfaz Elemento declara un método para “aceptar” visitantes.
4. Cada elemento concreto debe redirigir la llamada al método adecuado del visitante correspondiente a la clase de elemento actual.
5. El Cliente representa normalmente una colección o algún otro objeto complejo

Colaboraciones

Las clases de elementos se ayudan de la interfaz Visitor de manera que Visitor declara métodos ‘visit’ para cada tipo de elemento así se definen cómo se realizan las operaciones de cada elemento. De manera que estos interactúan dinámicamente pero antes se debe invocar el método accept para cada elemento y así iniciar el proceso de visita.

Consecuencias

Los cambios más pequeños en la clase de un elemento suelen requerir adaptaciones de las clases Visitor. Además, se requiere un trabajo adicional para introducir posteriormente nuevos elementos

Implementación

Para la implementación se debe tomar en cuenta lo siguiente:

Separación de algoritmo y estructura

Visitor promueve la separación de los algoritmos de la estructura de elementos.Esto facilita la introducción de nuevos algoritmos sin modificar las clases de los elementos existentes.

Consideraciones Prácticas

Visitor opera en tiempo de ejecución, por lo tanto, se consideran métodos que se invocan dinámicamente en la ejecución, lo cual puede penalizar el rendimiento pero ofrece mayor flexibilidad.

Gestión de Memoria

Visitor puede ayudar a reducir el acoplamiento entre las clases de elementos y los visitantes.

Compatibilidad

Agregar nuevos tipos de elementos no requiere modificar los visitantes existentes. Por lo tanto, nuevos visitantes pueden ofrecer específicamente una tarea sin afectar las demás clases.

Código de ejemplo

```
package refactoring_guru.visitor.example.visitor;

//ejemplos externos
import refactoring_guru.visitor.example.shapes.Circle;
import refactoring_guru.visitor.example.shapes.CompoundShape;
import refactoring_guru.visitor.example.shapes.Dot;
import refactoring_guru.visitor.example.shapes.Rectangle;

public interface Visitor {
    String visitDot(Dot dot);

    String visitCircle(Circle circle);

    String visitRectangle(Rectangle rectangle);

    String visitCompoundGraphic(CompoundShape cg);
}
```

Usos conocidos

- Se utiliza el patrón cuando un comportamiento sólo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

Patrones relacionados

- Se puede relacionar con el patrón Command de manera que las operaciones pueden ser ejecutadas sobre varios objetos en distintas clases.
- Composite porque con visitor podemos ejecutar una operación sobre un árbol completo.
- Iterator para recorrer una estructura de datos compleja y ejecutar alguna operación

Informe gestión de tiempo

Fecha y Hora de Inicio
2 de marzo, 8:00 AM

Fecha y Hora de Finalización
6 de marzo, 5:00 PM

Desglose del Proceso

- Investigación Preliminar: Las primeras 4 horas se dedicaron a obtener una comprensión básica de cada patrón de diseño.
- Recopilación de Información Detallada: Se emplearon 2 días en reunir información detallada sobre cada patrón, incluidos ejemplos de código y aplicaciones.
- Elaboración de la Plantilla: La redacción de la plantilla del informe tomó 1 día, enfocándose en estructurar coherentemente la información de los tres patrones.

Conclusiones

Aspectos Positivos
- Se demostró un amplio rango de conocimientos al investigar diferentes patrones, lo que enriqueció la calidad del informe.
- La capacidad para adaptarse y alternar entre los diferentes requerimientos y enfoques de cada patrón fue notable.
- La investigación permitió una integración efectiva de los aprendizajes, donde las comparaciones y contrastes entre los patrones enriquecieron la comprensión del equipo.

Aspectos a Mejorar

- A pesar de completar la tarea, es evidente la necesidad de mejorar en la estimación de tiempos para investigaciones futuras, especialmente cuando se abordan múltiples temas.
- La coordinación entre los miembros del equipo trabajando en diferentes patrones requiere mejoras para maximizar la eficiencia y evitar duplicidades de esfuerzo.

Recomendaciones para Futuros Proyectos

Implementar reuniones de revisión de avance más frecuentes, permitiendo ajustes ágiles y una distribución más equitativa del trabajo.

Referencias

Canelo, M. M. (2023, 5 septiembre). ¿Qué son los patrones de diseño de software? Profile Software Services. https://profile.es/blog/patrones-de-diseno-de-software/#Singleton

Decorator. (s. f.). https://refactoring.guru/es/design-patterns/decorator

Refactoring Guru. (s.f.). Patrón de Diseño Decorador. Recuperado de https://refactoring.guru/es/design-patterns/decorator

IONOS. (19 de febrero de 2021). Patrón decorador: Un patrón para la expansión dinámica de clases. Recuperado de https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/patron-decorador-explicacion/

Pentalog. (s.f.). Desmitificando el Patrón de Diseño Decorador: Una Guía Completa para Decorar tu Código. Recuperado de https://www.pentalog.es/blog/patron-de-diseno-decorador-desmitificado