Ingeniería de Software 1
Sección 20
Grupo 3
Santiago Pereira- 22318
Nancy Mazariegos- 22513
Mónica Salvatierra- 22249
Hugo Rivas- 22500
Mauricio Lemus- 22461
Giovanni Santos- 22523

Tarea Investigativa 1
Patrones de Diseño

Creacional

- **Factory Method**

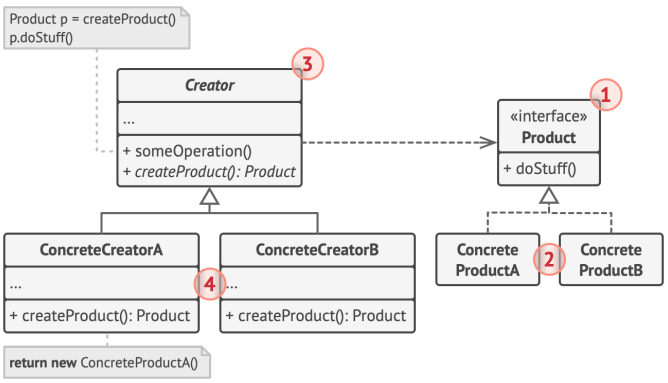
Intención:
- Tiene como objetivo la creación de objetos cuyas subclases pueden alterar su tipo, esto al invocar el factory method.
- Se define una interfaz o clase abstracta para crear un objeto el cual las subclases deciden que clase instanciar.

Conocido como:
- constructor virtual
- generador de fábrica.

Motivo:
- Se considera como un patrón, ya que permite la extensibilidad del código, además de solucionar un problema en la creación de objetos.
- También porque permite el desacoplamiento de clases, permitiendo que clases con aplicación específica no sean vinculadas.

Aplicaciones:
- Se puede utilizar cuando no se sabe qué subclases debe crear una clase.
- Cuando la clase necesita que las subclases especifiquen los objetos que deben ser creados.
- Para prevenir que código de creación de objetos sea duplicado.

Estructura:



Participantes:

- Product**:** declara la interfaz, igual para los objetos.
- ConcreteProduct: implementaciones de la interfaz y definición de objetos
- Creator: clase que declara el Factory Method cuyo tipo de retorno debe ser igual que la interfaz Product.
- ConcreteCreator: implementación del Factory Method y puede devolver tipos diferentes de Product.

Colaboraciones:

- ‘Creator’ va a depender de sus subclases para definir el método ‘factory’ que va a retornar el producto deseado.
- Los ‘Content Creators’ implementan el ‘factoryMethod’ y son responsables de crear, ya sea uno o más productos concretos.
- ‘Product’ es una interfaz o clase abstracta que define la forma de los objetos que el método ‘factoryMethod’ debe crear.
- ‘ConcereteProduct’ son las clases concretas que implementan ‘Product’, representando así los objetos específicos que van a ser creados dentro de él.

Consecuencias:

Beneficios:

- Flexibilidad en la creación de objetos, puesto que al ser creados dentro de una clase de ‘factoryMethod’, elimina la necesidad de la aplicación a una clase de productos concretos.
- Se pueden introducir nuevas variantes de productos, sin alterar el código ya existente.

Riesgos:

- Puede llegar a incrementar la complejidad del código al introducir múltiples subclases.

Implementación:

1. Definir una interfaz común que declare métodos comunes y que tengan sentido para todos los productos.

2. Definir un método de clase creadora que actúe como el Factory Method. El tipo de retorno debe de coincidir con la interfaz común de los productos.

3. Encontrar todas las llamadas a constructores de productos en el código de la clase creadora y reemplazarlas por invocaciones al Factory Method. Mueve el código de creación de productos dentro del Factory Method.

4. Si hay muchos tipos de productos, se pueden crear subclases creadoras para cada uno. Estas subclases sobrescribirán el Factory Method y crearán instancias del tipo de producto correspondiente.

5. Si el Factory Method base tiene un largo switch para elegir qué clase de producto instanciar, se puede refactorizar. Puedes mover las partes relevantes del código constructor a las subclases creadoras o reutilizar un parámetro de control para determinar qué tipo de producto crear.

6. Si después de las extracciones el Factory Method base queda vacío, se puede volverlo abstracto. Si aún queda algo dentro, puedes convertirlo en un comportamiento por defecto del método.

Código de Ejemplo:

```
from abc import ABC, abstractmethod

# Interfaz común para todos los productos
class Product(ABC):
    @abstractmethod
    def describe(self):
        pass

# Implementación de productos
class Phone(Product):
    def describe(self):
        print("Este es un teléfono.")

class Laptop(Product):
    def describe(self):
        print("Esta es una laptop.")

# Clase creadora base con Factory Method
class Store(ABC):
    @abstractmethod
    def create_product(self):
        pass

    def sell_product(self):
        product = self.create_product()
        product.describe()

# Subclases creadoras para cada tipo de producto
class PhoneStore(Store):
    def create_product(self):
        return Phone()

class LaptopStore(Store):
    def create_product(self):
        return Laptop()

# Ejemplo de uso
if __name__ == "__main__":
    phone_store = PhoneStore()
    phone_store.sell_product()

    laptop_store = LaptopStore()
    laptop_store.sell_product()
```

Usos conocidos:

- En los motores de juego como Unity o Unreal Engine, el Factory

Method se utiliza para crear instancias de diferentes tipos de objetos del juego, como personajes, armas o elementos del entorno.

- En aplicaciones de procesamiento de texto como Microsoft Word o Google Docs, se utiliza el Factory Method para crear instancias de diferentes tipos de documentos, como documentos de texto, hojas de cálculo o presentaciones.

- En frameworks como Django o Flask en Python, el Factory Method se utiliza para crear instancias de diferentes tipos de objetos web, como vistas, modelos o formularios.

Patrones relacionados:

Abstract Factory: Es más flexible pero más complicado que Factory Method. Define una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Puede basarse en un grupo de métodos de fábrica, o también puede utilizar Prototype para escribir los métodos de estas clases.

Prototype: No se basa en la herencia y no presenta sus inconvenientes. Permite la creación de nuevos objetos duplicando un prototipo existente, lo que puede ser útil cuando la inicialización del objeto clonado es complicada. Se puede utilizar en conjunto con Factory Method para definir los métodos de creación en las clases de Abstract Factory.

Builder: Es más flexible que Factory Method y se utiliza para construir objetos complejos paso a paso. Permite la construcción de objetos con diferentes representaciones utilizando el mismo proceso de construcción. Se puede utilizar en conjunción con Factory Method para crear partes de un objeto complejo.

Iterator: Puede utilizarse junto con Factory Method para permitir que las subclases de una colección devuelvan diferentes tipos de iteradores compatibles con la colección.

Template Method: Factory Method es una especialización de Template Method, ya que define un esqueleto de algoritmo en una superclase con pasos que las subclases pueden implementar. Un Factory Method puede servir como pasó en un gran Template Method, ya que puede definir uno de los pasos para crear un objeto en el esqueleto de algoritmo.

Estructural

Patrón: Decorator

Intención

- Agregar un comportamiento a objetos individuales de manera dinámica sin alterar su estructura básica.
- Permite la extensión de funcionalidades sin modificar las clases
- Promueve la separación de preocupaciones

Conocido como

- Wrapper

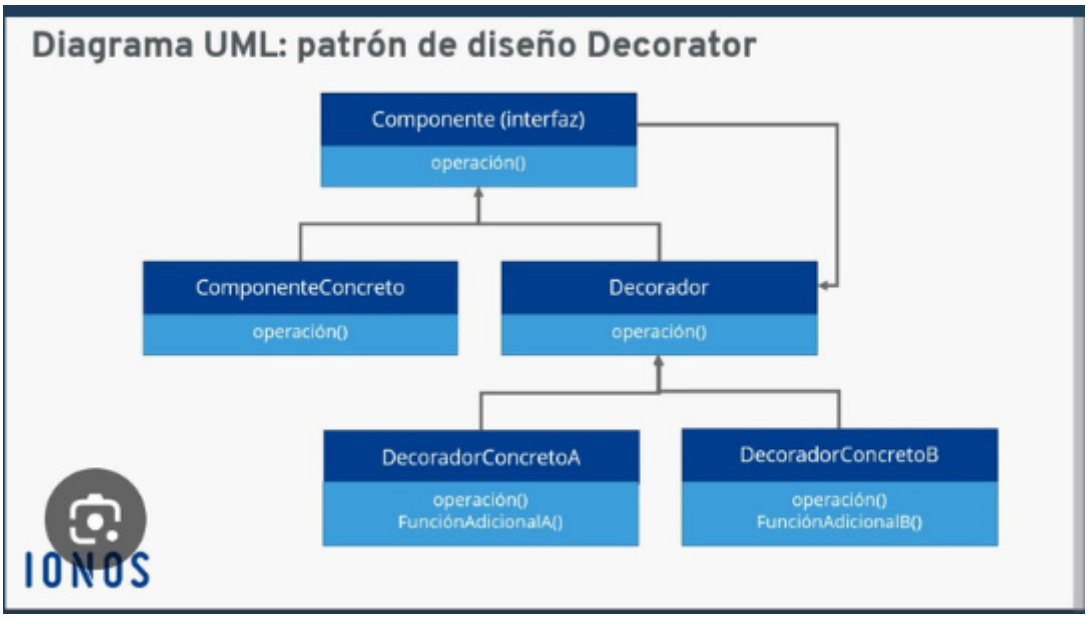
Motivo

- Este patrón de diseño es útil cuando se necesita agregar funcionalidad a objetos de forma dinámica y que no termine alterando la estructura. Lo que promueve la reutilización de código y gestionar comportamiento adicionales de manera modular.

Aplicaciones

- Agregar responsabilidades adicionales a objetos sin alterar su código.
- Extender funcionalidades de clases existentes de forma dinámica
- Implementar diferentes variantes de un objeto sin crear subclases.

Estructura



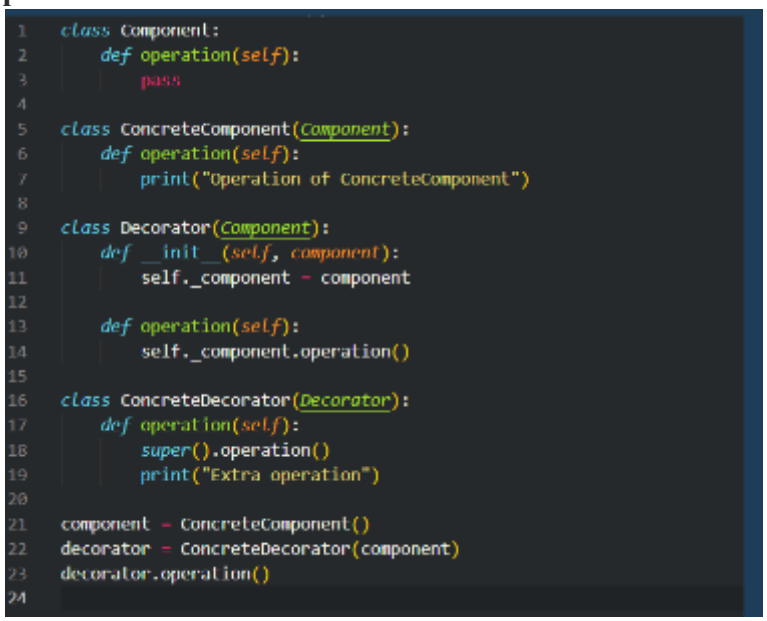
Participantes

- Component: Define la interfaz para los objetos a los que se puede agregar responsabilidades dinámicamente.
- Decorator: Mantiene una referencia a un objeto Component y define una interfaz que se ajusta a la interfaz del Component.
- ConcreteDecorator: Extiende la funcionalidad de un objeto Component.

Colaboraciones

El Decorator reenvía las solicitudes al Component y puede realizar operaciones adicionales antes o después de reenviar la solicitud.

Consecuencias
Beneficios: Permite agregar funcionalidades de forma dinámica y modular, favorece la reutilización de código y la separación de preocupaciones.
Riesgos: Puede resultar en una explosión de clases si se crean demasiados Decorators.
Implementación
Utiliza la composición en lugar de la herencia para extender funcionalidades.
Los Decorators pueden anidarse para agregar múltiples capas de funcionalidad.
Código ejemplo



```python
class Component:
    def operation(self):
        pass

class ConcreteComponent(Component):
    def operation(self):
        print("Operation of ConcreteComponent")

class Decorator(Component):
    def __init__(self, component):
        self._component = component

    def operation(self):
        self._component.operation()

class ConcreteDecorator(Decorator):
    def operation(self):
        super().operation()
        print("Extra operation")

component = ConcreteComponent()
decorator = ConcreteDecorator(component)
decorator.operation()
```

Usos conocidos

- Java I/O classes utilizan el patrón Decorator para proporcionar funcionalidades adicionales como encriptación, compresión, etc.
- Bibliotecas gráficas como Swing en Java utilizan Decorator para agregar funcionalidades de visualización a los componentes gráficos.

P. Relacionados

- Adapter: Se utiliza para hacer que interfaces incompatibles sean compatibles.
- Composite: Se puede usar junto con Decorator para permitir la composición de objetos complejos.
- Proxy: Controla el acceso a objetos, pudiendo agregar funcionalidades adicionales.

Comportamiento

- **Visitor**

Intención:

- Este patrón de arquitectura tiene como intención u objetivo el poder agregar nuevos algoritmos a algoritmos ya existentes sin cambiar la estructura de los que ya existen y están definidos.

Conocido como:

- Visita
- Visitor

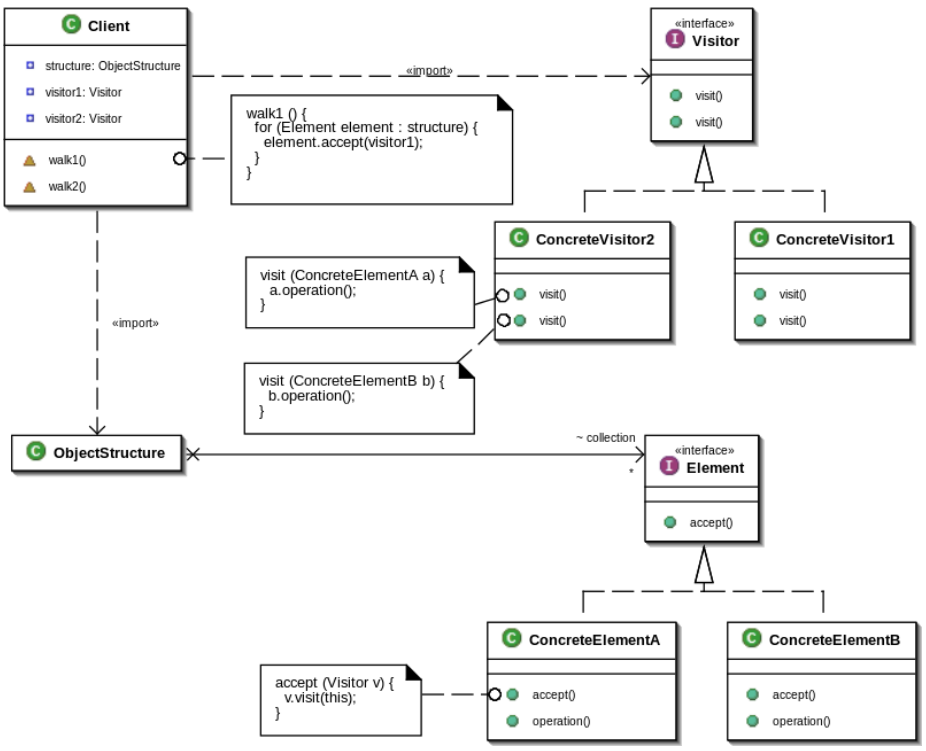
Motivo:

- Se le considera como un patrón ya que permite la escalabilidad de los algoritmos, agregando nuevos métodos sin perjudicar las estructuras ya establecidas.

Aplicaciones:

- Cuando se tiene una estructura de objetos compleja, como un árbol o un grafo, el patrón Visitor facilita el recorrido de la estructura y la realización de operaciones específicas en cada elemento.
- Puede utilizarse para realizar análisis o validaciones en una estructura de objetos. Cada clase puede implementar su propia versión de la operación definida por el Visitor, permitiendo realizar acciones específicas según el tipo de objeto.
- Permite optimizar operaciones específicas al centralizar la lógica en un objeto Visitor. Esto facilita la mejora del rendimiento y la gestión de cambios en dichas operaciones.

Estructura:



Participantes:

- Elemento: Define una interfaz que permite que el visitante acceda a sus elementos.
- Visitante: Declara una operación de visita para cada tipo de Elemento Concreto en la estructura.

Colaboraciones:

- Cuando el Visitante visita un Elemento, el Elemento llama al método de visita correspondiente en el Visitante.
- Cuando un Elemento llama al método de aceptación en el Visitante, este redirige la llamada al método de visita adecuado en el Visitante Concreto.

Consecuencias:

Beneficios:

- Permite separar el código que opera sobre una estructura de objetos del código que define la estructura misma.
- Facilita el mantenimiento al agrupar operaciones relacionadas en un único visitante.

Riesgos:

- Introducir el patrón Visitor puede hacer que el código sea más difícil de entender debido a la introducción de una capa adicional de abstracción.
- Existe el riesgo de exponer demasiados detalles de implementación de las clases de Elemento a los Visitantes, lo que puede hacer que el código sea más frágil.

Implementación:

1. Define la estructura de los elementos sobre los cuales se realizarán operaciones.
2. Crea una interfaz que describa los comportamientos de los visitantes.
3. Implementa clases que representen los visitantes y definan cómo operan sobre los elementos.
4. Crea clases que representen los elementos y les permitan aceptar visitantes.
5. Utiliza las clases de elementos y visitantes para realizar las operaciones deseadas, asegurándose de que los visitantes accedan a los elementos y realicen las operaciones adecuadas.

Código de ejemplo:

```
# Interfaz del Elemento

class Shape:

def accept(self, visitor):

        pass


# Elemento Concreto: Círculo

class Circle(Shape):

    def __init__(self, radius):

        self.radius = radius


    def accept(self, visitor):

        visitor.visit_circle(self)


# Elemento Concreto: Cuadrado

class Square(Shape):

    def __init__(self, side):

        self.side = side


    def accept(self, visitor):

        visitor.visit_square(self)
```

```
# Interfaz del Visitante
class ShapeVisitor:
    def visit_circle(self, circle):
        pass

    def visit_square(self, square):
        pass

# Visitante Concreto: Calculadora de Área
class AreaCalculator(ShapeVisitor):
    def __init__(self):
self.area = 0

    def visit_circle(self, circle):
        self.area += 3.14 * circle.radius * circle.radius

    def visit_square(self, square):
        self.area += square.side * square.side

# Visitante Concreto: Calculadora de Perímetro
class PerimeterCalculator(ShapeVisitor):
    def __init__(self):
        self.perimeter = 0
```

```
    def visit_circle(self, circle):
        self.perimeter += 2 * 3.14 * circle.radius

    def visit_square(self, square):
        self.perimeter += 4 * square.side

# Uso del patrón Visitor
def main():
    shapes = [Circle(5), Square(3)]
    area_calculator = AreaCalculator()
    perimeter_calculator = PerimeterCalculator()

    for shape in shapes:
        shape.accept(area_calculator)
        shape.accept(perimeter_calculator)

    print("Área total:", area_calculator.area)
    print("Perímetro total:", perimeter_calculator.perimeter)

if __name__ == "__main__":
    main
```

Usos conocidos:

- **Análisis y transformación de árboles sintácticos**: En compiladores y analizadores sintácticos, el patrón Visitor se utiliza para realizar análisis semántico, optimizaciones y generación de código. Cada nodo del árbol sintáctico puede ser visitado por diferentes visitantes para realizar diversas operaciones.
- **Operaciones sobre estructuras de datos complejas**: En aplicaciones que manejan estructuras de datos complejas como árboles, grafos o estructuras compuestas, el patrón Visitor se utiliza para realizar operaciones como búsqueda, filtrado, cálculos, etc., sin modificar las clases de los elementos de la estructura.

Patrones relacionados:

Iterator:El patrón Iterator podría relacionarse con el patrón Visitor. Pueden trabajar juntos cuando se trata de recorrer una estructura de objetos y realizar operaciones específicas en cada uno. El Iterator podría proporcionar la forma de recorrer los elementos, y el Visitor podría definir las operaciones a realizar en cada elemento durante el recorrido.

Observer: El patrón Observer podría relacionarse con el patrón Visitor en el sentido de que ambos patrones se utilizan para manejar cambios en un objeto. Mientras que el Observer se centra en notificar a los observadores sobre cambios de estado, el Visitor se centra en realizar operaciones específicas en una estructura de objetos.

Strategy: El patrón Strategy podría estar relacionado en situaciones donde el patrón Visitor necesita aplicar diferentes estrategias o algoritmos en la estructura de objetos. La implementación de Visitor podría usar diferentes estrategias según sea necesario.

Template Method: El patrón Template Method se menciona específicamente como relacionado. El Factory Method, que es un tipo de patrón de diseño, se considera una especialización del Template Method. Puede haber similitudes en cómo estos patrones estructuran el código, especialmente en la definición de esqueletos de algoritmos.

Command: El patrón Command podría relacionarse con Visitor si los comandos deben realizarse sobre una estructura de objetos específica. El Visitor podría actuar como un comando que se ejecuta en objetos particulares durante un recorrido.

Chain of Responsibility: El patrón Chain of Responsibility podría relacionarse en situaciones donde el patrón Visitor se utiliza para procesar una cadena de elementos. Cada elemento podría manejar una parte de la operación, y la cadena de responsabilidad puede pasar la solicitud de manejar al siguiente elemento en la cadena.

Gestión del tiempo:

Nombre: Hugo Eduardo Rivas Fajardo
Carné: 22500

Fecha	Inicio	Fin	Tiempo Interrupción	Delta tiempo	Fase	Comentarios
1/03/2024	17:30	18:46	10 mins	50 mins	Tarea Investigativa 1	Realicé parte del patrón Decorator

Nombre: José Santiago Pereira Alvarado
Carné: 22318

Fecha	Inicio	Fin	Tiempo Interrupción	Delta tiempo	Fase	Comentarios
1/03/2024	17:00	18:15	5 mins	65 mins	Tarea Investigativa 1	Realicé parte del patrón visitor

Nombre: Mónica Alejandra Salvatierra Chacón
Carné: 22249

Fecha	Inicio	Fin	Tiempo Interrupción	Delta tiempo	Fase	Comentarios
29-2-24	23:20	00:30	25 minutos	1:45 hrs	Tarea Investigativa 1	Realicé parte de la investigación para el Factory Method

Nombre: Nancy Gabriela Mazariegos Molina
Carné: 22513

Fecha	Inicio	Fin	Tiempo Interrupción	Delta tiempo	Fase	Comentarios
1-3-24	17:20	18:20	20 minutos	40 minutos	Tarea investigativa 1	Realicé la parte del comportamiento

Nombre: Giovanni Alejandro Santos Hernández
Carné: 22523

Fecha	Inicio	Fin	Tiempo Interrupción	Delta tiempo	Fase	Comentarios
1-03-24	11:30	12:05 pm	5	30	Tarea investigativa 1	Se realizó parte de la investigación de Factory Method.

Nombre: Mauricio Julio Rodrigo Lemus Guzman
Carné: 22461

Fecha	**Inicio**	**Fin**	**Tiempo Interrupción**	**Delta tiempo**	**Fase**	**Comentarios**
1/03/2024	17:20	18:30	10 mins	60 mins	Tarea Investigativa 1	Realicé parte del patrón Decorator

Link de la presentación:

https://www.canva.com/design/DAF-TMnbZ5Q/4Q6Bqy8Bqi3sGEhvE9kDaA/edit?utm_content=DAF-TMnbZ5Q&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Link del documento:
https://docs.google.com/document/d/12K_q2YyXQohe6kyCJNhchaHTsybiTdM8VB75VkoqeEs/edit?usp=sharing