Universidad del Valle de Guatemala

Facultad de Ingeniería

Ingeniería en Software

PROYECTO SANITAS

Tarea Investigativa 1

Patrones de diseño

Carnet/Autores:

22272, Bianca Renata Calderón Caravantes

22473, Madeline Nahomy Castro Morales

22716, Aroldo Xavier López Osoy

22233, Daniel Eduardo Dubon Ortiz

22386, Flavio André Galán Donis

Catedrático:

Cristián Muralles

Sección 20

Grupo 05

Fecha:

01/03/2024

Patrón de Diseño Creacional: Factory

- **Intención**: Proporcionar una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que serán creados.

- **Conocido como**: Factory Method, Método Fábrica.

- **Motivo**: Permite a una clase delegar en sus subclases la creación de objetos, lo que facilita la extensión de la funcionalidad de la aplicación modificando únicamente las subclases involucradas.

- **Aplicaciones**: Útil cuando se desconoce el tipo exacto de objetos a crear o se desea delegar la responsabilidad de la creación de objetos a clases especializadas.

- **Estructura**:
    - Creator: clase abstracta que declara el método fábrica.
    - ConcreteCreator: subclase que implementa el método fábrica para crear objetos específicos.
    - Product: interfaz para los objetos que crea el método fábrica.
    - ConcreteProduct: clase que implementa la interfaz Product.

- **Participantes**: Creator (Creador), ConcreteCreator (Creador Concreto), Product (Producto), ConcreteProduct (Producto Concreto).

- **Colaboraciones**: Los ConcreteCreators sobrescriben el método factory para retornar una instancia de ConcreteProduct.

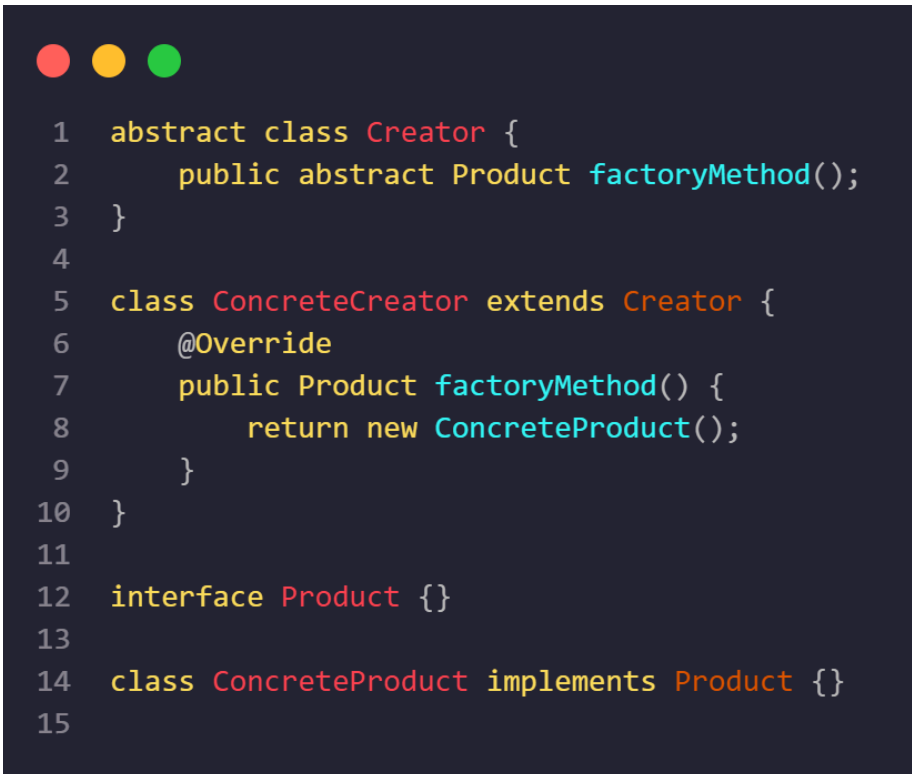
- **Consecuencias**: Promueve el desacoplamiento entre el cliente y las clases concretas. La extensión se hace más fácil, pero puede complicar el código al introducir varias subclases.

- **Implementación**: Se puede implementar con una interfaz o clase abstracta que define un método factory. Las subclases implementan este método para crear objetos.

- **Usos Conocidos**: Frameworks donde las librerías quieren permitir a los usuarios extender sus componentes internos.

- **Patrones Relacionados**: Abstract Factory, Builder, Prototype.

- **Código de Ejemplo:**



```java
abstract class Creator {
    public abstract Product factoryMethod();
}

class ConcreteCreator extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}

interface Product {}

class ConcreteProduct implements Product {}

```

Patrón de Diseño Estructural: Adapter

- **Intención**: Convertir la interfaz de una clase en otra interfaz que esperan los clientes. Adapter permite que clases con interfaces incompatibles trabajen juntas.

- **Conocido Como**: Wrapper, Envoltorio.

- **Motivo**: Permite la colaboración entre clases que de otro modo no podrían trabajar juntas debido a sus interfaces incompatibles.

- **Aplicaciones**: Cuando se quiere utilizar una clase existente, y su interfaz no se corresponde con la que necesitamos.

- **Estructura**:
    - Target: define la interfaz específica del dominio que utiliza el cliente.
    - Adapter: adapta la interfaz Adaptee a la interfaz Target.
    - Adaptee: define una interfaz existente que necesita ser adaptada.

- **Participantes**: Target, Adapter, Adaptee.

- **Colaboraciones**: El cliente llama a una operación en el Adapter utilizando la interfaz Target. El Adapter traduce esa llamada a una o más llamadas en la interfaz Adaptee.

- **Consecuencias**: Permite reutilizar clases existentes. Introduce solo un objeto, y no requiere cambiar el comportamiento existente.

- **Implementación**: Puede ser implementado extendiendo tanto la clase Target como la clase Adaptee o utilizando la composición para referenciar a Adaptee desde Adapter.

- **Usos Conocidos**: Integración de clases que no se pueden modificar para cumplir con interfaces específicas.

- **Patrones Relacionados**: Bridge, Decorator.

- **Código de Ejemplo**:

```java
// Interfaz objetivo
interface Target {
    void request();
}

// Clase a adaptar
class Adaptee {
    void specificRequest() {
        // Implementación específica
    }
}

// Adaptador
class Adapter implements Target {
    private Adaptee adaptee = new Adaptee();

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

```

Patrón de Diseño Estructural: Strategy

- **Intención:** Definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Strategy permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

- **Conocido Como:** Policy.

- **Motivo:** Permite seleccionar el algoritmo de comportamiento en tiempo de ejecución, proporcionando una alternativa flexible a la herencia para seleccionar comportamientos.

- **Aplicaciones:** Cuando hay varios algoritmos para realizar una tarea y se desea seleccionar dinámicamente cuál utilizar.

- **Estructura:**
    - **Context:** mantiene una referencia a una Strategy.
    - **Strategy:** interfaz común para todos los algoritmos soportados.
    - **ConcreteStrategy:** implementa los algoritmos usando la interfaz Strategy.

- **Participantes:** Context, Strategy, ConcreteStrategy.

- **Colaboraciones:** Context utiliza la interfaz Strategy para llamar al algoritmo definido por una ConcreteStrategy.

- **Consecuencias:** Proporciona una alternativa a la herencia para cambiar el comportamiento de una clase. Sin embargo, puede aumentar la complejidad del código al introducir múltiples clases y interfaces.

- **Implementación:** Se implementa definiendo una interfaz Strategy y derivando todas las variantes de esta interfaz.

- **Usos Conocidos:** Frameworks de ordenación donde los algoritmos de ordenación pueden variar.

- **Patrones Relacionados:** State, Command.

- **Código de Ejemplo:**

```
interface Strategy {
    void algorithmInterface();
}

class ConcreteStrategyA implements Strategy {
    public void algorithmInterface() {
        // Implementación del algoritmo
    }
}

class Context {
    private Strategy strategy;

    // Método para cambiar la estrategia en tiempo de ejecución
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy() {
        strategy.algorithmInterface();
    }
}

```

Anexos

Nombre: Bianca Renata Calderón Caravantes

Carné: 22272

Fecha	Inicio	Fin	Tiempo Interrupción	Delta Tiempo	Tarea	Comentarios
1/03/2024	16:00	16:32	0	32 minutos	Realización de presentación	—-
1/03/2024	16:50	17:00	0	10 minutos	Revisiones	—----

Nombre: Daniel Eduardo Dubon Ortiz

Carné: 22233

Fecha	Inicio	Fin	Tiempo Interrupción	Delta Tiempo	Tarea	Comentarios
1/03/2024	16:30	16:55	0	25 minutos	Realización de presentación	
1/03/2024	16:55	17:05	0	10 minutos	Revision Final	

Nombre: Madeline Nahomy Castro Morales

Carné: 22473

Fecha	Inicio	Fin	Tiempo Interrupción	Delta Tiempo	Tarea	Comentarios
2024-03-01	09:00	10:30	0	1h 30 min	Investigación de patrones de diseño	—
2024-03-01	10:45	12:00	15min	1h	Selección de patrones de diseño	—
2024-03-01	14:00	15:00	0	1h	Redacción	—
2024-03-01	15:00	16:20	20 min	1h 20 min	Codificación de Códigos de Ejemplo	—-

Nombre: Aroldo Xavier López Osoy

Carné: 22716

Fecha	Inicio	Fin	Tiempo Interrupción	Delta Tiempo	Tarea	Comentarios
2024-03-01	09:00	10:30	0	1h 30min	Investigación de Patrones de Diseño	Revisión de documentación y fuentes sobre patrones.
2024-03-01	10:45	12:00	15 min	1h	Selección de Patrones de Diseño	Decisión basada en requisitos del proyecto.
2024-03-01	13:00	15:00	0	2h	Redacción	Redacción de fichas y descripciones para cada patrón.
2024-03-01	15:15	17:00	15 min	1h 30min	Codificación de Códigos de Ejemplo	Implementación de ejemplos prácticos para cada patrón.

Nombre: Flavio André Galán Donis

Carné: 22386

Fecha	Inicio	Fin	Tiempo Interrupción	Delta Tiempo	Tarea	Comentarios
29/02/2024	16:20	17:30	10 mins	60 mins	Investigación de Patrones de Diseño	Se fue al baño en ese receso
1/03/2024	17:30	17:40	0 mins	10 mins	Retoques finales	