

PATRONES DE CREACIÓN

Abstract Factory

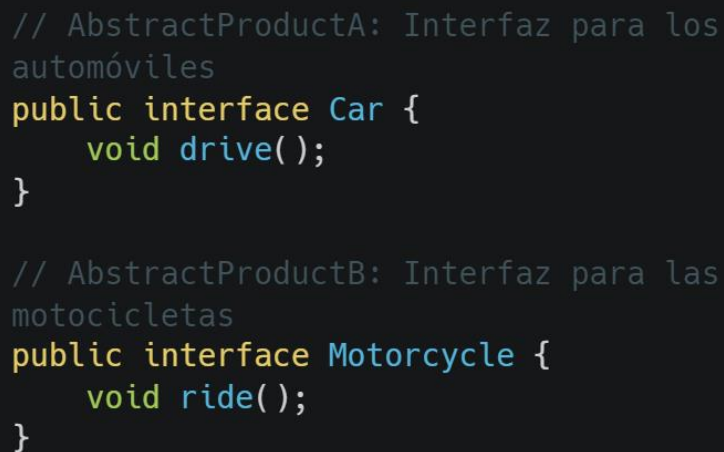
En qué consiste

- Es un patrón de diseño que se clasifica dentro de la categoría de patrones de creación. Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
- El objetivo principal del patrón es abstraer la creación de objetos y permitir la creación de familias de objetos relacionados sin acoplar el código a clases concretas específicas.
- Esto proporciona flexibilidad y extensibilidad al sistema, ya que se pueden introducir nuevas familias de objetos sin modificar el código existente.

Ejemplos

Tenemos un programa de simulación de una fábrica de vehículos y necesitamos crear diferentes tipos de vehículos, como automóviles y motocicletas, cada uno con sus variantes específicas (por ejemplo, deportivo, familiar, scooter, crucero, etc.).

1. Debemos definir las interfaces abstractas para los productos que se crearán:



```
// AbstractProductA: Interfaz para los
automóviles
public interface Car {
    void drive();
}

// AbstractProductB: Interfaz para las
motocicletas
public interface Motorcycle {
    void ride();
}
```

2. Luego, definimos las fábricas abstractas que crearán los objetos concretos

```
// AbstractFactory: Interfaz para la
// fábrica de vehículos
public interface VehicleFactory {
    Car createCar();
    Motorcycle createMotorcycle();
}
```

3. A continuación, implementamos fábricas concretas que crearán los objetos concretos correspondientes

```
// ConcreteFactory1: Fábrica para crear vehículos deportivos
public class SportsVehicleFactory implements VehicleFactory
{
    public Car createCar() {
        return new SportsCar();
    }

    public Motorcycle createMotorcycle() {
        return new SportsMotorcycle();
    }
}

// ConcreteFactory2: Fábrica para crear vehículos familiares
public class FamilyVehicleFactory implements VehicleFactory
{
    public Car createCar() {
        return new FamilyCar();
    }

    public Motorcycle createMotorcycle() {
        return new Scooter();
    }
}
```

4. Después, agregamos las clases concretas que representan los productos individuales

```
// ConcreteProductA1: Clase para un automóvil deportivo
public class SportsCar implements Car {
    public void drive() {
        System.out.println("Driving a sports car.");
    }
}

// ConcreteProductA2: Clase para un automóvil familiar
public class FamilyCar implements Car {
    public void drive() {
        System.out.println("Driving a family car.");
    }
}

// ConcreteProductB1: Clase para una motocicleta deportiva
public class SportsMotorcycle implements Motorcycle {
    public void ride() {
        System.out.println("Riding a sports motorcycle.");
    }
}

// ConcreteProductB2: Clase para un scooter
public class Scooter implements Motorcycle {
    public void ride() {
        System.out.println("Riding a scooter.");
    }
}
```

5. Finalmente, se utiliza la fábrica abstracta para crear objetos y trabajar con ellos sin conocer las clases concretas



```

public class Client {
    public static void main(String[] args) {
        VehicleFactory factory1 = new
SportsVehicleFactory();
        Car car1 = factory1.createCar();
        Motorcycle motorcycle1 =
factory1.createMotorcycle();

        car1.drive(); // Salida: "Driving a sports car."
        motorcycle1.ride(); // Salida: "Riding a sports
motorcycle."

        VehicleFactory factory2 = new
FamilyVehicleFactory();
        Car car2 = factory2.createCar();
        Motorcycle motorcycle2 =
factory2.createMotorcycle();

        car2.drive(); // Salida: "Driving a family car."
        motorcycle2.ride(); // Salida: "Riding a scooter."
    }
}

```

El cliente utiliza la fábrica abstracta para crear objetos concretos sin conocer las clases concretas involucradas. Esto proporciona flexibilidad, ya que el cliente puede cambiar entre diferentes familias de vehículos simplemente utilizando la fábrica correspondiente sin afectar su código.

En el ejemplo, se crean vehículos deportivos utilizando la fábrica SportsVehicleFactory y vehículos familiares utilizando la fábrica FamilyVehicleFactory. Luego, el cliente interactúa con los productos a través de las interfaces abstractas Car y Motorcycle, invocando métodos como drive() y ride(), respectivamente.

Funciones

- Abstracción de la creación de objetos: El patrón Abstract Factory proporciona una interfaz abstracta para crear familias de objetos relacionados sin especificar las clases concretas. Esto abstrae la lógica de creación de objetos y permite crear productos sin acoplar el código cliente a implementaciones concretas.
- Creación de familias de objetos relacionados: El patrón Abstract Factory permite crear una familia completa de objetos relacionados. Cada fábrica concreta es responsable de crear objetos de una variante específica dentro de la familia. Esto facilita la creación de objetos que están diseñados para trabajar juntos o seguir una temática común.
- Flexibilidad y extensibilidad: El uso del patrón Abstract Factory proporciona flexibilidad y extensibilidad al sistema. Al introducir una nueva fábrica concreta que implemente la interfaz abstracta, se pueden agregar nuevas variantes de productos sin modificar el código existente. Esto permite adaptar fácilmente el sistema a diferentes configuraciones o requisitos cambiantes.

Factory Method

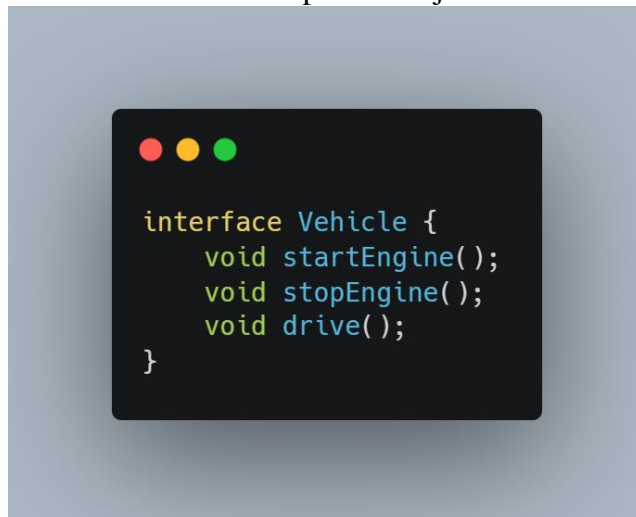
En qué consiste

- Es un patrón de diseño creacional en la programación orientada a objetos. Su objetivo principal es proporcionar una interfaz para la creación de objetos, pero permitiendo que las subclases decidan qué clase concreta instanciar.
- En términos más sencillos, el Factory Method proporciona un método en una clase base (la fábrica) que se encarga de crear objetos de diferentes tipos. Las subclases de la clase base implementan este método y deciden qué clase concreta crear y devolver.
- Este enfoque permite que el código cliente trabaje con la interfaz común proporcionada por la clase base, sin necesidad de conocer las clases concretas específicas que se están creando. Esto promueve la flexibilidad, el desacoplamiento y la extensibilidad en el diseño de software.

Ejemplos

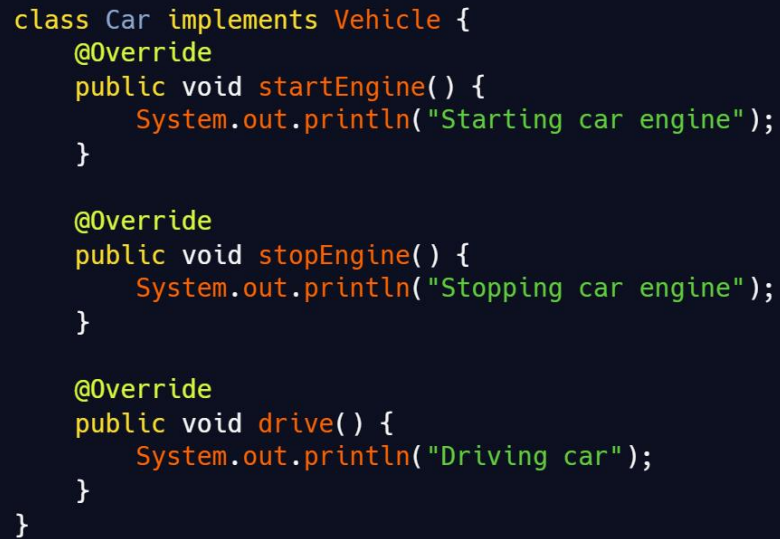
Tenemos una aplicación que maneja diferentes tipos de vehículos, como carros y motos. En lugar de crear una clase diferente para cada tipo de vehículo, podemos utilizar el patrón Factory Method para crear una interfaz “Vehicle” que define los métodos necesarios para manejar un vehículo. Luego, cada tipo de vehículo puede implementar esa interfaz y crear sus propios objetos.

- Interfaz que define los métodos necesarios para manejar un vehículo

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains the following code:

```
interface Vehicle {  
    void startEngine();  
    void stopEngine();  
    void drive();  
}
```

- Clase concreta para manejar un carro



```
class Car implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Starting car engine");
    }

    @Override
    public void stopEngine() {
        System.out.println("Stopping car engine");
    }

    @Override
    public void drive() {
        System.out.println("Driving car");
    }
}
```

- Clase concreta para manejar una moto

```

class Motorcycle implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Starting motorcycle engine");
    }

    @Override
    public void stopEngine() {
        System.out.println("Stopping motorcycle engine");
    }

    @Override
    public void drive() {
        System.out.println("Driving motorcycle");
    }
}

```

- Clase que utiliza el patrón Factory Method para crear objetos de tipo Vehicle

```

class VehicleFactory {
    public Vehicle createVehicle(String type) {
        switch (type) {
            case "car":
                return new Car();
            case "motorcycle":
                return new Motorcycle();
            default:
                return null;
        }
    }
}

```

- Utilización de la clase VehicleFactory



Como se puede ver, la clase VehicleFactory utiliza el patrón Factory Method para crear objetos de tipo Vehicle de acuerdo al tipo especificado en el método createVehicle. Esto permite manejar de forma genérica diferentes tipos de vehículos sin tener que crear una clase diferente para cada uno.

Otro ejemplo de uso del patrón Factory Method es en la creación de objetos de diferentes tipos de cajas de mensajes. Por ejemplo, podemos tener una interfaz MessageBox que define los métodos necesarios para mostrar una caja de mensajes, y luego crear subclases concretas para cajas de mensajes de alerta, información y error. La clase que utiliza el patrón Factory Method puede crear objetos de caja de mensajes de acuerdo al tipo especificado, lo que permite manejar de forma genérica diferentes tipos de cajas de mensajes.

En resumen, el patrón Factory Method es una técnica muy útil en la programación orientada a objetos ya que permite crear objetos de forma genérica sin tener que conocer la clase concreta del objeto. Esto facilita la creación de código modular y reusable.

Funciones

1. Encapsulación de la creación de objetos: El Factory Method encapsula la lógica de creación de objetos en una clase separada, permitiendo que el cliente interactúe con una interfaz común en lugar de conocer las clases concretas específicas que se están creando. Esto proporciona un nivel de abstracción y oculta los detalles de implementación.
2. Flexibilidad y extensibilidad: El Factory Method permite agregar nuevas variantes de productos o clases concretas sin modificar el código existente. Solo se necesita crear una nueva subclase que implemente el Factory Method y devuelva el nuevo tipo de objeto. Esto facilita la incorporación de cambios y mejoras en el sistema sin afectar otras partes del código.
3. Desacoplamiento: Al utilizar el Factory Method, el código cliente no está acoplado directamente a las clases concretas, sino a la interfaz común o la clase base. Esto promueve

la modularidad y el bajo acoplamiento en el diseño, lo que facilita la prueba, el mantenimiento y la reutilización del código.

4. Soporte para el principio de inversión de dependencia: El Factory Method sigue el principio de inversión de dependencia, que establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel, sino de abstracciones. Al utilizar el Factory Method, el código cliente depende de la interfaz común en lugar de las implementaciones concretas, lo que facilita la inversión de dependencia y la aplicación de principios sólidos de diseño orientado a objetos.
5. Control sobre la creación de objetos: El Factory Method brinda control y flexibilidad sobre el proceso de creación de objetos. Las subclases pueden aplicar lógica adicional en el Factory Method para personalizar la creación de objetos según sus necesidades específicas. Esto puede incluir la configuración inicial de los objetos, la validación de parámetros o la aplicación de reglas de negocio relacionadas con la creación de objetos.

En resumen, el Factory Method proporciona una forma estructurada y flexible de crear objetos, promoviendo la encapsulación, la flexibilidad, el desacoplamiento y la extensibilidad en el diseño de software.