Taller 5 DPOO
PATRON FACTORY METHOD

Carlos Felipe Vargas Morales, 202220064, cf.vargas@uniandes.edu.co

Proyecto escogido: CreationalDesignPatternsSample by ZahraHeydari

Url: https://github.com/ZahraHeydari/CreationalDesignPatternsSample

Url para Clonar:
https://github.com/ZahraHeydari/CreationalDesignPatternsSample.git

Información general del proyecto

Para qué sirve:

El proyecto implementa el patrón Factory Method usando como ejemplo una compañía que produce dulces de distintos tipos como chocolate (oscuro, blanco y de leche) y dulces duros (Bastones de caramelo, bombones y mentas duras). En este contexto el patrón Factory Method permite instanciar estos caramelos cuando sea necesario.

Cuál es la estructura general del diseño:

NOTA: Cabe recalcar que el método Factory Method tiene 4 elementos Product, ConcreteProduct, Creator y ConcreteCreator. Estos se explorarán a fondo más adelante, pero por ahora es importante saber la existencia de estos para entender como está estructurado un proyecto que implementa Factory Method.

Clase Abstracta Candy (Product): Candy es una clase abstracta que define el comportamiento que van a compartir todos los dulces que hacen parte del proyecto. Esta clase tiene un método makeCandyPackage que será parte del contrato que deberán implementar las clases que proporcionan una implementación de este método.

Código:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public abstract class Candy{

    abstract ArrayList<Candy> makeCandyPackage(int quantity);
}
```

Clase Abstracta CandyFactory (Creator): CandyFactory es una clase abstracta que servirá como creador abstracto. Esta clase declara el método abstracto getCandy () que servirá para que las subclases ChocolateFactory y HardCandyFactory puedan instanciar dulces específicos (Chocolate oscuro, Chocolate blanco, Chocolate de leche, Bastones de caramelo, Bombones o Mentas duras).

Código:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public abstract class CandyFactory{

    public abstract Candy getCandy(String type);

    public ArrayList<Candy> getCandyPackage(int quantity, String type){
        Candy candy = getCandy(type);
        ArrayList<Candy> candyPackage = candy.makeCandyPackage(quantity);
        return candyPackage;
    }
}
```

Clase ChocolateFactory y HardCandyFactory (Concrete Creators): Son dos subclases de CandyFactory que implementan getCandy () para instanciar las clases que proporcionan una implementación de chocolates/ChocolateFactory y dulces duros/HardCandyFactory.

Código ChocolateFactory:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

public class ChocolateFactory extends CandyFactory{

    public Candy getCandy(String type){
        switch(type){
            case "white":
                return new Chocolate_White();
            case "dark":
                return new Chocolate_Dark();
            case "milk":
                return new Chocolate_Milk();
            default:
                return null;
        }
    }
}
```

Código HardCandyFactory:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

public class HardCandyFactory extends CandyFactory{


    public Candy getCandy(String type){
        switch(type){
            case "candy cane":
                return new HardCandy_CandyCane();
            case "lollipop":
                return new HardCandy_Lollipop();
            case "peppermint":
                return new HardCandy_Peppermint();
            default:
                return null;
        }
    }
}
```

(Clase Chocolate_Dark, Clase Chocolate_Milk, Clase Chocolate_White, Clase HardCandy_CandyCane, Clase HardCandy_Lollipop y Clase HardCandy_PepperMint) (Concrete Products): son las clases que extienden a la clase Candy para implementar los métodos makeCandyPackage para crear el paquete de un dulce en específico.

Código Chocolate_Dark:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class Chocolate_Dark extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> chocolatePackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            Chocolate_Dark chocolate = new Chocolate_Dark();
            chocolatePackage.add(chocolate);
        }
        System.out.println("One package of "+ quantity + " chocolates has been made!");
        return chocolatePackage;
    }

}
```

Código Chocolate_Milk:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class Chocolate_Milk extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> chocolatePackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            Chocolate_Milk chocolate = new Chocolate_Milk();
            chocolatePackage.add(chocolate);
        }
        System.out.println("One package of "+ quantity + " chocolates has been made!");
        return chocolatePackage;
    }

}
```

Código Clase Chocolate_White:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class Chocolate_White extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> chocolatePackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            Chocolate_White chocolate = new Chocolate_White();
            chocolatePackage.add(chocolate);
        }
        System.out.println("One package of "+ quantity + " chocolates has been made!");
        return chocolatePackage;
    }

}
```

Código HardCandy_CandyCane:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class HardCandy_CandyCane extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> hardCandyPackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            HardCandy_CandyCane hardCandy = new HardCandy_CandyCane();
            hardCandyPackage.add(hardCandy);
        }
        System.out.println(quantity + " of 10 candy canes have been made!");
        return hardCandyPackage;
    }

}
```

Código HardCandy_Lollipop:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class HardCandy_Lollipop extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> hardCandyPackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            HardCandy_Lollipop hardCandy = new HardCandy_Lollipop();
            hardCandyPackage.add(hardCandy);
        }
        return hardCandyPackage;
    }

}
```

Código HardCandy_PepperMint:

```java
package com.android.creationaldesignpatterns.factorymethod.full;

import java.util.ArrayList;

public class HardCandy_Peppermint extends Candy {

    @Override
    ArrayList<Candy> makeCandyPackage(int quantity){

        ArrayList<Candy> hardCandyPackage = new ArrayList<>();
        for(int i=0 ; i < quantity ; i++){
            HardCandy_Peppermint hardCandy = new HardCandy_Peppermint();
            hardCandyPackage.add(hardCandy);
        }
        System.out.print(quantity + " of 10 candy canes have been made!");
        return hardCandyPackage;
    }

}
```

Clase CandyStore: Esta clase utiliza se encarga de crear diferentes tipos de paquetes de caramelos.

En resumen, este proyecto utiliza el patrón Factory Method para dar a las subclases ChocolateFactory y HardCandyFactory (Creator) la libertad de elegir el tipo de caramelo que desean hacer. Sin ser consciente de las clases concretas precisas que genera, la clase de cliente CandyStore utiliza la interfaz común CandyFactory para crear muchos tipos de dulces. El sistema es extensible y flexible gracias a este método.

Qué grandes retos de diseño enfrenta (i.e. ¿qué es lo difícil?):

Extensible y Mantenimiento:

El código debe ser extensible para poder agregar nuevos tipos de caramelos y fabricas fácilmente sin necesidad de modificar el código que ya está. Esta es una de las características del método Factory Method la cual es fácil mantenimiento y extensibilidad.

Gestión de acoplamiento: La estructura del código debe estar enfocada en reducir el acoplamiento entre las clases. La implementación de Factory Method en este proyecto ayuda a encapsular la creación de objetos para reducir el acoplamiento entre el cliente (CandyStore) y las clases concretas (ChocolateFactory y HardCandyFactory)

Creación de Objetos: Cada tipo de caramelo (Concrete Producto) debe estar encapsulado a su respectiva fabrica (Concrete Creator). De esta manera cada fabrica tendrá la responsabilidad de crear instancias de su tipo de caramelo.

Información general sobre el patrón:

Que patrón es: Factory Method

Factory Method es un patrón de diseño de creación que brinda a las subclases la capacidad de modificar el tipo de instancias que se crean y al mismo tiempo proporciona una interfaz para hacerlo. El patrón en cuestión pertenece a la "creación de objetos" que se enfoca en generar objetos a partir de una jerarquía de clases.

El propósito principal del diseño del método de fábrica es dar a las subclases la autoridad para crear objetos de modo que la clase principal pueda otorgar a sus subclases acceso a la instalación. Debido a que las subclases pueden ofrecer varias implementaciones del método de fábrica para representar tipos de objetos particulares, esto fomenta la flexibilidad del diseño.

Uso usual:

1.Una clase no puede predecir qué tipo de objetos debería crear.

2.Una clase quiere que sus subclases especifiquen los objetos que crean.

3. Dependiendo del contexto, una clase delega la responsabilidad de la instalación a una de sus subclases.

Elementos principales:

Producto (Product): Define la interfaz de los objetos a crear.

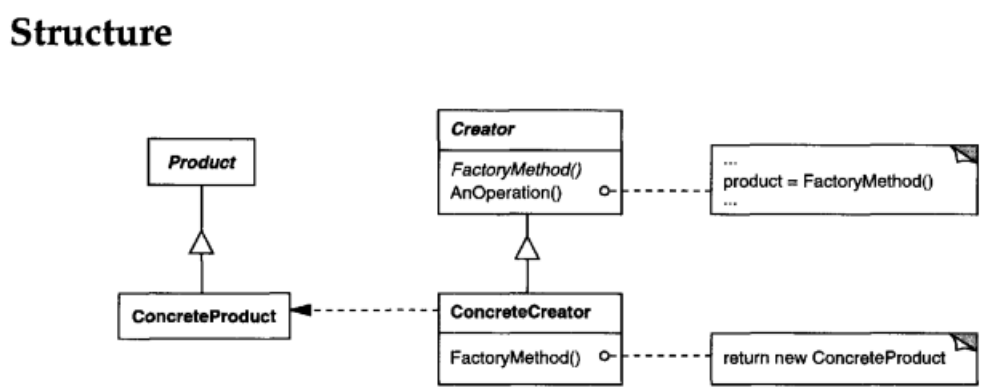
Creador: declara el método de fábrica abstracto que las subclases deben implementar para crear instancias de productos.

Producto Concreto: Implementa la interfaz definida por el producto.

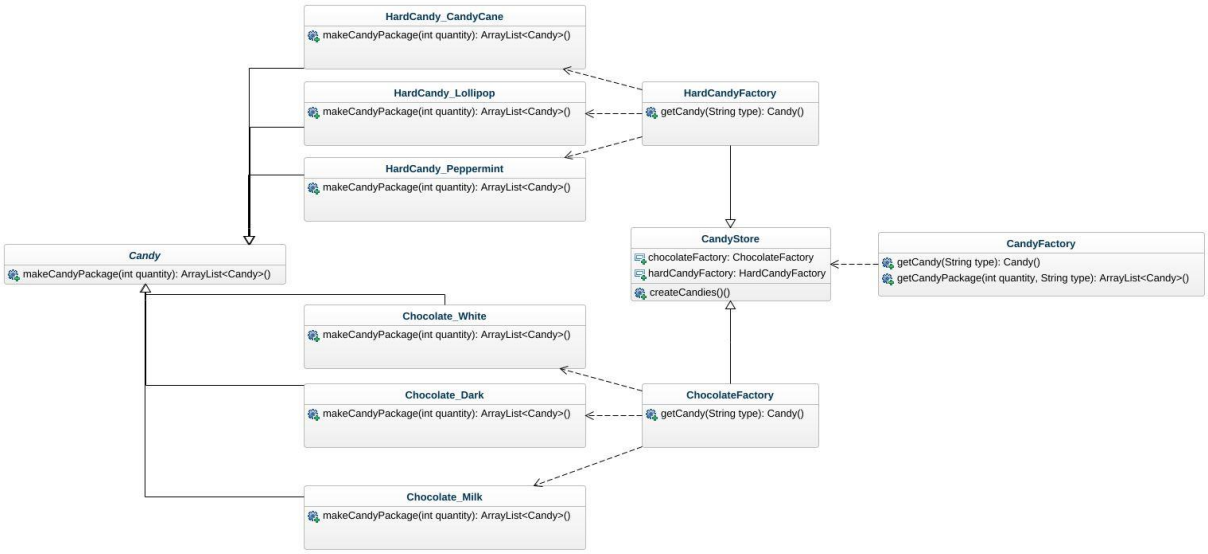
Concrete Creator: implementa el método de fábrica para crear instancias de productos concretos.

Análisis Uml:

UML del método Factory Method del libro Design Patterns



UML fabrica de caramelos:



La organización fundamental y las relaciones entre clases en este proyecto se reflejan en este UML. Las fábricas concretas (ChocolateFactory y HardCandyFactory) implementan el método getCandy para crear instancias específicas de dulces, y cada clase concreta de dulces tiene su propia implementación de makeCandyPackage.

¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto?

Descripción

Adaptabilidad en la creación:

Permite que las subclases, puedan elegir qué tipo particular de caramelo producir gracias al patrón Factory Method. En este caso HardCandyFactory y ChocolateFactory (Concrete Creators) pueden crear instancias de dulces particulares usando su propia lógica. Esto es útil cuando diferentes tipos de dulces pueden tener diferentes lógicas de creación.

Desacoplamiento del cliente:

No es necesario que la clase de cliente (CandyStore) conozca las clases de dulces concretas específicas de las que se están creando instancias. Para obtener instancias de dulces, utiliza la interfaz común de CandyFactory. Como resultado, hay menos acoplamiento entre el cliente y las implementaciones particulares, lo que facilita la expansión y modificación del sistema sin afectar al cliente.

Adaptabilidad y extensibilidad:

Es sencillo incorporar nuevas variedades de dulces al sistema utilizando el patrón Factory Method. Todo lo que se debe hacer para agregar otro tipo de dulces en el futuro es crear una nueva fábrica que expanda Candy Factory y una nueva clase (Concrete creator) que expanda Candy. Además CandyStore y el cliente, no necesitan ningún cambio.

Encapsulamiento de la lógica de la creación:

Los Concrete Creator (HardCandyFactory y ChocolateFactory) ofrecen un encapsulamiento del razonamiento detrás de la elaboración de todo tipo de dulces. Es el deber de cada Concrete Creator producir instancias del tipo particular de caramelo que produce. Esto se adhiere al principio de encapsulación ya que la lógica de creación está restringida a clases particulares.

Reutilizar código:

Es sencillo reutilizar código cuando se utiliza el patrón Factory Method. La lógica de fábrica encapsula la lógica para crear instancias de dulces específicos, y esta lógica se puede reutilizar en todo el sistema.

¿Qué ventajas tiene?

Descripción

Separación:

Las clases de cliente y los Concrete Creator (HardCandyFactory y ChocolateFactory) que generan objetos están menos acopladas cuando se utiliza el patrón Factory Method. Es sencillo alterar y hacer crecer el sistema sin tener un impacto en el cliente porque la clase de cliente aprovecha la interfaz del creador abstracto sin ser consciente de las implementaciones concretas precisas.

Tanto extensibilidad como flexibilidad:

Permite que las subclases tengan la opción de elegir qué clase específica debe crear una instancia. Debido a que las subclases pueden ofrecer varias implementaciones del método de fábrica para crear instancias de varios tipos de objetos, esto le da flexibilidad al sistema. Además, facilita la expansión del sistema al agregar nuevas clases concretas y las fábricas que las acompañan.

Encapsulamiento:

Las clases creadoras, o Concrete Creators, contienen la lógica para crear objetos. Debido a que la lógica relacionada está contenida en ubicaciones discretas, cada clase creadora está a cargo de crear instancias de un tipo particular de objeto, lo que facilita el mantenimiento y la comprensión del código.

Reutilizar código:

La lógica utilizada en la creación de objetos se puede aplicar a otras áreas del sistema. Una nueva clase concreta y su fábrica correspondiente simplemente se crean cada vez que se introduce un nuevo tipo de objeto y el código existente se deja intacto.

Polimorfismo:

El uso del polimorfismo se simplifica mediante el patrón Factory Method. La clase cliente no tiene que preocuparse por implementaciones concretas, solo usa la interfaz del creador abstracto. Esto hace posible que el código del cliente interactúe de forma transparente con varios tipos de productores y bienes.

Autoridad de producción:

Proporciona a la creación de objetos una ubicación centralizada. Dado que las subclases tienen control sobre la creación de objetos, el proceso se puede personalizar y ajustar según sea necesario.

Ayuda con pruebas y mantenimiento:

Debido a que las clases de cliente se pueden probar con implementaciones concretas u objetos simulados en lugar de tener que crear instancias de objetos concretos directamente, el proceso de escritura de pruebas unitarias se simplifica. También facilita el mantenimiento del código al colocar la lógica de compilación en lugares designados.

¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

Descripción

Complicaciones adicionales:

La complejidad del código puede aumentar con la introducción del patrón Factory Method, particularmente en proyectos pequeños y simples. En los casos en los que no se espera una variabilidad sustancial y la creación de objetos es relativamente simple, puede considerarse innecesario introducir el patrón Factory Method.

Un aumento en la cantidad de clases ofrecidas:

Podría haber más clases en el sistema si se utiliza el patrón. Se utilizan diferentes clases para representar cada tipo de producto (en este caso, dulces) y su fábrica única, lo que puede llevar a un aumento en la complejidad del proyecto y en el número de archivos.

Sobre-Abstracción:

El patrón Factory Method puede causar una abstracción innecesaria si las subclases no son realmente necesarias para personalizar la creación de objetos. Es posible que utilizar un método de fábrica no siempre sea más conveniente o fácil que crear instancias de objetos directamente.

Tener problemas para elegir un nombre:

Puede resultar difícil nombrar clases y métodos conectados al patrón Factory Method de forma coherente. Puede que sea necesario más trabajo para seleccionar nombres significativos y comprensibles para clases y métodos con el fin de garantizar el buen mantenimiento y legibilidad del código.

Elementos Particulares de Cada Producto:

Puede haber un número considerable de clases adicionales si cada tipo de producto (dulces) necesita su propio Concrete Creator. Puede ser más facil utilizar una sola fábrica o quizás eliminar el patrón del Método de Fábrica cuando los productos tienen una lógica de creación bastante similar.

Preocupaciones sobre la herencia múltiple:

Cuando se utiliza el patrón Factory Method, la flexibilidad de creación de objetos puede verse restringida en lenguajes que no permiten la herencia múltiple. Las fábricas concretas pueden complicar la estructura de clases, y es posible que las clases creadoras ya tengan una jerarquía de herencia.

¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Descripción

Constructor Parametrizado:

En lugar de utilizar Factory Method, se podría utilizar constructores parametrizados en las clases de caramelos. Los parámetros del constructor podrían permitir la personalización de las propiedades del caramelo. Esto podría simplificar la creación de objetos, especialmente si la variabilidad en la creación es limitada.

Métodos Estáticos de Creación:

Utilizar métodos estáticos de creación en lugar de un método de fábrica en una clase concreta. Esto permite la creación de instancias sin necesidad de crear una fábrica adicional. Sin embargo, este enfoque no proporciona la flexibilidad de cambiar la lógica de creación en subclases.

Inyección de Dependencias:

Utilizar la inyección de dependencias para proporcionar instancias de caramelos ya creadas en lugar de crearlas dentro de una fábrica. Esto permite que las dependencias (caramelos) sean configuradas externamente, facilitando la prueba y la modificación.