Factory Method

Nombre:
Factory Method

Clasificación del patrón:
Creacional

Intención:
Define una interfaz para crear un objeto, pero deja que las subclases decidan la clase a instanciar. Factory Method permite dejar la instanciación de las clases a las subclases.

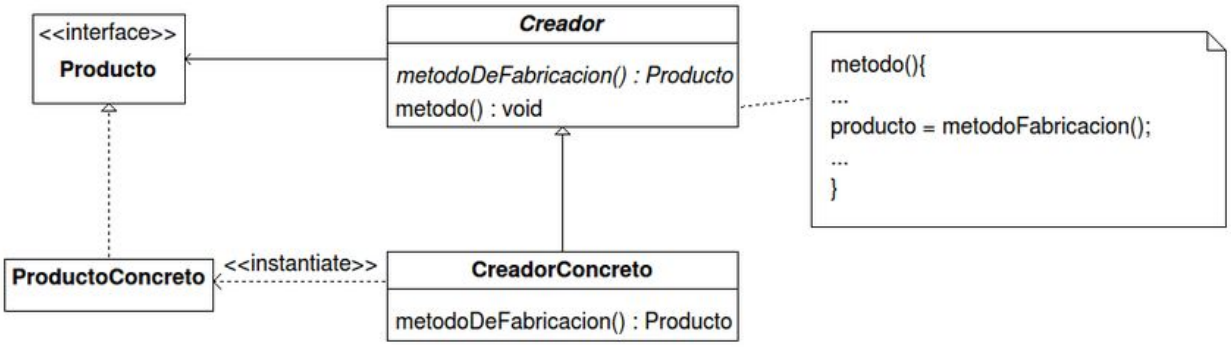
Otros nombres:
Constructor virtual

Motivación:
En un Framework se puede identificar o suponer que se usan las clases abstractas para el mantenimiento y la definición de objetos.

Aplicabilidad:
Se debe usar cuando:

- El cliente no sepa que clase puede requerir un Runtime
- Una clase quiere que sus subclases especifiquen los objetos que crea
- Se requiere encapsular la creación de objetos
- Una instancia de un objeto debe ser inicializada con algún dato no disponible por el cliente
- Una instanciación requiere muchos datos y hay muchas varaciones basados en los datos. Se provee un static Factory Method que crea las instancias con base en las diferentes variaciones.

Estructura:



Participantes:

- Producto: Define la interfaz de los objetos que la factoría crea.
- Producto Concreto: Define la interfaz del producto.
- Creador: Declara el método factoría que devuelve un producto concreto.
- Creador Concreto: Sobreescribe el método factoría para que devuelva un Producto Concreto.

Colaboraciones:

El creador busca entre las subclases y devuelve una instancia adecuada de Producto Concreto.

Ventajas:

- Promueve el bajo acoplamiento entre el cliente y las clases que utiliza.
    - No necesita depender de clases concretas.
- Encapsula la creación.
    - Controla la creación.
- Se comporta como un constructor virtual.
    - Se crea una instancia virtual. El cliente no sabe cual instancia obtuvo.
- Puede no crear un nuevo objeto con cada llamado.
    - Los objetos pueden almacenarse en caché para su rendimiento y reutilización.

Desventajas:

- Se requiere que corresponda una clase factoría para cada clase concreta.

Implementación:

- Se deben considerar dos variantes:
    - La clase Creador es abstracta y no existe una implementación del método Factoría.
    - Creador es una clase concreta y provee una implementación del método Factoría.

- Existe una variación del patrón que consiste en Factory Method con parámetros.
- Variantes y especificaciones del lenguaje.
- Plantillas para evitar la herencia.

Código de ejemplo:

- Producto:

```
1  public interface Payment {
2
3      public boolean isValidLogin(Login login);
4
5      public void doPayment(ShoppingList shoppingList);
6
7  }
```

- Pago con tarjeta:

```
1  public class CardPayment implements Payment {
2
3      @Override
4      public boolean isValidLogin(Login login) {
5          //todo implementar la lógica del login para la tarjeta.
6          return true; //valor para el ejemplo.
7      }
8
9      @Override
10     public void doPayment(ShoppingList shoppingList) {
11         //todo se implementa el método/para llevar acabo el pago con el TPV elegido.
12         //...
13     }
14 }
```

- Pago con Paypal:

```
1  public class PaypalPayment implements Payment {
2
3      @Override
4      public boolean isValidLogin(Login login) {
5          //todo implementar la lógica del login para PayPal
6          return true; //valor para el ejemplo.
7      }
8
9      @Override
10     public void doPayment(ShoppingList shoppingList) {
11         //todo hacer la implementación de la API de Paypal
12     }
13 }
```

- Pago transferencia bancaria:

```java
public class WireTransferPayment implements Payment {

    @Override
    public boolean isValidLogin(Login login) {
        //todo implementar la lógica del login para la transferencia bancaria.
        return true; //valor para el ejemplo.
    }

    @Override
    public void doPayment(ShoppingList shoppingList) {
        //todo hacer la implementación de la API de la transferencia
    }
}
```

- Creador concreto:

```java
public class PaymentFactory {

    public enum TypePayment { CARD, PAYPAL, WIRE_TRANSFER }

    public static Payment getPayment(TypePayment typePayment){
        switch (typePayment){
            case CARD:
                return new CardPayment();
            case PAYPAL:
                return new PaypalPayment();
            case WIRE_TRANSFER:
                return new WireTransferPayment();
            default:
                //Para hacerlo más entendible por defecto devolvemos tarjeta.
                return new CardPayment();
        }
    }

}
```

Principal:

```
public class MainPayment {

    private Payment payment;

    /**
     * Se obtiene la forma de pago elegida por el usuario. En esta clase no importa que pago haya
     * elegido, para nosotros es una abstracción.
     *
     * @param typePayment
     */
    public MainPayment(PaymentFactory.TypePayment typePayment){
        this.payment = PaymentFactory.getPayment(typePayment);
    }


    public boolean isLogged(Login login){
        return (payment.isValidLogin(login));
    }

    /**
     * Método para realizar el pago de una lista de compra.
     *
     * @param shoppingList Lista de Compra
     *
     * Como podemos ver, podemos ampliar los tipos de pagos sin que el proceso se vea alterado.
     */
    public void payment(ShoppingList shoppingList){
        payment.doPayment(shoppingList);
    }

}
```

Usos conocidos:

- Frameworks.
- Creación de proxys en middlewares.

Patrones relacionados:

- Abstract Factory

Bibliografía:

No específico. (No específico). GoF Design Patterns (Versión 2.1.0) [Aplicación móvil]. Descargado de: https://drive.google.com/file/d/0BywiVyFlIabXcVhGZlJBcnhWTkU/view.

Patrones de diseño clásicos [Página web]. (s. f.). Ubicación http://www.lsi.us.es/docencia/get.php?id=604.

Zuluaga, P. (21 de abril de 2015). Factory Method. Patrones de Software. http://patronesdedis.blogspot.com/2015/04/factory-method.html

Junta de Andalucía. (s.f). Factoría. Marco de Desarrollo de la Junta de Andalucía. http://www.juntadeandalucia.es/servicios/madeja/print/424.