Factory Method Pattern

Introducción y nombre

Factory Method. Creacional. Libera al desarrollador sobre la forma correcta de crear objetos.

Intención

Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.

También conocido como

Virtual Constructor, Método de Factoría.

Motivación

Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

Solución

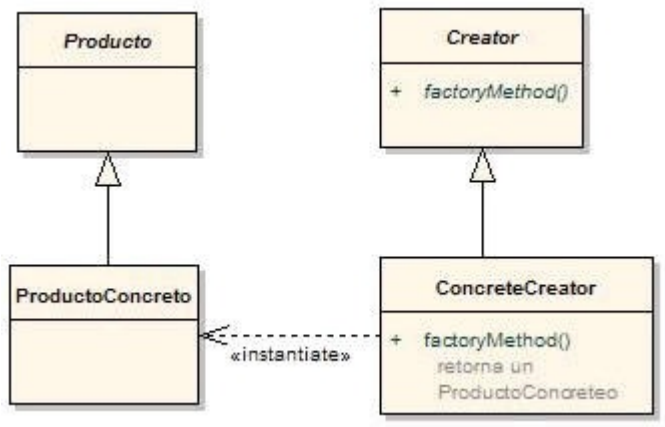
Este patrón debe ser utilizado cuando:

Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.

Hay clases que delegan responsabilidades en una o varias subclases.

Una aplicación es grande y compleja y posee muchos patrones creacionales.

Diagrama UML



Participantes

Responsabilidad de cada clase participante:

Creator: declara el método de fabricación, que devuelve un objeto de tipo product. Puede llamar a dicho método para crear un objeto producto .

ConcretCreator: redefine el método de fabricación para devolver un objeto concretProduct.

Colaboraciones

ProductoConcreto: es el resultado final. El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

Consecuencias

Como ventaja se destaca que elimina la necesidad de introducir clases específicas en el código del creador. Solo maneja la interfaz Product, por lo que permite añadir cualquier clase ConcretProduct definida por el usuario.

Otra ventaja: es más flexible crear un objeto con un Factory Method que directamente: un método factoría puede dar una implementación por defecto.

Un inconveniente es tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Implementación de la clase Creator

El método factoría es abstracto

El método factoría proporciona una implementación por defecto:

Permite extensibilidad: se pone la creación de objetos en una operación separada por si el usuario quiere cambiarla

Código de muestra

Asumamos que tenes una clase Triángulo y necesitamos crear un tipo de triángulo: escaleno, isosceles o equilatero. Para ello, esta la clase abstracta Triangulo y sus clases hijas:

[code]

```
public abstract class Triangulo {

    private int ladoA;
    private int ladoB;
    private int ladoC;

    public Triangulo(int ladoA, int ladoB, int ladoC) {
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public abstract String getDescripcion();

    public abstract double getSuperficie();

    public abstract void dibujate();
}

public class Equilatero extends Triangulo {

    public Equilatero(int anguloA, int anguloB, int anguloC) {
        super(anguloA, anguloB, anguloC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
```

```
    }

    @Override
    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}

public class Escaleno extends Triangulo {

    public Escaleno(int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Escaleno";
    }

    @Override
    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}

public class Isosceles extends Triangulo {

    public Isosceles(int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Isosceles";
    }

    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}
```

Para evitar que nuestros clientes deban conocer la estructura de nuestra jerarquía creamos un Factory de triángulos:

```
public class TrianguloFactory {
    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC) {
        if ((ladoA == ladoB) && (ladoA == ladoC)) {
            return new Equilatero(ladoA, ladoB, ladoC);
        } else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)) {
```

```
            return new Escaleno(ladoA, ladoB, ladoC);
        } else {
            return new Isosceles(ladoA, ladoB, ladoC);
        }
    }
}
```

De esta forma, no sólo no tienen que conocer nuestra estructura de clases, sino que además tampoco es necesario que conozcan nuestra implementación (el conocimiento del algoritmo que resuelve que clases se deben crear).

Simplemente deberían crear un triángulo de la siguiente forma:

```
public class Main {
    public static void main(String[] args) {
        TrianguloFactory factory = new TrianguloFactory();

        Triangulo triangulo = factory.createTriangulo(10, 10, 10);
        System.out.println(triangulo.getDescripcion());
    }
}
[/code]
```

Cabe aclara que las clases Factory deberian ser Singleton, pero para evitar confundir al estudiante nos hemos concentrado sólo en este patrón.

Cuando utilizarlo

El escenario típico para utilizar este patrón es cuando existe una jerarquía de clases complejas y la creación de un objeto específico obliga al cliente a tener que conocer detalles específicos para poder crear un objeto. En este caso se puede utilizar un Factory Method para eliminar la necesidad de que el cliente tenga la obligación de conocer las todas especificaciones y variaciones posibles.

Por otro lado, imaginemos una aplicación grande, realizada por un grupo de desarrolladores que han utilizado muchos patrones creacionales. Esto hace que el grupo de programadores pierda el control sobre como se debe crear una clase. Es decir, cual es un Singleton o utiliza un Builder o un Prototype. Entonces se realiza un Factory Method para que instancie los objetos de una manera estandard. El desarrollador siempre llama a uno o varios Factory y se olvida de la forma en que se deben crear todos los objetos.

Patrones relacionados

Abstract Factory: suele ser implementada por varios Factory Method.

Prototype y Builder: Si bien hay casos donde parece que se excluyen mutuamente, en la mayoría de los casos suelen trabajar juntos.

Singleton: un Factory Method suele ser una clase Singleton.