



Universidad Autónoma de Nuevo León

---



# Facultad de Ciencias Físico Matemáticas

Diseño Orientado a Objetos

---

**Erick Alejandro Campos Rivero**

**1506449**

**Patrón de diseño**

**Prof.: Miguel Ángel Salazar Santillán**



## ¿Qué es un patrón?

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de la solución al problema, de forma que puede utilizarse un millón de veces sin tener que hacer dos veces lo mismo.

**Definición de un patrón de diseño:** un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular.

## Puntos clave de un patrón de diseño.

Es un tema importante en el desarrollo de software actual:

- permite capturar la experiencia
- Busca ayudar a la comunidad de desarrolladores de software a resolver problemas comunes, creando un cuerpo literario de base – Crea un lenguaje común para comunicar ideas y experiencia acerca de los problemas y sus soluciones
- El uso de patrones ayuda a obtener un software de calidad (reutilización y extensibilidad)

## Elementos de un patrón

- Nombre: describe el problema de diseño.
- El problema: describe cuándo aplicar el patrón.
- La solución: describe los elementos que componen el diseño, sus relaciones, responsabilidades y colaboración

## Los patrones creacionales más conocidos son:

- **ABSTRACT FACTORY:** Nos provee una interfaz que delega la creación de un conjunto de objetos relacionados sin necesidad de especificar en ningún momento cuáles son las implementaciones concretas.

### ***Estructura típica***

Cliente: La clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría, es decir, Cliente lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB).

AbstractFactory: Es de definición de las interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear. ("crearProductoA()" y "crearProductoB()")

Factorías Concretas: Estas son las diferentes familias de productos. Provee de la instancia concreta de la que se encarga de crear. De esta forma podemos tener una factoría que cree los elementos gráficos para Windows y otra que los cree para Linux, pudiendo poner fácilmente (creando una nueva) otra que los cree para MacOS, por ejemplo.

Producto abstracto: Definición de las interfaces para la familia de productos genéricos. En el diagrama son "ProductoA" y "ProductoB". En un ejemplo de interfaces gráficas podrían ser todos los elementos: Botón, Ventana, Cuadro de Texto, Combo... El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.

Producto concreto: Implementación de los diferentes productos. Podría ser por ejemplo "BotónWindows" y "BotónLinux". Como ambos implementan "Botón" el cliente no sabrá si está en Windows o Linux, puesto que trabajará directamente sobre la superclase o interfaz.

### ***Un ejemplo***

Para mostrar el concepto del Abstract Factory vamos a hacer un sencillo reloj que nos muestra la hora actual. Como sabemos, la hora puede ser desplegada en formato de 24Hrs o puede ser desplegada en formato AM/PM. Recordando que es a manera de ejemplo, vamos a utilizar la clase Date de una manera que no se debe, y probablemente el reloj lo haríamos de una manera más sencilla, pero para nuestro ejemplo queda perfecta su uso. Como en el caso del diccionario, haremos una clase abstracta de Reloj y dos implementaciones para cada una de los formatos, y una clase que contenga el método del Factory Method. La cosa quedaría algo así:

```
public abstract class Reloj {
    abstract String dameLaHora();
}
```

La clase que se da la hora en formato AM/PM:

```
public class RelojAmPm extends Reloj{
    public RelojAmPm(){
    }
    public String dameLaHora() {
        Date d = new Date();
        int hora = d.getHours();
        int minutos = d.getMinutes();
        int segundos = d.getSeconds();
        String tr;
        if (hora<12){
            tr="Son las "+hora+":"+minutos+"."+segundos+" AM";
        } else {
            tr="Son las "+(hora-12)+":"+minutos+"."+segundos+" PM";
        }
        return tr;
    }
}
```

La que nos da la hora en formato de 24 horas:

```
public class Reloj24Hrs extends Reloj {
    public String dameLaHora() {
        Date d = new Date();
        int hora = d.getHours();
        int minutos = d.getMinutes();
        int segundos = d.getSeconds();
        String tr;
        tr = "Son las " + hora + ":" + minutos + ":" + segundos + " ";
        return tr;
    }
}
```

Nuestra clase que contiene la el método que elije las instancias.

```
public class RelojFactory {
    public static final int RELOJ_AM_PM=0;
    public static final int RELOJ_24_HRS=1;
    public RelojFactory(){
    }
    public static Reloj createReloj(int tipoDeReloj){
        if (tipoDeReloj==RelojFactory.RELOJ_24_HRS){
            return new Reloj24Hrs();
        }
        if (tipoDeReloj==RelojFactory.RELOJ_AM_PM){
            return new RelojAmPm();
        }
        return null;
    }
}
```

Y finalmente la clase cliente, que será el usuario final:

```
public class MainClient {  
    public static void main(String[] args) {  
        Reloj r = RelojFactory.createReloj(RelojFactory.RELOJ_24_HRS);  
        System.out.println(r.dameLaHora());  
    }  
}
```

Hasta aquí tenemos dos fábricas: una de palabras, y la que acabamos de hacer que nos da la hora.

-----

**FACTORY METHOD**: es un patrón de diseño que define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos. Factory Method permite escribir aplicaciones que son más flexibles respecto de los tipos a utilizar difiriendo la creación de las instancias en el sistema a subclases que pueden ser extendidas a medida que evoluciona el sistema. Permite también encapsular el conocimiento referente a la creación de objetos. Factory Method hace también que el diseño sea más adaptable a cambio de sólo un poco más de complejidad.

### ***Propósito***

Definir una interfaz para la creación de un objeto, pero permitiendo a las subclases decidir de qué clase instanciarlo. Permite, por tanto, que una clase difiera la instanciación en favor de sus subclases.

### **Participantes**

- **Producto**: Define la interfaz de los objetos que crea el método de fabricación.
- **Producto Concreto**: Implementa la interfaz Producto.
- **Creador**: Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto Producto Concreto. Puede llamar al método de fabricación para crear un objeto Producto.
- **Creador Concreto**: Redefine el método de fabricación para devolver una instancia de Producto Concreto.

### Ejemplo

En nuestro ejemplo tenemos una clase abstracta llamada Triangulo, de la cual heredan los 3 tipos de triángulos conocidos.

```
public abstract class Triangulo {
    private int ladoA;
    private int ladoB;
    private int ladoC; // con sus get y set

    public Triangulo(int ladoA, int ladoB, int ladoC) {
        setLadoA(ladoA);
        setLadoB(ladoB);
        setLadoC(ladoC);
    }

    //Cada subclase debe redefinir estos tres métodos abstractos.
    public abstract String getDescripcion();

    public abstract double getSuperficie();

    public abstract void dibujate();

    public int getLadoA() {
        return ladoA;
    }

    public void setLadoA(int ladoA) {

}

public class Equilatero extends Triangulo {

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

    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
    }

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo equilatero.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo equilatero.
    }
}
```

```

public class Escaleno extends Triangulo {

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

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

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo escaleno.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo escaleno.
    }
}

public class Isosceles extends Triangulo {

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

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

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo isosceles.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo isosceles.
    }
}

```

Quien se encargue de crear un tipo de triángulo concreto no debería tener que conocer como se compone internamente. Para ello, hemos creado la clase TrianguloFactory con su correspondiente interface.

```

public interface TrianguloFactoryMethod {
    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC);
}

public class TrianguloFactory implements TrianguloFactoryMethod {

    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC) {

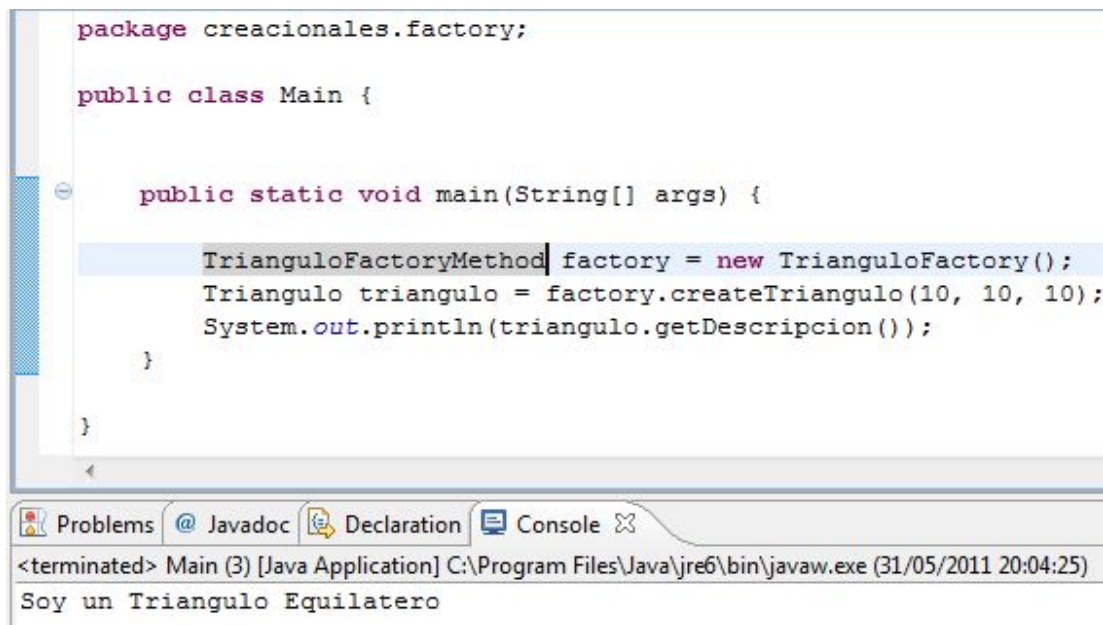
        if ((ladoA == ladoB) && (ladoA == ladoC)) {
            return new Equilatero(ladoA, ladoB, ladoC);
        }

        else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)) {
            return new Escaleno(ladoA, ladoB, ladoC);
        }

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

```

Desde el punto de vista del cliente, es muy sencillo poder crear un triángulo:



```
package creacionales.factory;

public class Main {

    public static void main(String[] args) {

        TrianguloFactoryMethod factory = new TrianguloFactory();
        Triangulo triangulo = factory.createTriangulo(10, 10, 10);
        System.out.println(triangulo.getDescripcion());
    }

}
```

Problems @ Javadoc Declaration Console

<terminated> Main (3) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (31/05/2011 20:04:25)

Soy un Triangulo Equilatero

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.

Por otro lado, es más flexible crear un objeto con un Factory Method que directamente: un método factoría puede dar una implementación por defecto.

-----

**BUILDER:** permite la creación de un objeto complejo, a partir de una variedad de partes que contribuyen individualmente a la creación y ensamblación del objeto mencionado. Hace uso de la frase "divide y conquistarás". Por otro lado, centraliza el proceso de creación en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes. Se debe utilizar este patrón cuando sea necesario:

- Independizar el algoritmo de creación de un objeto complejo de las partes que constituyen el objeto y cómo se ensamblan entre ellas.
- Que el proceso de construcción permita distintas representaciones para el objeto construido, de manera dinámica.
- Este patrón debe utilizarse cuando el algoritmo para crear un objeto suele ser complejo e implica la interacción de otras partes independientes y una coreografía entre ellas para formar el ensamblaje. Por ejemplo: la construcción de un objeto Computadora, se compondrá de otros muchos objetos, como puede ser un objeto PlacaDeSonido, Procesador, PlacaDeVideo, Gabinete, Monitor, etc.



El objetivo del ejemplo es poder crear un objeto Auto. El auto se compondrá de varios atributos que lo componen: motor, marca, modelo y cantidad de puertas. En nuestro ejemplo, el auto no se compone de muchos objetos complejos. De hecho, se compone de sólo 4 objetos relativamente sencillos. Esto es para poder hacer entendible la propuesta del Builder y no perderse en los objetos que lo componen. Queda en la imaginación del lector la posibilidad de trabajar con ejemplos más complejos. Yo particularmente usé mucho este patrón cuando trabajé con archivos.

```
package creacionales.builder;

public class Auto {
    private int cantidadDePuertas;
    private String modelo;
    private String marca;
    private Motor motor;

    public int getCantidadDePuertas() {
        return cantidadDePuertas;
    }

    public void setCantidadDePuertas(int cantidadDePuertas) {
        this.cantidadDePuertas = cantidadDePuertas;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}
```

```
package creacionales.builder;

public class Motor {
    private int numero;
    private String potencia;

    public int getNumero() {
        return numero;
    }

    public void setNumero(int numero) {
        this.numero = numero;
    }

    public String getPotencia() {
        return potencia;
    }

    public void setPotencia(String potencia) {
        this.potencia = potencia;
    }
}
```

Definimos nuestro Builder llamado AutoBuilder. El Builder define al menos dos cosas: un método para devolver el Producto (el auto en nuestro caso) y los métodos necesarios para la construcción del mismo.

```

package creacionales.builder;

public abstract class AutoBuilder {
    protected Auto auto;

    public Auto getAuto() {
        return auto;
    }

    public void crearAuto() {
        auto = new Auto();
    }

    public abstract void buildMotor();

    public abstract void buildModelo();

    public abstract void buildMarca();

    public abstract void buildPuertas();
}

```

Serán los ConcreteBuilders los encargados de colocarle la lógica de construcción de cada Auto en particular. En nuestro caso, tendremos dos ConcreteBuilder: FiatBuilder y FordBuilder. Recordemos que, en nuestro ejemplo, son clases que construyen objetos muy sencillos con datos hardcodeados para facilitar el aprendizaje del patrón en sí.

```

package creacionales.builder;

public class FiatBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Fiat");
    }

    public void buildModelo() {
        auto.setModelo("Palio");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(232323);
        motor.setPotencia("23 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(2);
    }
}

package creacionales.builder;

public class FordBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Ford");
    }

    public void buildModelo() {
        auto.setModelo("Focus");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(21212);
        motor.setPotencia("20 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(4);
    }
}

```

Realizaremos el Director. Lo primero que debe hacerse con esta clase es enviarle el tipo de auto que se busca construir (Ford, Fiat, etc). Luego, al llamar al método constructAuto(), la construcción se realizará de manera automática

```
package creacionales.builder;

public class AutoDirector {
    // No es necesario que exista la palabra Director
    // Esta clase podría llamarse Concesionaria, Garage, FabricaDeAutos, etc

    private AutoBuilder autoBuilder;

    public void constructAuto() {
        autoBuilder.crearAuto();
        autoBuilder.buildMarca();
        autoBuilder.buildModelo();
        autoBuilder.buildMotor();
        autoBuilder.buildPuertas();
    }

    public void setAutoBuilder(AutoBuilder ab) {
        autoBuilder = ab;
    }

    public Auto getAuto() {
        return autoBuilder.getAuto();
    }
}
```

La invocación desde un cliente sería:

```
package creacionales.builder;

public class Main {

    public static void main(String[] args) {
        AutoDirector director = new AutoDirector();
        director.setAutoBuilder(new FordBuilder());
        director.constructAuto();
        Auto auto = director.getAuto();

        System.out.println(auto.getMarca());
        System.out.println(auto.getModelo());
    }
}
```

Problems @ Javadoc Declaration Console

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (

Ford  
Focus

### Resultados:

- Permite variar la representación interna de un producto.
- El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.
- Si se cambia la representación interna basta con crear otro Builder que respete la interfaz.
- Separa el código de construcción del de representación.
- Las clases que definen la representación interna del producto no aparecen en la interfaz del Builder.
- Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto.
- Distintos Directores pueden usar un mismo ConcreteBuilder.
- Da mayor control en el proceso de construcción.
- Permite que el Director controle la construcción de un producto paso a paso.
- Sólo cuando el producto está acabado lo recupera el director del builder.

### OTROS PATRONES DE DISEÑO:

- **Singleton:** limita a uno el número de instancias posibles de una clase en nuestro programa, y proporciona un acceso global al mismo.
- **Prototype:** Permite la creación de objetos basados en “plantillas”. Un nuevo objeto se crea a partir de la clonación de otro objeto.

### **Conclusión**

Nos dan la oportunidad de crear nuestro código de manera mucho más sencilla con estructuras probadas y que funcionan. La mayor complejidad radica en saber cuándo utilizarlas, algo que nos dará la práctica.