

Patrones de diseño

Un patrón de diseño es una solución reutilizable para un problema que ocurre dentro de un contexto de programación dado. En ocasiones, los programadores encuentran el mismo problema varias veces en distintos proyectos. Así que, en vez de que cada uno aporte o diseñe su propia solución, se crean los patrones de diseño en Java.

Elementos de los patrones de diseño

En general, un patrón de diseño en Java tiene cuatro elementos esenciales:

- Es un identificador que se puede usar para describir el problema de diseño, sus soluciones y las consecuencias en una o dos palabras.
- Este elemento describe cuándo se puede aplicar el patrón y explica el problema y el contexto del mismo.
- Solución. Describe los elementos que componen el diseño. La solución no describe la implementación concreta, porque un patrón es como una plantilla que se puede aplicar en muchas situaciones diferentes.
- Son los resultados esperables de aplicar el patrón de diseño en Java.

Algunos de los patrones de diseño más utilizados son:

Factory Method

El patrón Factory Method define una interfaz para crear objetos, pero deja que las subclases decidan qué clase instanciar. Esto permite a las clases diferir la creación de objetos a subclases.

El patrón Factory Method define un método que debe utilizarse para crear objetos, en lugar de una llamada directa al constructor (operador new). Las subclases pueden sobrescribir este método para cambiar las clases de los objetos que se crearán.

Este patrón es bastante común en las principales bibliotecas de Java:

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String)` (Devuelve distintos objetos singleton, dependiendo de un protocolo).
- `java.util.EnumSet#of()`
- `javax.xml.bind.JAXBContext#createMarshaller()`

Observer:

Observer es un patrón de diseño de comportamiento que permite a un objeto notificar a otros objetos sobre cambios en su estado. Este proporciona una forma de suscribirse y cancelar la suscripción a estos eventos para cualquier objeto que implementa una interfaz suscriptora.

El patrón Observer es bastante habitual en el código Java, sobre todo en los componentes GUI. Proporciona una forma de reaccionar a los eventos que suceden en otros objetos, sin acoplarse a sus clases.

Estos son algunos ejemplos del patrón en las principales bibliotecas Java:

- Todas las implementaciones de **java.util.EventListener** (prácticamente por todos los componentes Swing)
- **javax.servlet.http.HttpSessionBindingListener**
- **javax.servlet.http.HttpSessionAttributeListener**
- **javax.faces.event.PhaseListener**

Strategy

Es un patrón de diseño de comportamiento que convierte un grupo de comportamientos en objetos y los hace intercambiables dentro del objeto de contexto original.

El objeto original, llamado contexto, contiene una referencia a un objeto de estrategia y le delega la ejecución del comportamiento. Para cambiar la forma en que el contexto realiza su trabajo, otros objetos pueden sustituir el objeto de estrategia actualmente vinculado, por otro.

Este patrón se utiliza a menudo en varios frameworks para proporcionar a los usuarios una forma de cambiar el comportamiento de una clase sin extenderla. Java 8 brindó el soporte de funciones lambda, que pueden servir como alternativas más sencillas al patrón Strategy.

Estos son algunos ejemplos del patrón Strategy en las principales bibliotecas Java:

- **java.util.Comparator#compare()** invocado desde el método **Collections#sort()**.
- **javax.servlet.http.HttpServlet: service()**, más todos los métodos **doXXX()** que aceptan objetos **HttpServletRequest** y **HttpServletResponse** como argumentos.
- **javax.servlet.Filter#doFilter()**

Builder

Builder es un patrón de diseño creacional que permite construir objetos complejos paso a paso. Al contrario que otros patrones creacionales, Builder no necesita que los productos tengan una interfaz común. Esto hace posible crear distintos productos utilizando el mismo proceso de construcción.

El patrón Builder es muy útil cuando se crea un objeto con muchas opciones posibles de configuración.

El uso del patrón Builder está muy extendido en las principales bibliotecas Java, algunas de ellas son:

- **java.lang.StringBuilder#append() (unsynchronized)**
- **java.lang.StringBuffer#append() (synchronized)**
- **java.nio.ByteBuffer#put()** (también en CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer y DoubleBuffer)
- **javax.swing.GroupLayout.Group#addComponent()**
- **Todas las implementaciones java.lang.Appendable**

Adapter

Adapter es un patrón de diseño estructural que permite colaborar a objetos incompatibles. El patrón Adapter actúa como envoltorio entre dos objetos. Atrapa las llamadas a un objeto y las transforma a un formato y una interfaz reconocible para el segundo objeto.

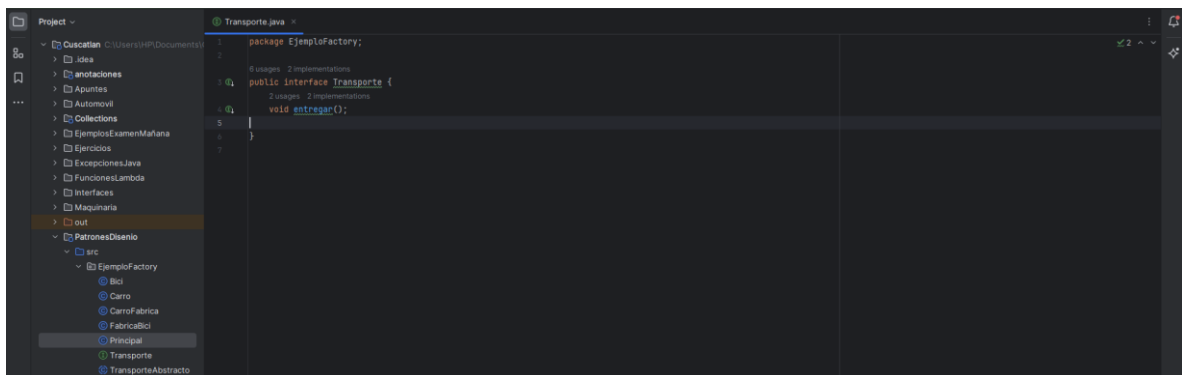
Se utiliza muy a menudo en sistemas basados en algún código heredado (legacy). En estos casos, los adaptadores crean código heredado con clases modernas.

Estos son algunos adaptadores estándar en las principales bibliotecas de Java:

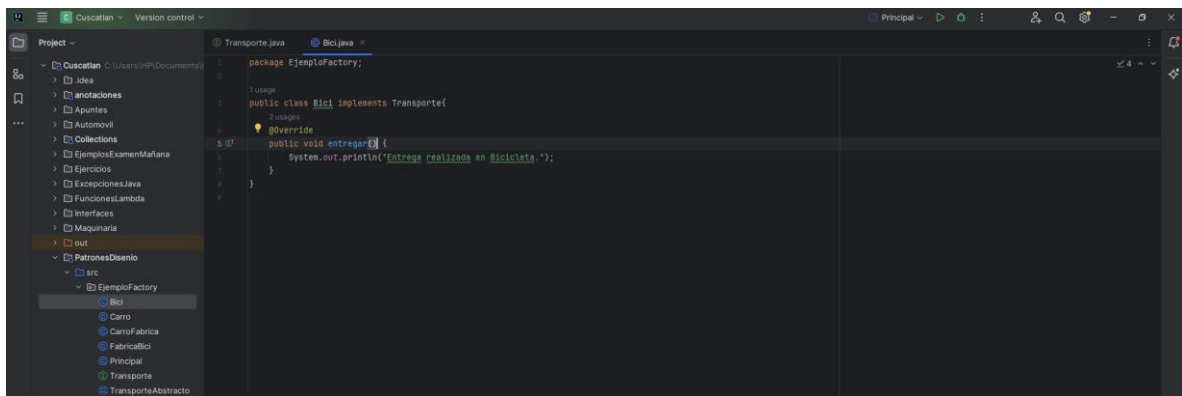
- **java.util.Arrays#asList()**
- **java.util.Collections#list()**
- **java.util.Collections#enumeration()**
- **java.io.InputStreamReader(InputStream)** (devuelve un objeto Reader)
- **java.io.OutputStreamWriter(OutputStream)** (devuelve un objeto Writer)
- **javax.xml.bind.annotation.adapters.XmlAdapter#marshal()** y **#unmarshal()**

Ejemplo de Aplicación Factory Method:

Interfaz

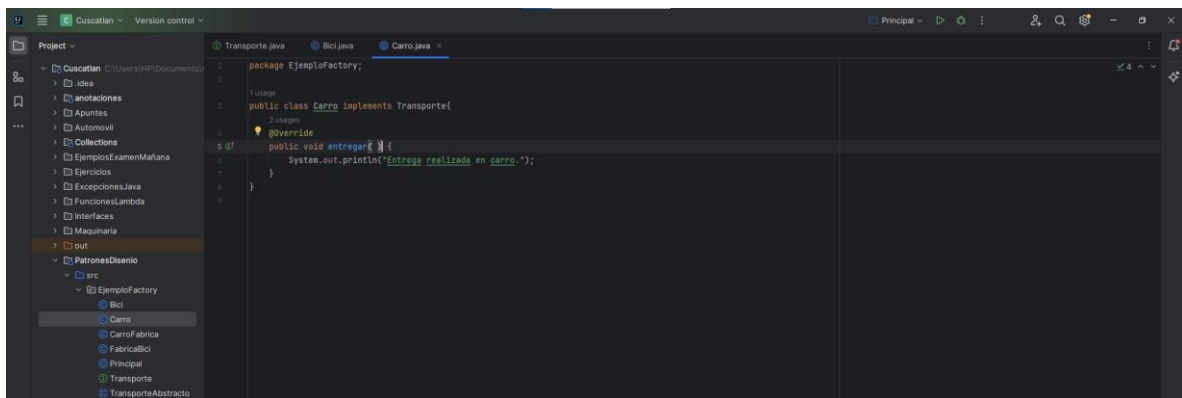


Clase Bici



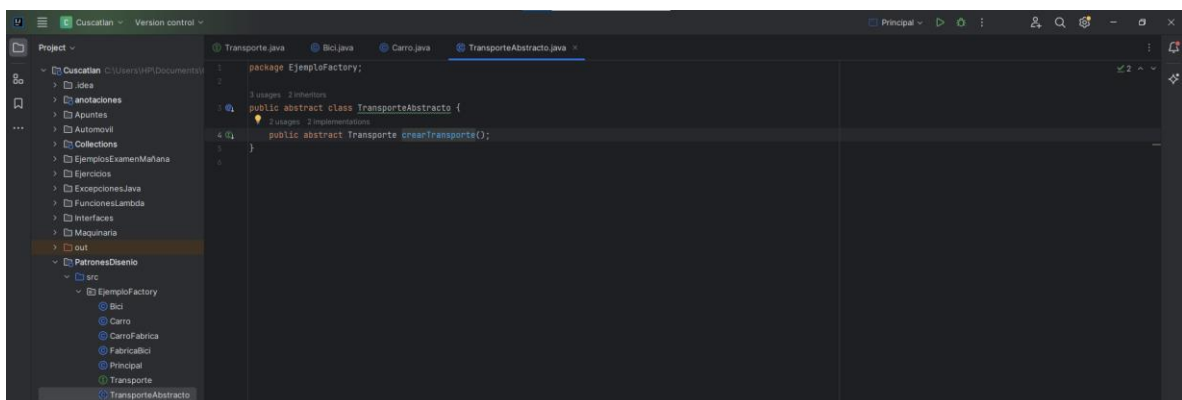
```
1 package EjemploFactory;
2
3 1 usage
4 public class Bici implements Transporte{
5     2 usages
6     @Override
7     public void entregar() {
8         System.out.println("Entrega realizada en Bici.");
9     }
10 }
```

Clase Carro



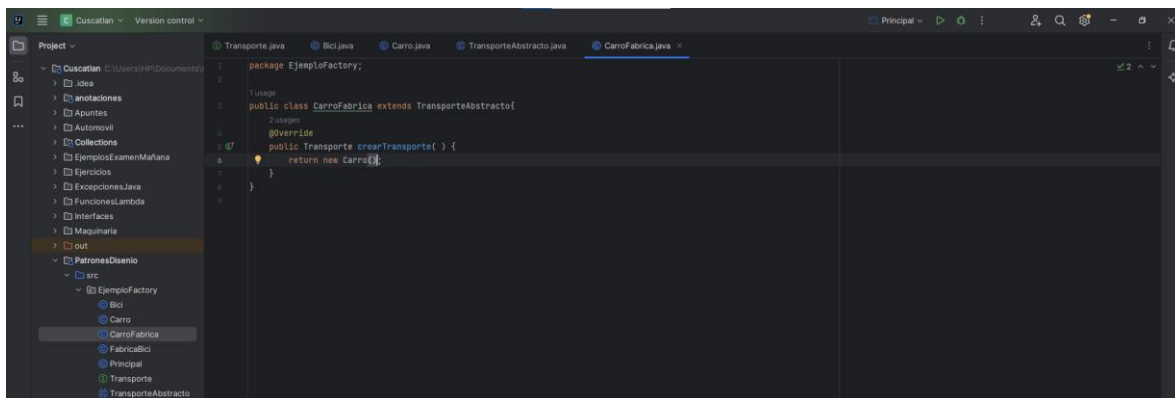
```
1 package EjemploFactory;
2
3 1 usage
4 public class Carro implements Transporte{
5     2 usages
6     @Override
7     public void entregar() {
8         System.out.println("Entrega realizada en Carro.");
9     }
10 }
```

Clase abstracta TransporteAbstracto



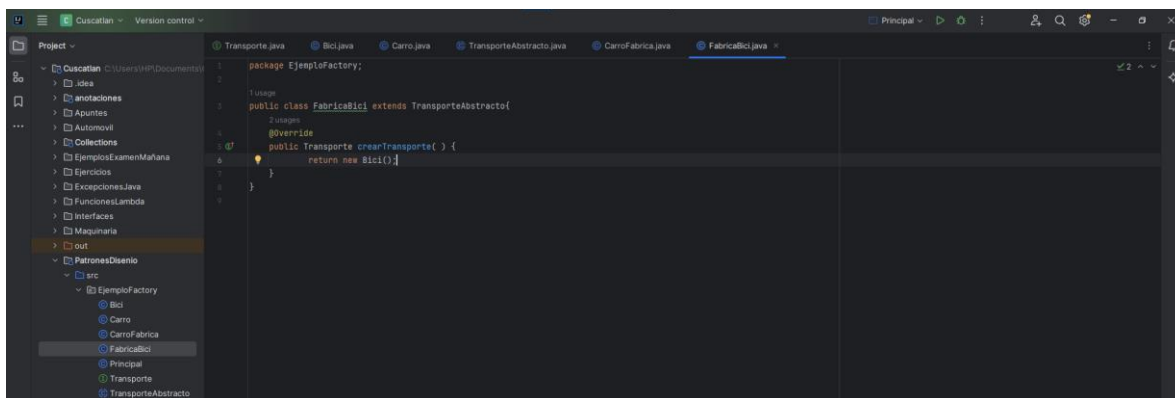
```
1 package EjemploFactory;
2
3 2 usages, 2 inheritors
4 public abstract class TransporteAbstracto {
5     1 usage, 2 implementations
6     public abstract Transporte crearTransporte();
7 }
8
```

Clase CarroFabrica



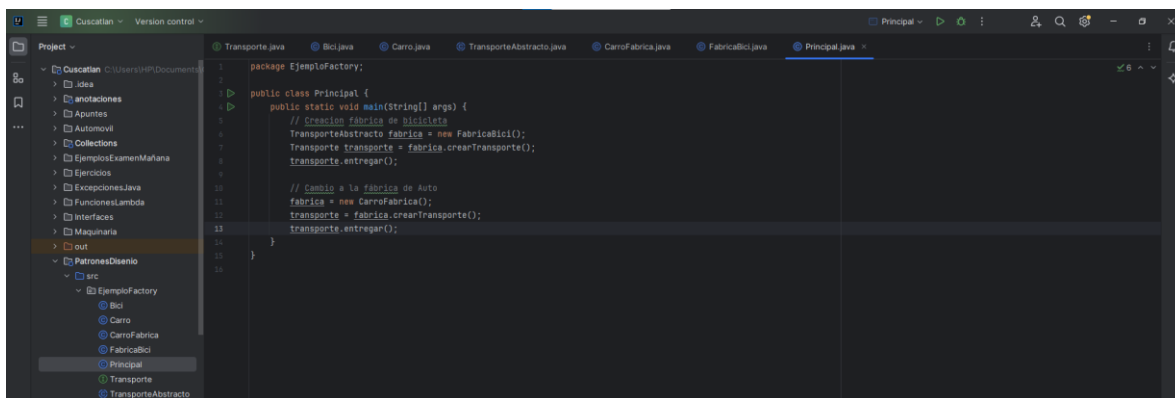
```
1 package EjemploFactory;
2
3 import TransporteAbstracto;
4
5 public class CarroFabrica extends TransporteAbstracto {
6     @Override
7     public Transporte crearTransporte() {
8         return new Carro();
9     }
10 }
```

Clase FabricaBici



```
1 package EjemploFactory;
2
3 import TransporteAbstracto;
4
5 public class FabricaBici extends TransporteAbstracto {
6     @Override
7     public Transporte crearTransporte() {
8         return new Bici();
9     }
10 }
```

Main Principal



```
1 package EjemploFactory;
2
3 public class Principal {
4     public static void main(String[] args) {
5         // Creacion fabricas de bicicletas
6         TransporteAbstracto fabrica = new FabricaBici();
7         Transporte transporte = fabrica.crearTransporte();
8         transporte.entregar();
9
10        // Cambio a la fabrica de Auto
11        fabrica = new CarroFabrica();
12        transporte = fabrica.crearTransporte();
13        transporte.entregar();
14    }
15 }
```