



INGENIERÍA EN DESARROLLO Y GESTIÓN DE SOFTWARE

Nombre de la materia:

ADS

Entregable .

Patrones de diseño

Nombre del alumno:

Anette Yunuen Ruiz Martínez

Nombre del profesor:

Rogelio Bautista Sánchez

Santiago de Querétaro, Qro. Noviembre 30 de 2024

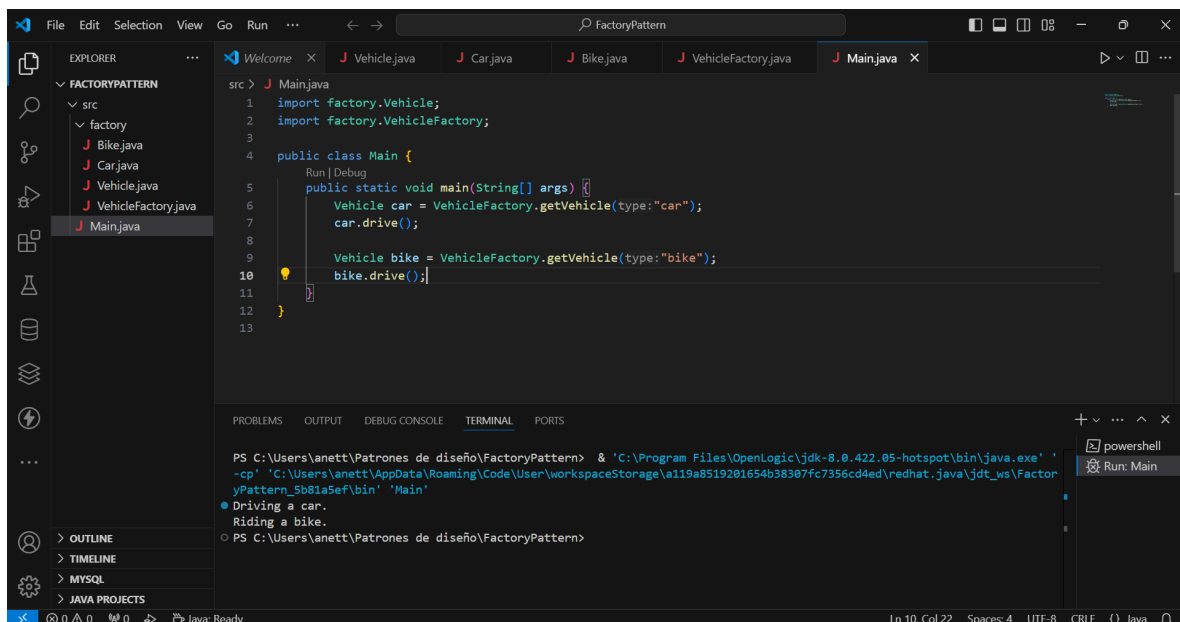
Patrón Factory

Interfaz Vehicle: Define un contrato común para los tipos de vehículos. Todas las clases concretas (como Car y Bike) implementan esta interfaz, lo que garantiza que compartan el mismo método drive().

Clases concretas (Car y Bike): Son implementaciones específicas de la interfaz Vehicle. Cada clase tiene su propia versión del método drive().

Clase VehicleFactory: Esta es la fábrica encargada de crear los objetos. Contiene un método estático (getVehicle) que recibe un parámetro (type) y decide qué tipo de objeto instanciar. Si el tipo es "car", se crea un objeto de la clase Car; si es "bike", se crea uno de la clase Bike. Esto evita que el código principal tenga que preocuparse por cómo se crean los objetos y centraliza la lógica de creación en la fábrica.

Clase principal Main: Aquí se utiliza la fábrica para obtener los objetos necesarios (Car y Bike). Gracias a la fábrica, no necesitas instanciar los objetos directamente.



```
src > J Main.java
1  import factory.Vehicle;
2  import factory.VehicleFactory;
3
4  public class Main {
5      public static void main(String[] args) {
6          Vehicle car = VehicleFactory.getVehicle(type:"car");
7          car.drive();
8
9          Vehicle bike = VehicleFactory.getVehicle(type:"bike");
10         bike.drive();
11     }
12 }
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\anett\Patrones de diseño\FactoryPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.05-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\al19a8519201654b38307fc7356cd4ed\redhat.java\jdt_ws\FactoryPattern_5b81a5ef\bin' 'Main'
● Driving a car.
○ Riding a bike.
○ PS C:\Users\anett\Patrones de diseño\FactoryPattern>
```

Patrón Abstract Factory

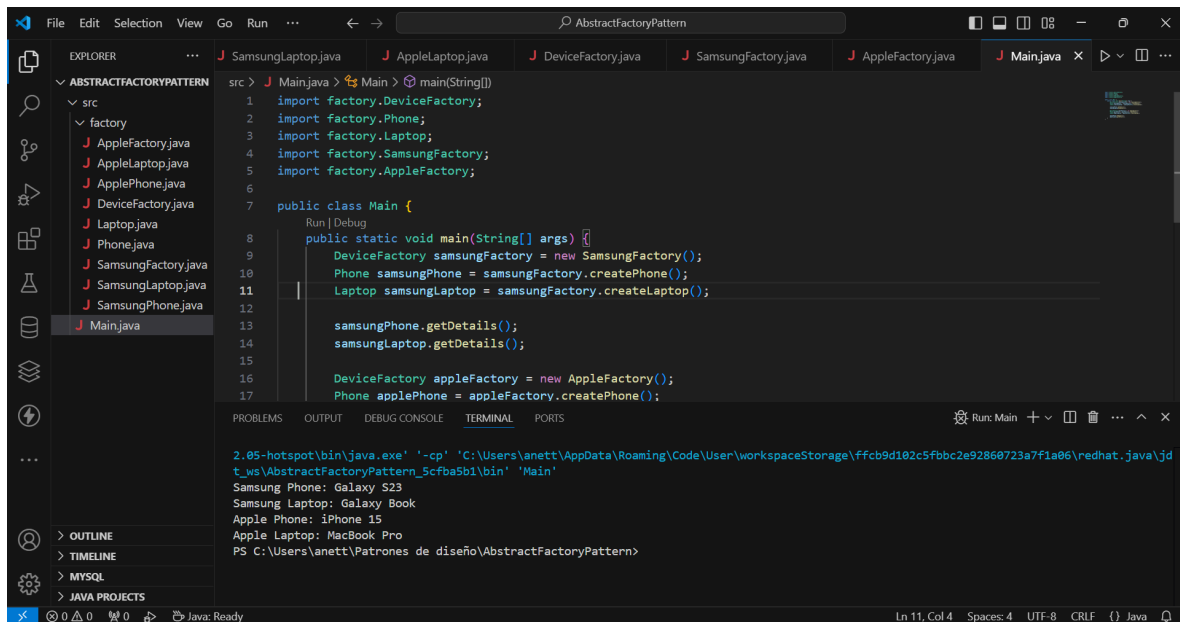
Abstract Factory (DeviceFactory): Define un contrato para crear familias de productos (teléfonos y laptops).

Fábricas concretas (SamsungFactory y AppleFactory): Implementan el contrato y crean objetos específicos (Samsung o Apple).

Productos concretos (SamsungPhone, ApplePhone, etc.): Son las clases específicas de los productos.

Ciente (Main): Usa las fábricas para obtener los productos sin preocuparse por cómo se crean.

Este patrón es útil cuando necesitas crear objetos relacionados entre sí de manera consistente, y asegura que los objetos creados pertenezcan a la misma "familia".



The screenshot shows an IDE with the following components:

- EXPLORER:** A project named 'ABSTRACTFACTORYPATTERN' with a 'src' folder containing:
 - 'factory' package: 'DeviceFactory.java', 'Phone.java', 'Laptop.java'.
 - 'Samsung' package: 'SamsungFactory.java', 'SamsungPhone.java', 'SamsungLaptop.java'.
 - 'Apple' package: 'AppleFactory.java', 'ApplePhone.java', 'AppleLaptop.java'.
 - 'Main.java' at the root of the src folder.
- EDITOR:** Displays 'Main.java' with the following code:

```
1 import factory.DeviceFactory;
2 import factory.Phone;
3 import factory.Laptop;
4 import factory.SamsungFactory;
5 import factory.AppleFactory;
6
7 public class Main {
8     public static void main(String[] args) {
9         DeviceFactory samsungFactory = new SamsungFactory();
10        Phone samsungPhone = samsungFactory.createPhone();
11        Laptop samsungLaptop = samsungFactory.createLaptop();
12
13        samsungPhone.getDetails();
14        samsungLaptop.getDetails();
15
16        DeviceFactory appleFactory = new AppleFactory();
17        Phone applePhone = appleFactory.createPhone();
```
- TERMINAL:** Shows the output of running the program:

```
2.05-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\ffcb9d102c5fbbc2e92860723a7f1a06\redhat.java\jd
t_w\AbstracFactoryPattern_5cfba5b1\bin' 'Main'
Samsung Phone: Galaxy S23
Samsung Laptop: Galaxy Book
Apple Phone: iPhone 15
Apple Laptop: MacBook Pro
PS C:\Users\anett\Patrones de diseño\AbstracFactoryPattern>
```

Patrón Abstract Factory

Variable estática instance: Almacena la única instancia de la clase Logger. Es estática para que sea accesible sin necesidad de instanciar la clase.

Constructor privado: Evita que otras clases puedan crear instancias directamente usando `new Logger()`.

Método estático getInstance(): Proporciona acceso a la única instancia de Logger. Si la instancia no existe, la crea. Si ya existe, devuelve la misma instancia.

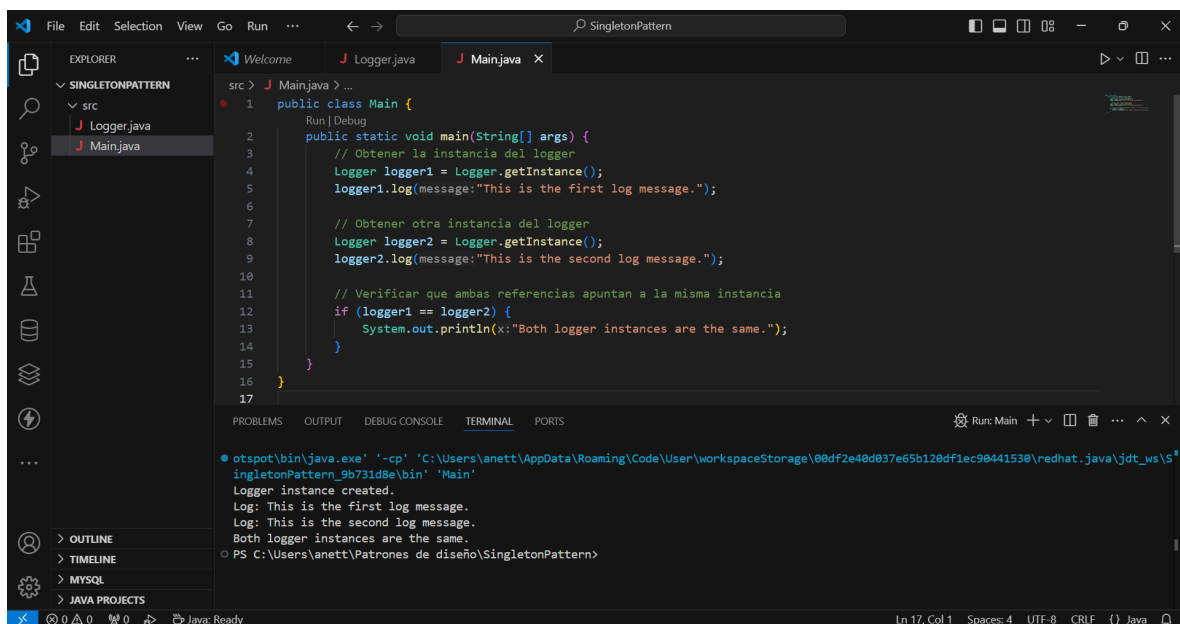
Método log(String message): Permite registrar mensajes en el sistema de logs.

Verificación de unicidad: En Main, al comparar `logger1` y `logger2`, verificamos que ambas referencias apuntan a la misma instancia.

Control de acceso único: Garantiza que solo haya una instancia de la clase en toda la aplicación.

Punto de acceso global: Permite acceder a la instancia desde cualquier parte del código.

Fácil de implementar: Es un patrón sencillo y útil en escenarios como manejo de configuraciones, registro de logs, y conexión a bases de datos.



```
src > J Main.java > ...
1 public class Main {
2     public static void main(String[] args) {
3         // Obtener la instancia del logger
4         Logger logger1 = Logger.getInstance();
5         logger1.log(message:"This is the first log message.");
6
7         // Obtener otra instancia del logger
8         Logger logger2 = Logger.getInstance();
9         logger2.log(message:"This is the second log message.");
10
11        // Verificar que ambas referencias apuntan a la misma instancia
12        if (logger1 == logger2) {
13            System.out.println(x:"Both logger instances are the same.");
14        }
15    }
16 }
17
```

```
otspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\00df2e40d837e65b120df1ec90441530\redhat.java\jdt_ws\S
ingletonPattern_9b731d8e\bin' 'Main'
Logger instance created.
Log: This is the first log message.
Log: This is the second log message.
Both logger instances are the same.
PS C:\Users\anett\Patrones de diseño\SingletonPattern>
```

Patrón Prototype

El patrón Prototype es otro patrón creacional que se utiliza para crear nuevos objetos mediante la clonación de una instancia existente. Este patrón es útil cuando la creación de un objeto es costosa en términos de tiempo o recursos.

Interfaz Form: Define el contrato para clonar objetos (clone) y mostrar su contenido (display).

Clases concretas (RegistrationForm y ContactForm): Implementan la lógica específica para clonar objetos y mostrar información. Al clonar, crean una nueva instancia con los mismos valores de la instancia original.

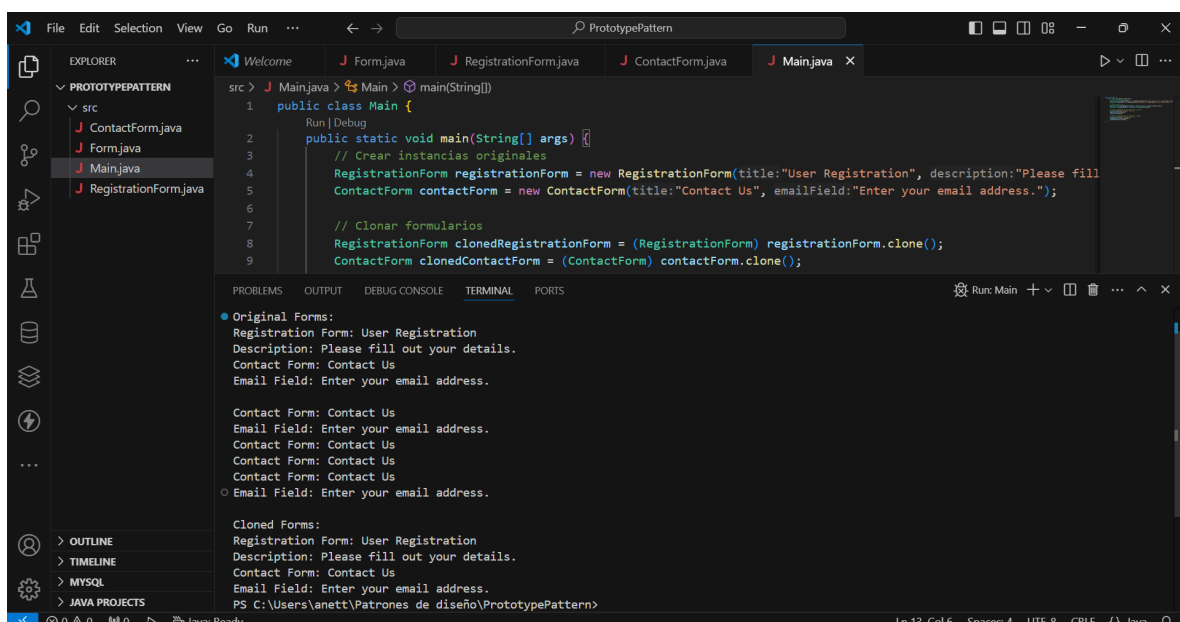
Clase principal (Main): Crea instancias originales de formularios. Clona esas instancias utilizando el método clone().

Muestra que los objetos originales y clonados son independientes pero tienen los mismos valores.

Reducción de costos: Evita crear nuevos objetos desde cero, lo que puede ser costoso en términos de tiempo o recursos.

Flexibilidad: Permite personalizar y reutilizar objetos clonados según sea necesario.

Simplicidad: Es fácil de implementar y de extender a nuevos tipos de objetos.



```
src > J Main.java > Main > main(String[])
1 public class Main {
2     public static void main(String[] args) {
3         // Crear instancias originales
4         RegistrationForm registrationForm = new RegistrationForm(title:"User Registration", description:"Please fill
5         ContactForm contactForm = new ContactForm(title:"Contact Us", emailField:"Enter your email address.");
6
7         // Clonar formularios
8         RegistrationForm clonedRegistrationForm = (RegistrationForm) registrationForm.clone();
9         ContactForm clonedContactForm = (ContactForm) contactForm.clone();
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Run: Main

Original Forms:
Registration Form: User Registration
Description: Please fill out your details.
Contact Form: Contact Us
Email Field: Enter your email address.

Contact Form: Contact Us
Email Field: Enter your email address.
Contact Form: Contact Us
Contact Form: Contact Us
Contact Form: Contact Us

Email Field: Enter your email address.

Cloned Forms:
Registration Form: User Registration
Description: Please fill out your details.
Contact Form: Contact Us
Email Field: Enter your email address.

PS C:\Users\anetti\Patrones de diseño\PrototypePattern>

Patrón Adapter

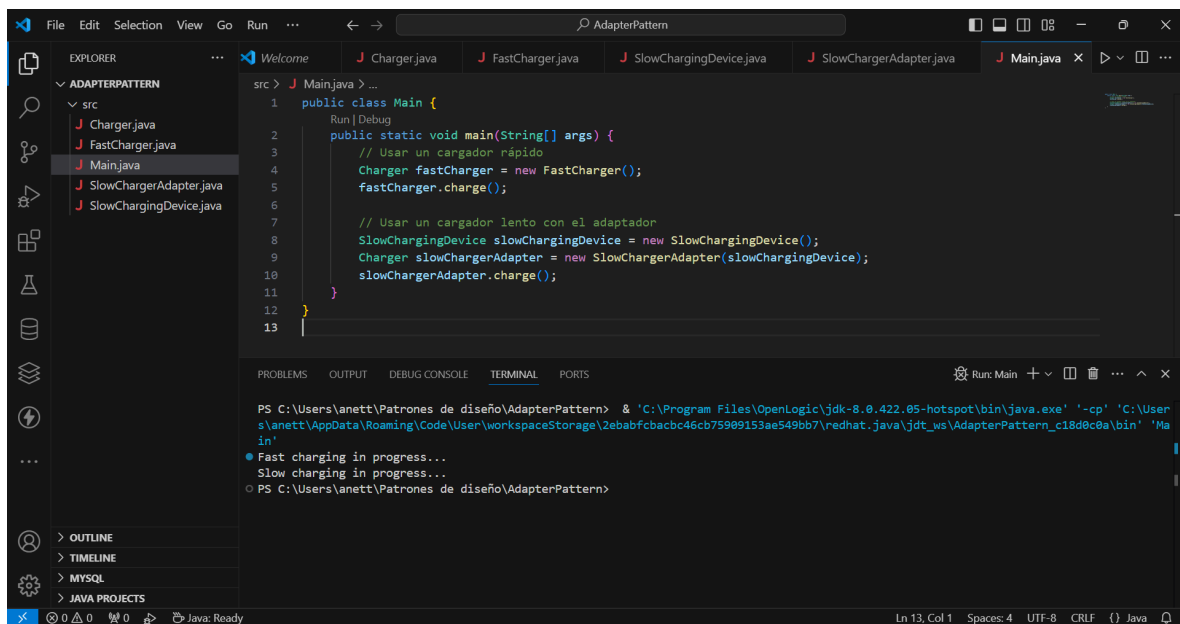
Interfaz Charger: Define un método estándar `charge()` que todos los cargadores deben implementar.

Clase FastCharger: Implementa directamente la interfaz Charger para carga rápida.

Clase SlowChargingDevice: Tiene un método propio `slowCharge()` que no es compatible con la interfaz Charger.

Clase SlowChargerAdapter: Actúa como un puente entre `SlowChargingDevice` y `Charger`. Adapta el método `slowCharge()` para que sea compatible con la interfaz `charge()`.

Clase Main: Permite usar ambas implementaciones (carga rápida y carga lenta) de manera uniforme mediante la interfaz `Charger`.



```
src > J Main.java > ...
1  public class Main {
2      public static void main(String[] args) {
3          // Usar un cargador rápido
4          Charger fastCharger = new FastCharger();
5          fastCharger.charge();
6
7          // Usar un cargador lento con el adaptador
8          SlowChargingDevice slowChargingDevice = new SlowChargingDevice();
9          Charger slowChargerAdapter = new SlowChargerAdapter(slowChargingDevice);
10         slowChargerAdapter.charge();
11     }
12 }
13
```

```
PS C:\Users\anett\Patrones de diseño\AdapterPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.05-hotspot\bin\java.exe' '-cp' 'C:\User
s\anett\AppData\Roaming\Code\User\workspaceStorage\2ebabfcabc46cb75969153ae549bb7\redhat.java\jdt_ws\AdapterPattern_c18d0c0a\bin' 'Ma
in'
```

```
● Fast charging in progress...
○ Slow charging in progress...
```

```
PS C:\Users\anett\Patrones de diseño\AdapterPattern>
```

Patrón Decorator

Flexibilidad: Puedes agregar funcionalidades adicionales a un objeto en tiempo de ejecución.

Reusabilidad: Los decoradores son reutilizables y pueden combinarse en diferentes configuraciones.

Cumple con el principio de abierto/cerrado: Puedes extender la funcionalidad de las clases sin modificar el código existente.

The screenshot shows the IntelliJ IDEA IDE with the DecoratorPattern project. The Explorer on the left lists the project structure, including the src directory and the Main.java file. The Main.java file is open in the editor, showing the main method with comments in Spanish. The Run console at the bottom shows the output of the program, displaying the cost of different coffee orders.

```

src > J Main.java > ...
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         // Crear un café básico
5         Beverage coffee = new Coffee();
6         System.out.println(coffee.getDescription() + " - $" + coffee.getCost());
7
8         // Agregar leche al café
9         coffee = new MilkDecorator(coffee);
10        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());
11
12        // Agregar chocolate al café con leche
13        coffee = new ChocolateDecorator(coffee);
14        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());
15    }
16 }
  
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● PS C:\Users\anett\Patrones de diseño\DecoratorPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.85-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\be755a4182373e06a9151c854afac68\redhat.java\jdt_ws\DecoratorPattern_3625c75e\bin' 'Main'

```

Coffee - $2.0
Coffee, Milk - $2.5
Coffee, Milk, Chocolate - $3.2
Coffee, Milk, Chocolate, Whipped Cream - $4.2
  
```

○ PS C:\Users\anett\Patrones de diseño\DecoratorPattern>

Patrón Observer

Desacoplamiento:

Los sujetos no necesitan saber quiénes son sus observadores ni cómo funcionan.

Escalabilidad:

Puedes agregar más observadores fácilmente sin modificar el código del sujeto.

Automatización:

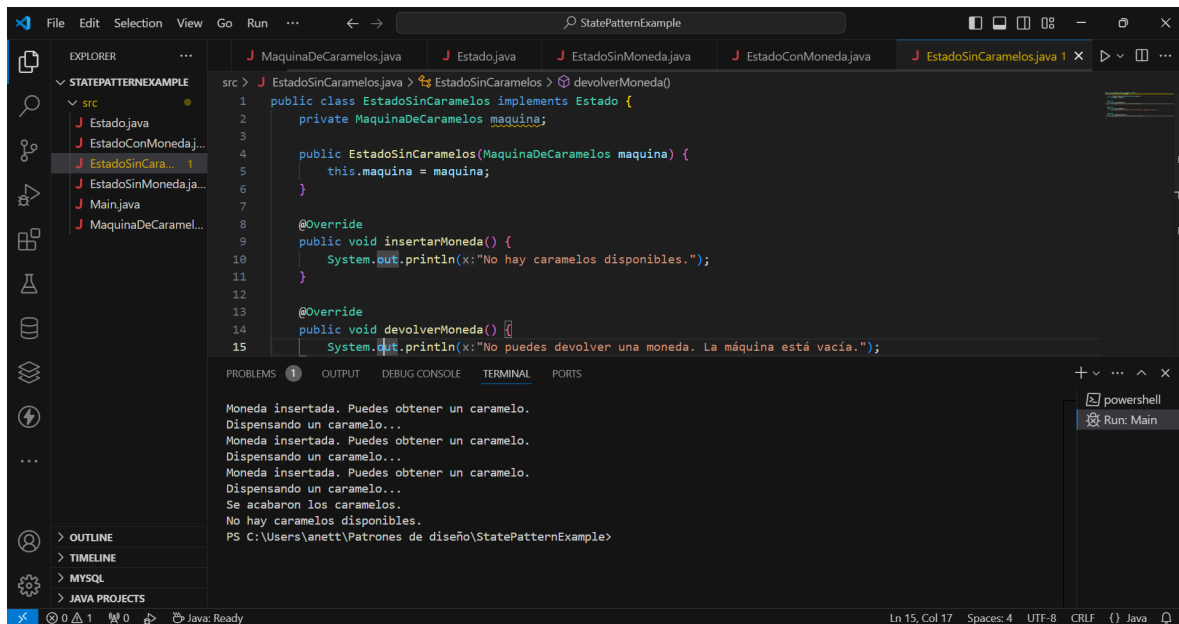
Los observadores reciben notificaciones automáticamente cuando ocurre un cambio.

Patrón State

Clases de Estado: Cada estado (sin moneda, con moneda, sin caramelos) tiene su propia implementación de las acciones posibles: insertar moneda, devolver moneda y despachar caramelo.

Cambio de Estado: La máquina cambia su estado interno (estadoActual) dependiendo de las acciones realizadas (por ejemplo, al insertar una moneda o quedarse sin caramelos).

Encapsulación: El comportamiento específico de cada estado está encapsulado en clases individuales, lo que hace el código más fácil de mantener y extender.



```
src > J EstadoSinCaramelos.java > EstadoSinCaramelos > devolverMoneda()
1 public class EstadoSinCaramelos implements Estado {
2     private MaquinaDeCaramelos maquina;
3
4     public EstadoSinCaramelos(MaquinaDeCaramelos maquina) {
5         this.maquina = maquina;
6     }
7
8     @Override
9     public void insertarMoneda() {
10        System.out.println("No hay caramelos disponibles.");
11    }
12
13    @Override
14    public void devolverMoneda() {
15        System.out.println("No puedes devolver una moneda. La máquina está vacía.");
16    }
17 }
```

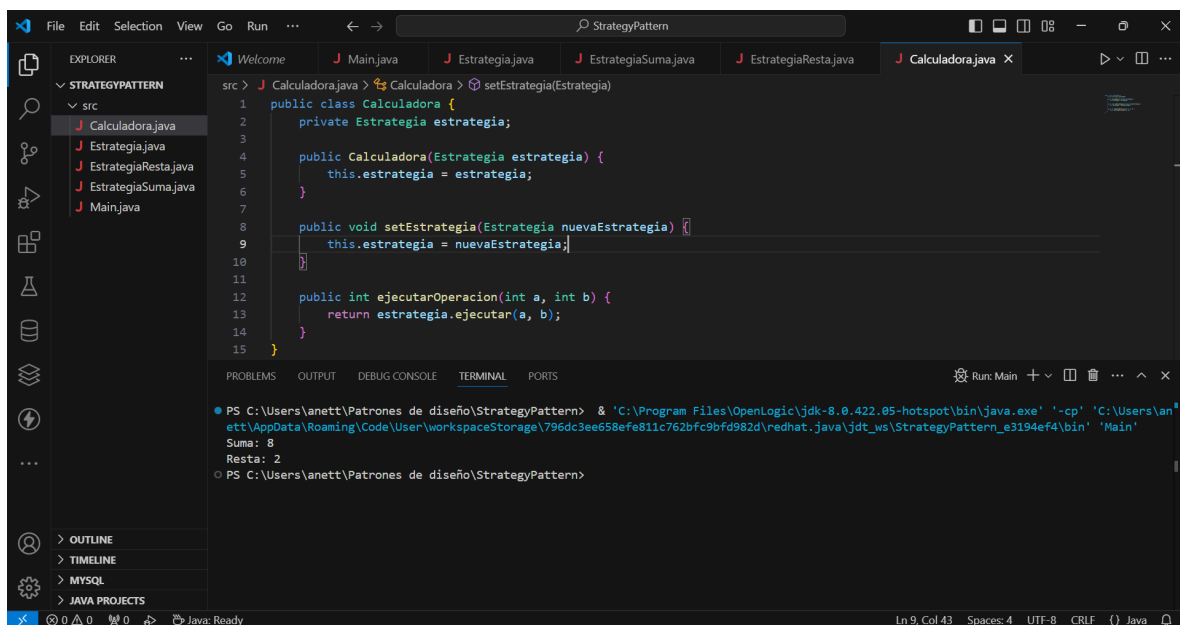
Moneda insertada. Puedes obtener un caramelo.
Dispensando un caramelo...
Moneda insertada. Puedes obtener un caramelo.
Dispensando un caramelo...
Moneda insertada. Puedes obtener un caramelo.
Dispensando un caramelo...
Se acabaron los caramelos.
No hay caramelos disponibles.
PS C:\Users\anett\Patrones de diseño\StatePatternExample>

Patrón State

Qué es: El patrón Strategy permite definir una familia de algoritmos (estrategias), encapsularlas en clases separadas y usarlas de forma intercambiable.

Cómo funciona aquí: Tenemos dos estrategias: EstrategiaSuma y EstrategiaResta. La clase Calculadora usa una estrategia y permite cambiarla en tiempo de ejecución.

Beneficio principal: Hace el código más flexible al permitir cambiar o agregar nuevas estrategias sin modificar el código existente.



```
src > J Calculadora.java > Calculadora > setEstrategia(Estrategia)
1 public class Calculadora {
2     private Estrategia estrategia;
3
4     public Calculadora(Estrategia estrategia) {
5         this.estrategia = estrategia;
6     }
7
8     public void setEstrategia(Estrategia nuevaEstrategia) {
9         this.estrategia = nuevaEstrategia;
10    }
11
12    public int ejecutarOperacion(int a, int b) {
13        return estrategia.ejecutar(a, b);
14    }
15 }
```

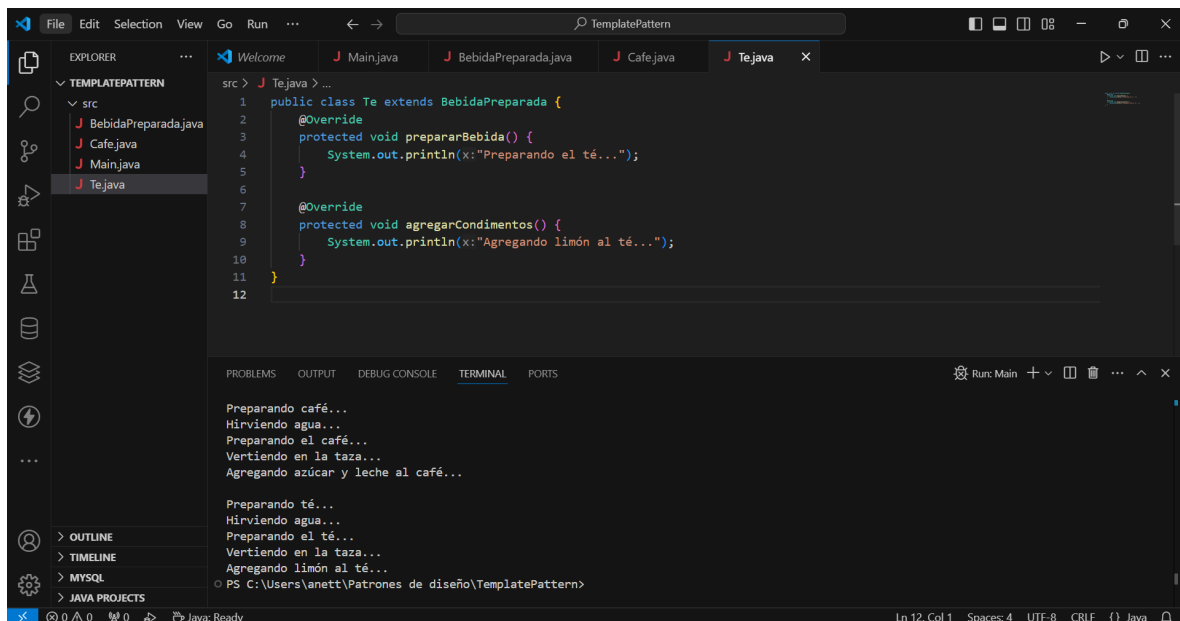
```
PS C:\Users\anett\Patrones de diseño\StrategyPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.05-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\796dc3ee658efe811c762bfc9bfd982d\redhat.java\jdt_ws\StrategyPattern_e3194ef4\bin' 'Main'
Suma: 8
Resta: 2
PS C:\Users\anett\Patrones de diseño\StrategyPattern>
```

Patrón Template

Qué es: El patrón Template define el esqueleto de un algoritmo en una clase base y permite que las subclases implementen detalles específicos de algunos pasos.

Cómo funciona aquí: La clase abstracta `BebidaPreparada` contiene los pasos comunes y el método plantilla `preparar()`. Las subclases (`Cafe` y `Te`) implementan los pasos específicos como `prepararBebida()` y `agregarCondimentos()`.

Beneficio principal: Promueve la reutilización de código común, permite modificar pasos específicos sin alterar la estructura general del algoritmo.



```
src > J Te.java > ...
1 public class Te extends BebidaPreparada {
2     @Override
3     protected void prepararBebida() {
4         System.out.println(x:"Preparando el té...");
5     }
6
7     @Override
8     protected void agregarCondimentos() {
9         System.out.println(x:"Agregando limón al té...");
10    }
11
12 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Run: Main

Preparando café...
Hirviendo agua...
Preparando el café...
Vertiendo en la taza...
Agregando azúcar y leche al café...

Preparando té...
Hirviendo agua...
Preparando el té...
Vertiendo en la taza...
Agregando limón al té...

PS C:\Users\anett\Patrones de diseño\TemplatePattern>

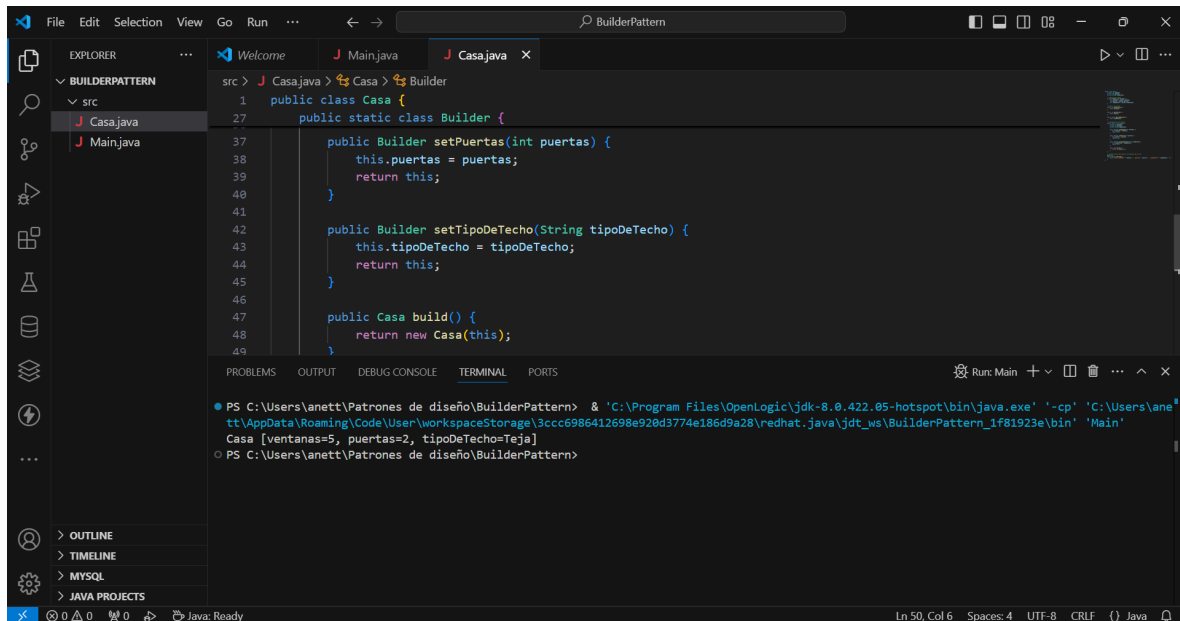
Ln 12, Col 1 Spaces: 4 UTF-8 CRLF {} Java

Patrón Builder

Qué es: El Patrón Builder se utiliza para construir objetos complejos paso a paso. Es útil cuando un objeto tiene múltiples atributos opcionales.

Cómo funciona aquí: La clase Casa tiene un constructor privado que solo puede ser invocado por el Builder. La clase Builder permite configurar los atributos de la casa y finalmente crearla con el método build().

Beneficio principal: Hace que la creación de objetos con muchos parámetros sea más legible y evita constructores con demasiados argumentos.



The screenshot shows an IDE with the following components:

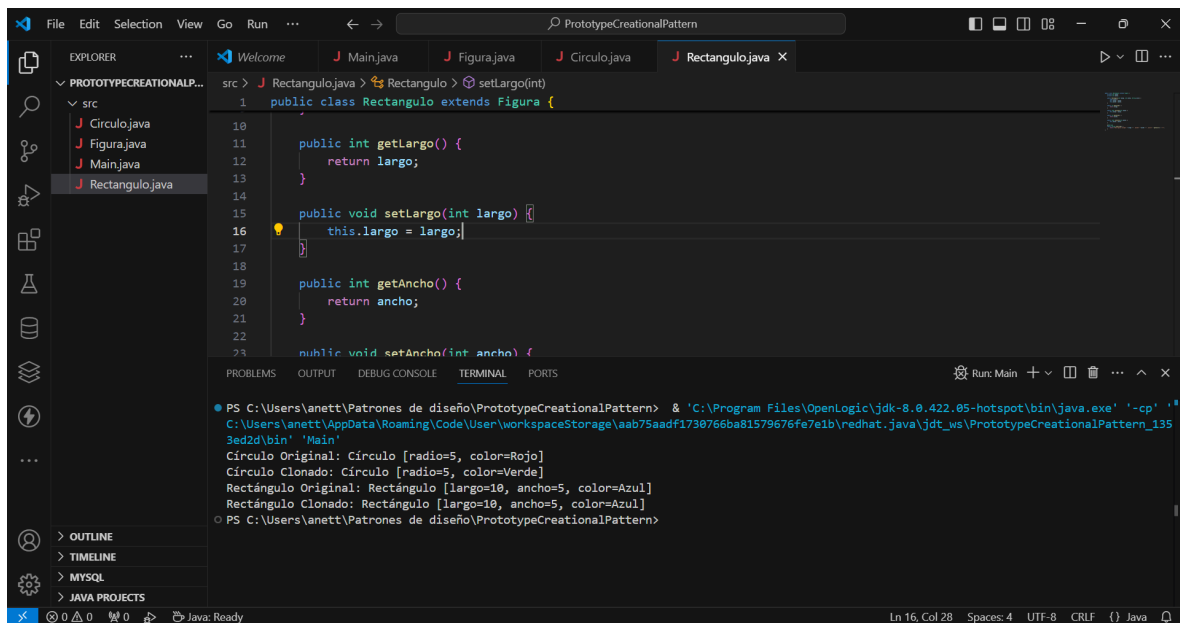
- EXPLORER:** Shows the project structure with 'BUILDERPATTERN' and 'src' folders. 'Casajava' and 'Main.java' are listed under 'src'.
- Editor:** Displays the code for 'Casajava'. It contains a 'public class Casa' and a 'public static class Builder'. The 'Builder' class has methods 'setPuestas(int puertas)', 'setTipoDeTecho(String tipoDeTecho)', and 'build()' which returns a new 'Casa' object.
- TERMINAL:** Shows the command to run the application: 'PS C:\Users\anett\Patrones de diseño\BuilderPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.05-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\3ccc6986412698e928d3774e186d9a28\redhat.java\jdt_ws\BuilderPattern_1f81923e\bin' 'Main''. The output shows 'Casa [ventanas=5, puertas=2, tipoDeTecho=Teja]'.

Patrón Prototype Creational

Qué es: El patrón Prototype permite crear nuevos objetos copiando una instancia existente en lugar de instanciarla directamente.

Cómo funciona aquí: La clase base Figura implementa la interfaz Cloneable y define el método clone(). Las clases concretas (Circulo y Rectangulo) heredan la capacidad de clonar.

Beneficio principal: Permite crear copias de objetos complejos de manera eficiente y sin depender de constructores.



```
src > J Rectangulo.java > Rectangulo > setLargo(int)
1 public class Rectangulo extends Figura {
10
11     public int getLargo() {
12         return largo;
13     }
14
15     public void setLargo(int largo) {
16         this.largo = largo;
17     }
18
19     public int getAncho() {
20         return ancho;
21     }
22
23     public void setAncho(int ancho) {
24
25     }
26 }
```

```
PS C:\Users\anett\Patrones de diseño\PrototypeCreationalPattern> & 'C:\Program Files\OpenLogic\jdk-8.0.422.05-hotspot\bin\java.exe' '-cp' 'C:\Users\anett\AppData\Roaming\Code\User\workspaceStorage\aab75aadf1738766ba81579676fe7e1b\redhat.java\jdt_ws\PrototypeCreationalPattern_1353ed2d\bin' 'Main'
Circulo Original: Circulo [radio=5, color=Rojo]
Circulo Clonado: Circulo [radio=5, color=Verde]
Rectangulo Original: Rectangulo [largo=10, ancho=5, color=Azul]
Rectangulo Clonado: Rectangulo [largo=10, ancho=5, color=Azul]
PS C:\Users\anett\Patrones de diseño\PrototypeCreationalPattern>
```