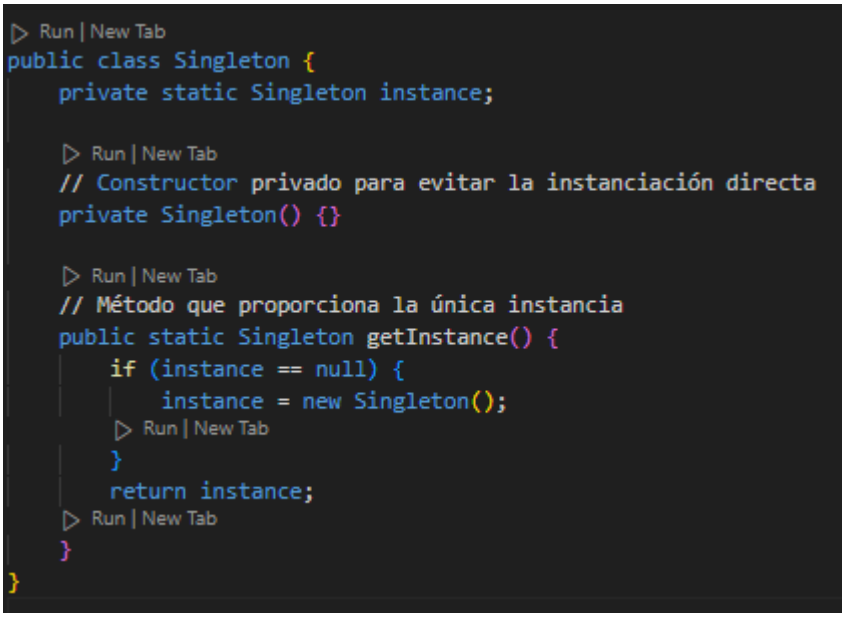
Patrones Creacionales

1. Singleton

Definición: El patrón Singleton es un patrón de diseño creacional que se utiliza para garantizar que una clase sólo tenga una única instancia y proporciona un punto de acceso global a dicha instancia. Este patrón es particularmente útil cuando un único objeto necesita coordinar acciones en todo el sistema. El Singleton asegura que solo haya una instancia disponible en todo el ciclo de vida de la aplicación, y controla el acceso mediante un método que devuelve la instancia única.

Uso típico: El patrón Singleton es comúnmente usado en situaciones como el manejo de conexiones a bases de datos, configuración global de aplicaciones, registros de log, entre otros. Estos son casos donde tener múltiples instancias podría llevar a inconsistencias o comportamientos inesperados.



2. Factory Method

Definición: El patrón Factory Method es un patrón de diseño creacional que define una interfaz para crear un objeto, pero permite que las subclases decidan cuál clase concreta instanciar. En lugar de instanciar objetos directamente, una clase delega la creación de objetos a subclases mediante la implementación de un método de fábrica. Esto permite que el código sea más flexible y fácil de mantener, ya que las subclases pueden cambiar la forma en que los objetos se crean sin modificar la clase base.

Uso típico: El patrón Factory Method se utiliza cuando una clase no puede prever el tipo de objetos que necesita crear, o cuando se desea que la lógica de creación de objetos esté

centralizada en una subclase específica. Es común en frameworks y bibliotecas que necesitan ser extendidos por usuarios para crear objetos específicos.

```java
// Producto abstracto
public abstract class Product {
    abstract void use();
}

// Productos concretos
public class ConcreteProductA extends Product {
    @Override
    void use() {
        System.out.println("Using Product A");
    }
}

public class ConcreteProductB extends Product {
    @Override
    void use() {
        System.out.println("Using Product B");
    }
}

// Creador abstracto
public abstract class Creator {
    public abstract Product factoryMethod();
}
```

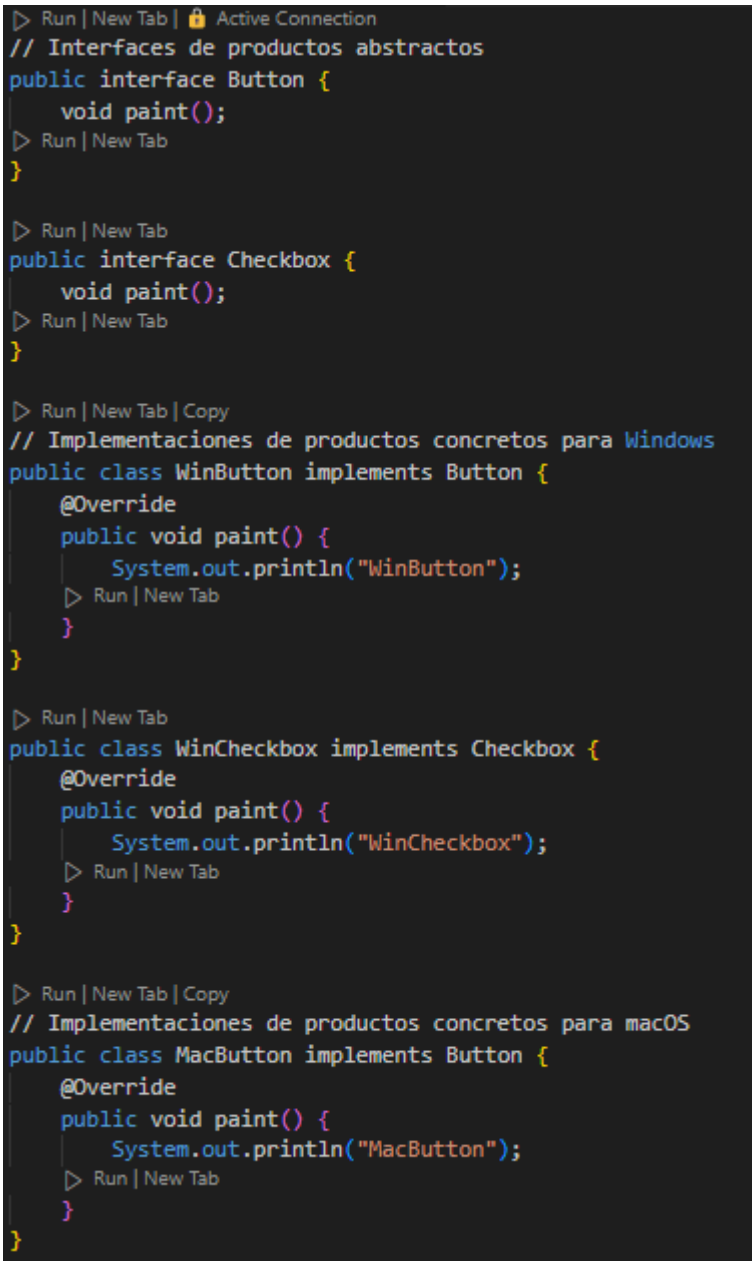
```java
// Creadores concretos
public class ConcreteCreatorA extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}

public class ConcreteCreatorB extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}
```

3. Abstract Factory

Definición: El patrón Abstract Factory es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Es una evolución del patrón Factory Method, donde en lugar de un solo método de fábrica, la fábrica abstracta define un conjunto de métodos para crear productos relacionados. Esto asegura que los productos creados sean compatibles entre sí y permite cambiar las familias de productos de manera fácil.

Uso típico: Abstract Factory es útil en sistemas que deben ser independientes de cómo se crean y ensamblan los productos. Es comúnmente utilizado en aplicaciones que requieren la creación de interfaces gráficas (GUI) en diferentes plataformas (Windows, macOS), donde cada plataforma tiene su propio conjunto de widgets.



```
Run | New Tab | Active Connection
// Interfaces de productos abstractos
public interface Button {
    void paint();
Run | New Tab
}

Run | New Tab
public interface Checkbox {
    void paint();
Run | New Tab
}

Run | New Tab | Copy
// Implementaciones de productos concretos para Windows
public class WinButton implements Button {
    @Override
    public void paint() {
        System.out.println("WinButton");
    Run | New Tab
    }
}

Run | New Tab
public class WinCheckbox implements Checkbox {
    @Override
    public void paint() {
        System.out.println("WinCheckbox");
    Run | New Tab
    }
}

Run | New Tab | Copy
// Implementaciones de productos concretos para macOS
public class MacButton implements Button {
    @Override
    public void paint() {
        System.out.println("MacButton");
    Run | New Tab
    }
}
```

```
Run | New Tab | Copy
// Fábrica concreta para macOS
public class MacFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacButton();
    Run | New Tab
    }

    Run | New Tab
    @Override
    public Checkbox createCheckbox() {
        return new MacCheckbox();
    Run | New Tab
    }
}
```

4. Builder

Definición: El patrón Builder es un patrón de diseño creacional que permite la construcción de objetos complejos paso a paso. En lugar de tener un constructor con múltiples parámetros opcionales, el patrón Builder proporciona una forma más clara y flexible de crear un objeto mediante la separación de su construcción de su representación. Esto es especialmente útil para objetos que requieren inicialización con muchos parámetros o que tienen una estructura compleja.

Uso típico: El patrón Builder se usa frecuentemente cuando un objeto puede ser creado de muchas maneras diferentes o cuando el proceso de construcción debe permitir que se pase por alto ciertos pasos. Es común en la construcción de objetos inmutables (por ejemplo, objetos `String` complejos) o en la creación de documentos, formularios, o configuraciones.

```java
public class Product {
    private final String part1;
    Run | New Tab
    private final String part2;

    Run | New Tab
    // Constructor privado que solo puede ser llamado por el Builder
    private Product(Builder builder) {
        this.part1 = builder.part1;
        Run | New Tab
        this.part2 = builder.part2;
    Run | New Tab
    }

    Run | New Tab
    // Clase interna estática que actúa como el Builder
    public static class Builder {
        private String part1;
        Run | New Tab
        private String part2;

        Run | New Tab
        public Builder part1(String part1) {
            this.part1 = part1;
            Run | New Tab
            return this;
        Run | New Tab
        }

        Run | New Tab
        public Builder part2(String part2) {
            this.part2 = part2;
            Run | New Tab
            return this;
        Run | New Tab
        }
```

```
        ▷ Run | New Tab
        // Método que devuelve el objeto Product completamente construido
        public Product build() {
            return new Product(this);
        ▷ Run | New Tab
        }
    }
}
```

5. Prototype

Definición: El patrón Prototype es un patrón de diseño creacional que se utiliza para crear nuevos objetos mediante la clonación de una instancia existente, evitando así la creación repetitiva de objetos costosos. Este patrón permite la creación de copias exactas de un objeto con la capacidad de modificar ligeramente las copias si es necesario. Es especialmente útil cuando la creación de un objeto es costosa en términos de recursos o tiempo.

Uso típico: El patrón Prototype es útil en sistemas donde la creación de objetos es compleja o costosa, y se desea crear múltiples instancias de estos objetos. Es común en juegos, aplicaciones gráficas, y sistemas que requieren la creación rápida de objetos similares.

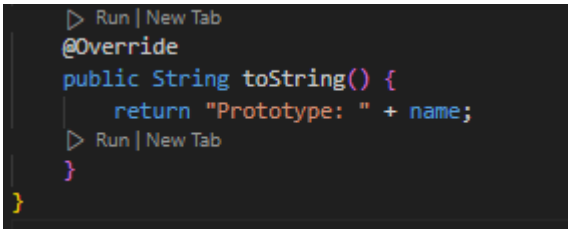
```
public abstract class Prototype implements Cloneable {
    abstract Prototype clonePrototype();

    ▷ Run | New Tab
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    ▷ Run | New Tab
    }
}

▷ Run | New Tab
public class ConcretePrototype extends Prototype {
    private String name;

    ▷ Run | New Tab
    public ConcretePrototype(String name) {
        this.name = name;
    ▷ Run | New Tab
    }

    ▷ Run | New Tab
    @Override
    public Prototype clonePrototype() {
        try {
            return (Prototype) this.clone();
        ▷ Run | New Tab
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            ▷ Run | New Tab
            return null;
        ▷ Run | New Tab
        }
    }
```


Run | New Tab
@Override
public String toString() {
return "Prototype: " + name;
Run | New Tab
}
}