UNIVERSIDAD NEBRIJA

Técnicas de programación avanzada

Curso 2023-2024

Tema 3.2 Patrones creacionales

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

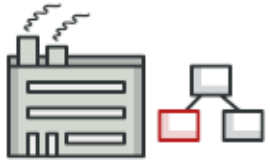
5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

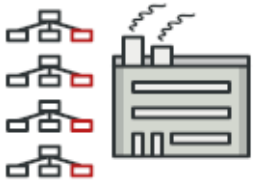
3.2 Patrones de creación

Introducción



Factory Method

Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

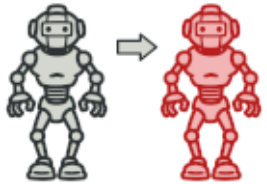


Abstract Factory

Permite producir familias de objetos relacionados sin especificar sus clases concretas.

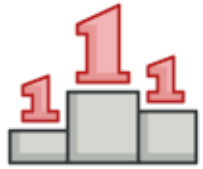
Builder

Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



Prototype

Permite copiar objetos existentes sin que el código dependa de sus clases.



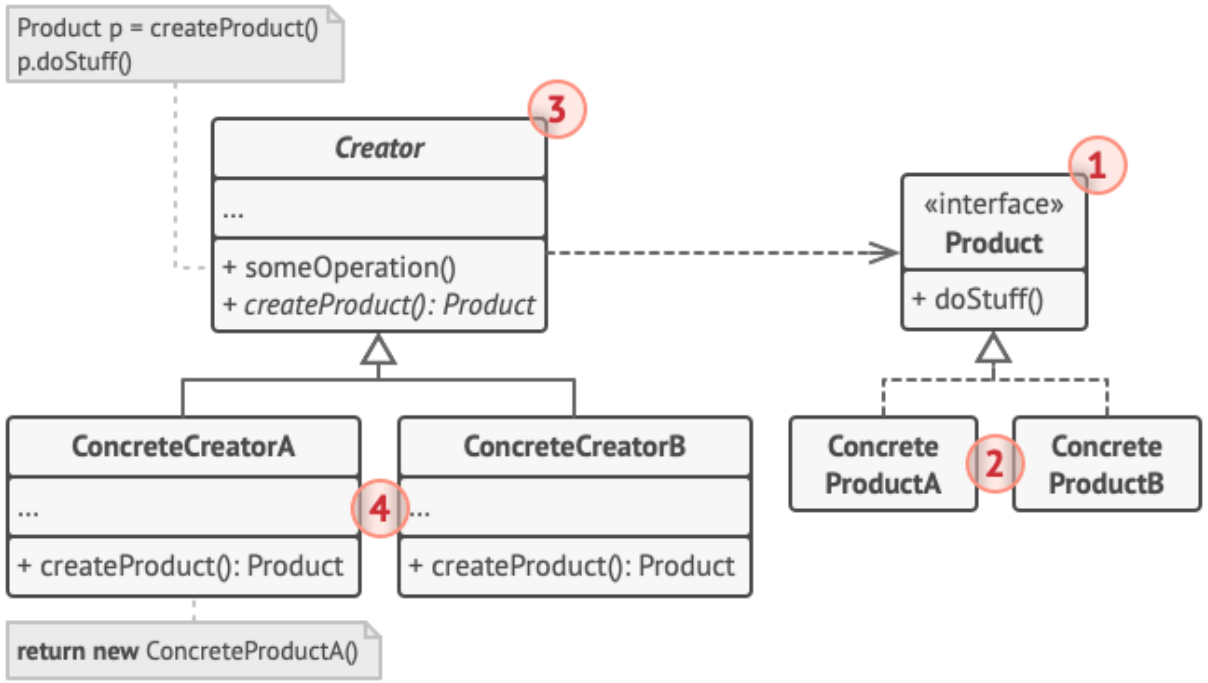
Singleton

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

3.2 Patrones Creacionales

FACTORY METHOD

Factory Method. Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Este patrón delega en las subclases la decisión de qué objeto crear.

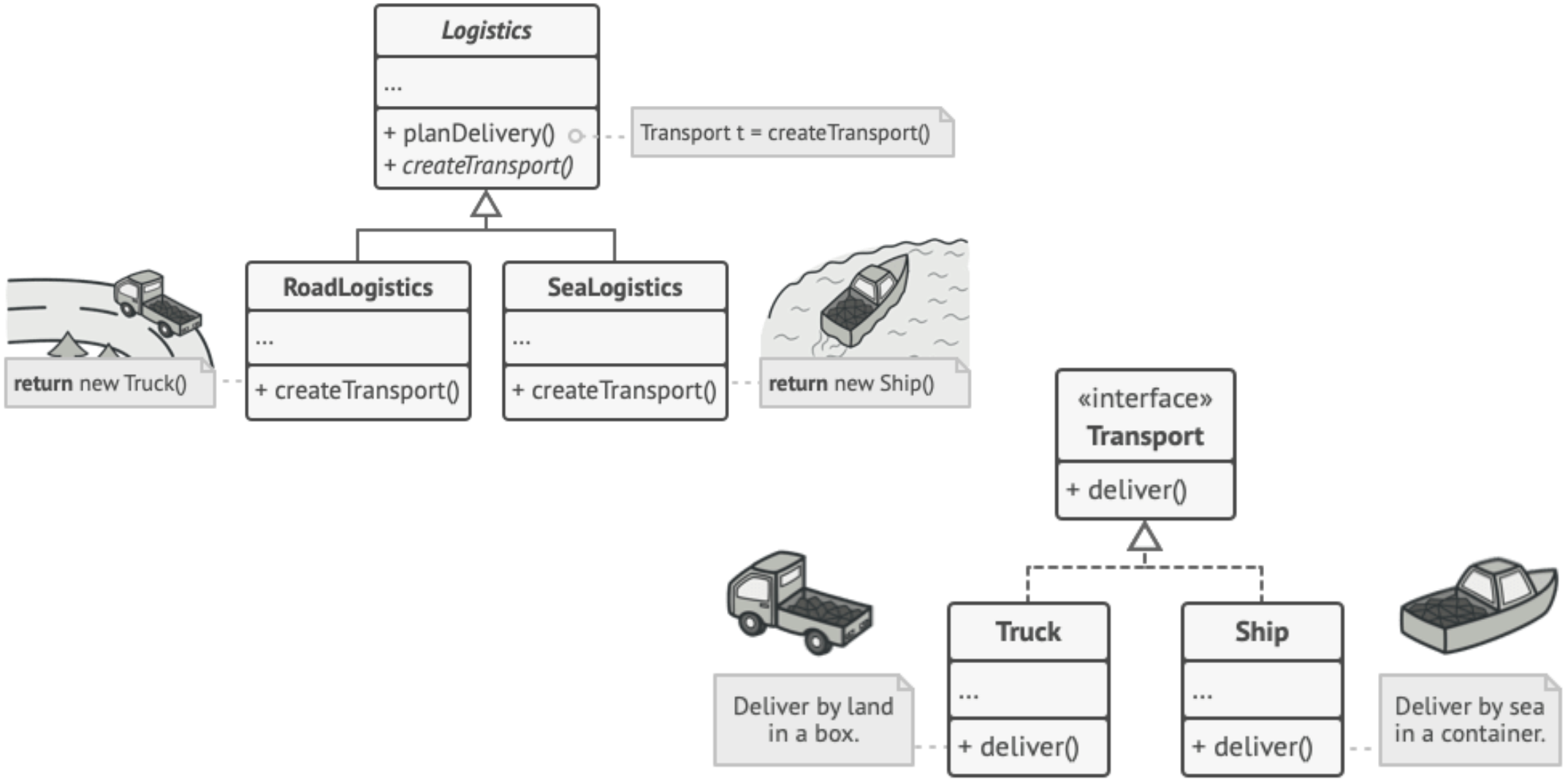


1. **Definir una Interfaz de Producto:** Comienza por definir una interfaz o una clase abstracta que declare los métodos que todos los objetos "producto" concretos deben implementar. Este producto es lo que el factory method eventualmente creará.
2. **Crear Implementaciones Concretas del Producto:** Desarrolla una o más implementaciones concretas de la interfaz del producto. Estas son las clases que el Factory Method instanciará.
3. **Declarar la Interfaz del Creador:** Define una interfaz o clase abstracta (el "Creador") que declare el método factory. Este método es el que retornará objetos de tipo producto. A menudo, el método factory es abstracto y debe ser implementado por las subclases concretas.
4. **Implementar Creadores Concretos:** Crea clases concretas que extiendan la interfaz del Creador. Estas implementaciones concretas del Creador sobrescribirán el método factory para crear y retornar objetos de producto concreto.
5. **Utilizar el Factory Method:** En tu código, reemplaza las instancias de creación de objeto directa (usando new) con llamadas al Factory Method. Esto te permitirá cambiar el tipo de producto creado simplemente cambiando la instancia del creador que estás utilizando.

https://refactoring.guru/es/design-patterns/factory-method

3.2 Patrones Creacionales

FACTORY METHOD



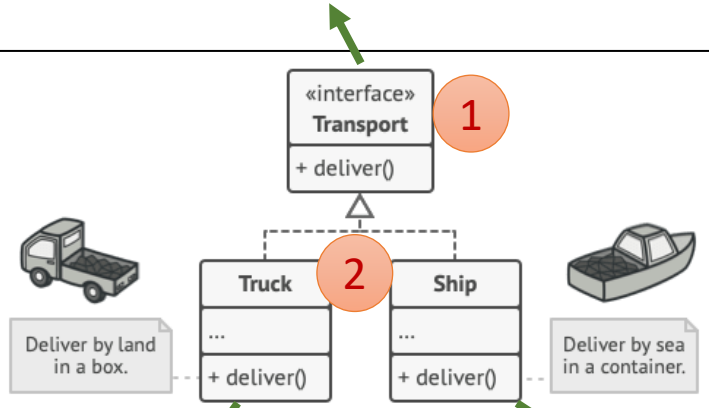
https://refactoring.guru/es/design-patterns/factory-method

3.2 Patrones Creacionales

FACTORY METHOD

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/a_Patron_FactoryMethod

```java
/**
 * Interfaz Transport que define la funcionalidad de los transportes.
 */
public interface Transport {
    void deliver();
}
```



1. **Definir una Interfaz de Producto:** Comienza por definir una interfaz o una clase abstracta que declare los métodos que todos los objetos "producto" concretos deben implementar. Este producto es lo que el factory method eventualmente creará.
2. **Crear Implementaciones Concretas del Producto:** Desarrolla una o más implementaciones concretas de la interfaz del producto. Estas son las clases que el Factory Method instanciará.

```java
/**
 * Clase concreta Truck que implementa Transport.
 */
class Truck implements Transport {

    @Override
    public void deliver() {
        System.out.println("Entrega por camión");
    }
}
```

```java
/**
 * Clase concreta Ship que implementa Transport.
 */
class Ship implements Transport {

    @Override
    public void deliver() {
        System.out.println("Entrega por barco");
    }
}
```

3.2 Patrones Creacionales

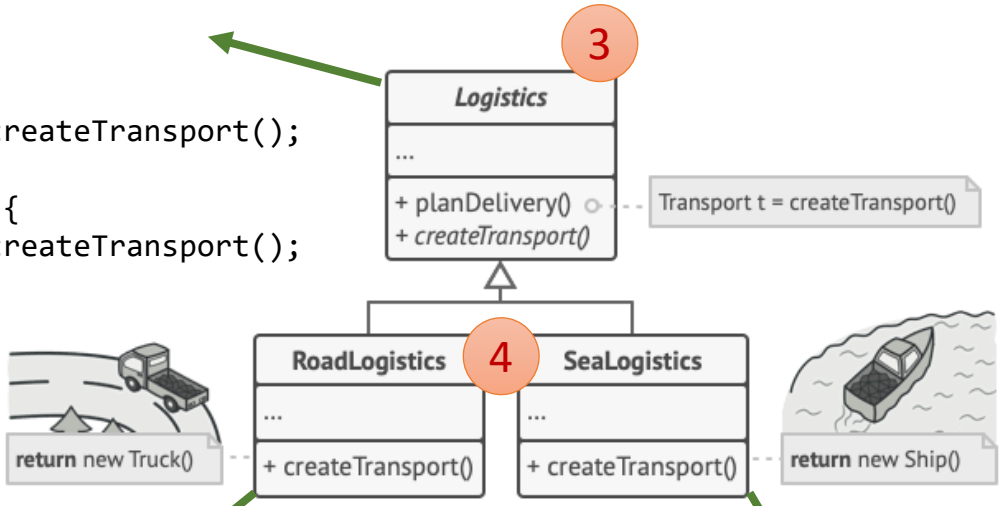
FACTORY METHOD

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/a_Patron_FactoryMethod

```java
/**
 * Clase abstracta Logistics que define el método factory.
 */
abstract class Logistics {

    public abstract Transport createTransport();

    public void planDelivery() {
        Transport transport = createTransport();
        transport.deliver();
    }
}
```



3. **Declarar la Interfaz del Creador:** Define una interfaz o clase abstracta (el "Creador") que declare el método factory. Este método es el que retornará objetos de tipo producto. A menudo, el método factory es abstracto y debe ser implementado por las subclases concretas.
4. **Implementar Creadores Concretos:** Crea clases concretas que extiendan la interfaz del Creador. Estas implementaciones concretas del Creador sobrescribirán el método factory para crear y retornar objetos de producto concreto.

```java
/**
 * Clase RoadLogistics que crea instancias de Truck.
 */
class RoadLogistics extends Logistics {

    @Override
    public Transport createTransport() {
        return new Truck();
    }
}
```

```java
/**
 * Clase SeaLogistics que crea instancias de Ship.
 */
class SeaLogistics extends Logistics {

    @Override
    public Transport createTransport() {
        return new Ship();
    }
}
```

3.2 Patrones Creacionales

FACTORY METHOD

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/a_Patron_FactoryMethod

5. **Utilizar el Factory Method:** En tu código, reemplaza las instancias de creación de objeto directa (usando new) con llamadas al Factory Method. Esto te permitirá cambiar el tipo de producto creado simplemente cambiando la instancia del creador que estás utilizando.

```
/**
 * Clase MainClass para probar el patrón Factory Method.
 */
public class MainClass {
    public static void main(String[] args) {
        Logistics logistics = new RoadLogistics();
        logistics.planDelivery();

        logistics = new SeaLogistics();
        logistics.planDelivery();
    }
}
```

3.2 Patrones Creacionales

FACTORY METHOD

El patrón Factory Method es ampliamente utilizado en aplicaciones reales donde se requiere flexibilidad y extensibilidad en la creación de objetos. Algunos escenarios comunes en los que es recomendable usar este patrón incluyen:

- **Cuando la Clase Exacta del Objeto no se Conoce Hasta el Tiempo de Ejecución:** El Factory Method es ideal en situaciones donde no sabes con anticipación qué clase específica necesitarás instanciar. Por ejemplo, en un software de procesamiento de documentos, podrías necesitar crear diferentes tipos de procesadores de texto o lectores de documentos basados en el formato del archivo (PDF, Word, etc.), y esta información solo se conoce en tiempo de ejecución.

- **Cuando se Requiere Desacoplar la Creación de Objetos del Uso de los Mismos:** El patrón permite desacoplar el código que crea objetos del código que los utiliza, mejorando así la modularidad y la mantenibilidad del código. Esto es útil en sistemas complejos como frameworks o bibliotecas, donde los detalles de la implementación de un objeto pueden cambiar frecuentemente.

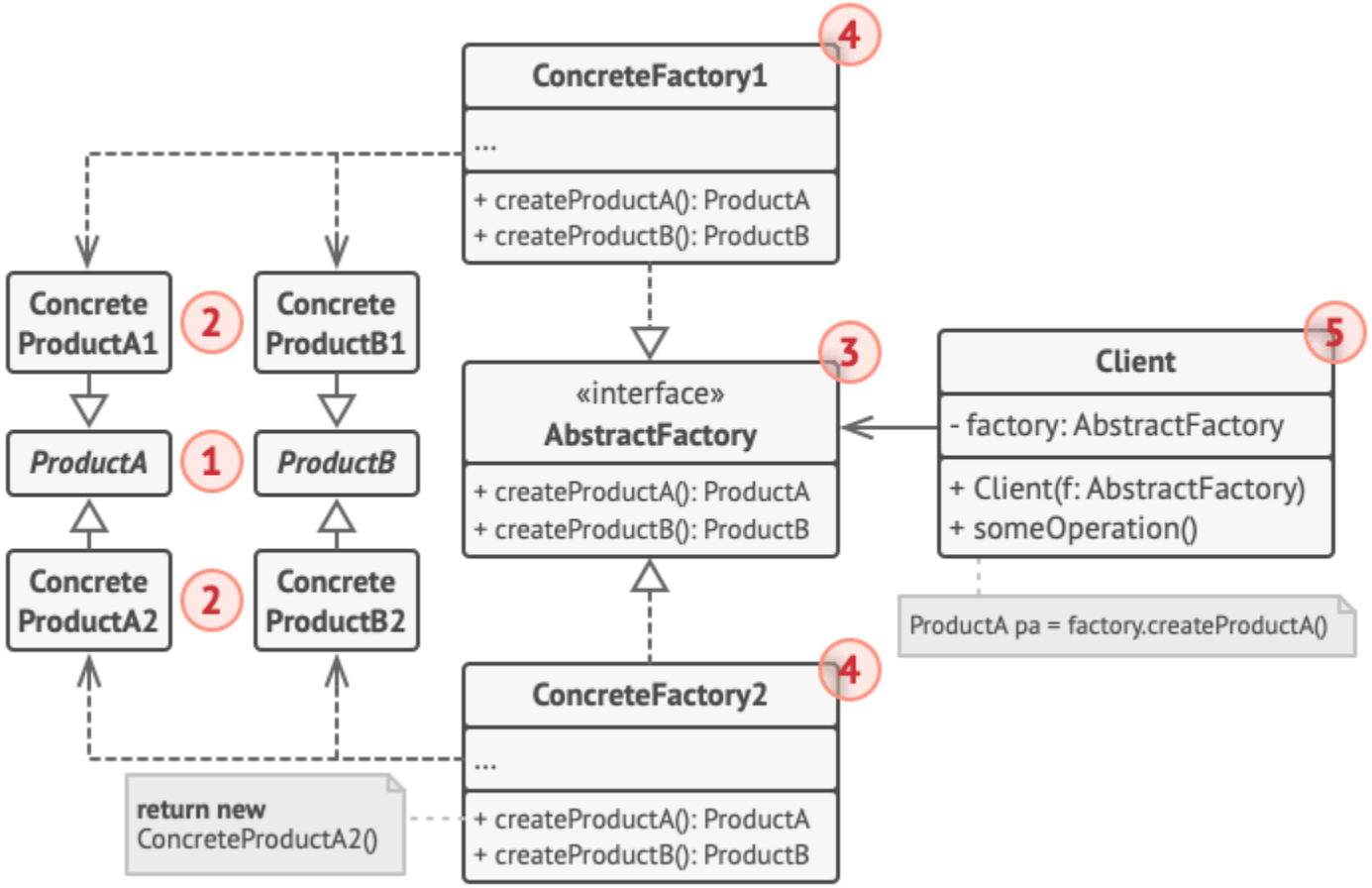
- **En Aplicaciones que Necesitan Extensibilidad:** Si estás desarrollando una aplicación que necesita ser extensible, donde otros desarrolladores pueden necesitar agregar nuevas clases y formas de crear objetos, el Factory Method proporciona un punto claro de extensión.

- **Para Encapsular la Lógica de Creación de Objetos:** En sistemas donde la creación de objetos es compleja e involucra lógica de negocio o configuraciones, centralizar esta lógica en un factory method mejora la claridad y la reutilización del código.

3.2 Patrones Creacionales

ABSTRACT FACTORY

Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



(1) Declaramos de forma explícita interfaces para cada producto diferente de la familia de productos

(2) Después podemos hacer que todas las variantes de los productos sigan esas interfaces.

(3) El siguiente paso consiste en declarar la *Fábrica abstracta*: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos. Estos métodos deben devolver productos **abstractos** representados por las interfaces que extrajimos previamente.

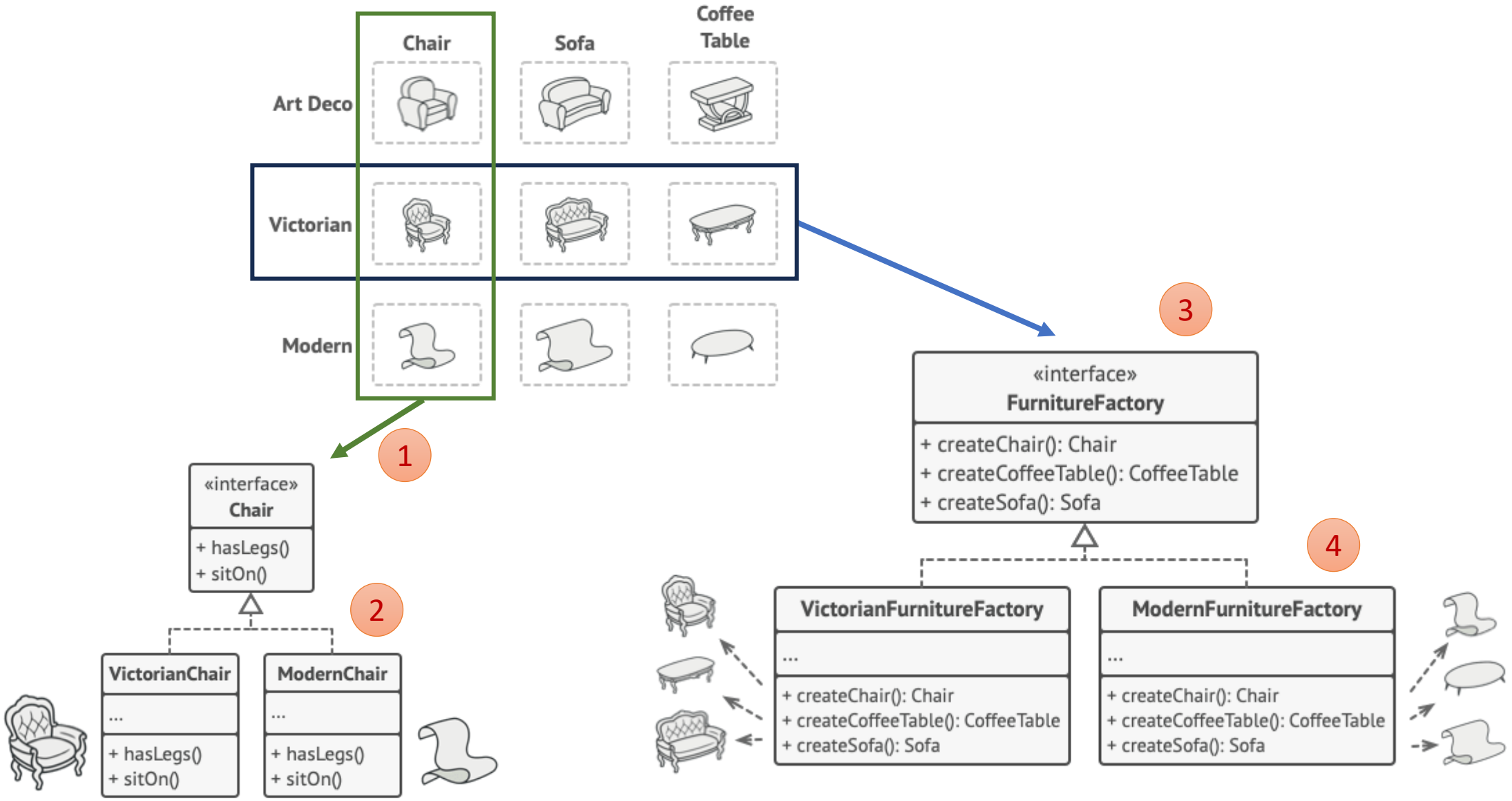
(4) Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz FábricaAbstracta

https://refactoring.guru/es/design-patterns/abstract-factory

3.2 Patrones Creacionales

ABSTRACT FACTORY

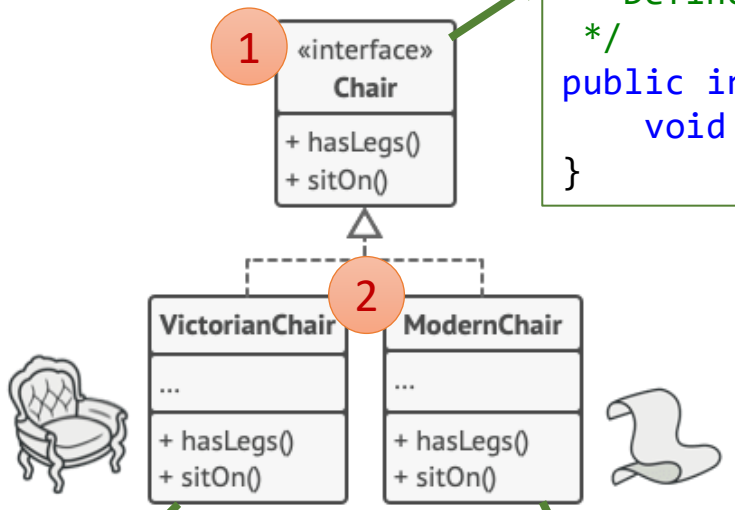
https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/b_Patron_AbstractFactory



3.2 Patrones Creacionales

ABSTRACT FACTORY

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/b_Patron_AbstractFactory




```java
/**
 * Interfaz para el producto Silla.
 * Define comportamientos para las sillas.
 */
public interface Chair {
    void describe();
}
```

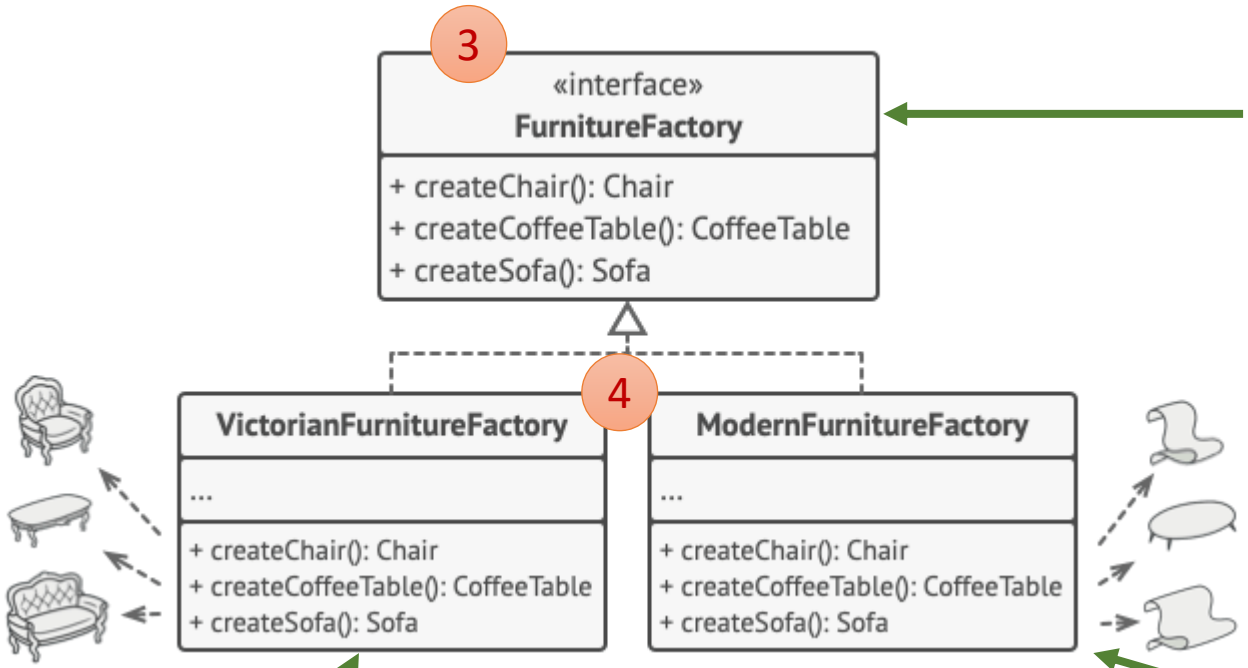
```java
/**
 * Implementación concreta de una silla de estilo victoriano.
 */
class VictorianChair implements Chair {
    public void describe() {
        System.out.println("Una silla victoriana.");
    }
}
```

```java
/**
 * Implementación concreta de una silla de estilo moderno.
 */
class ModernChair implements Chair {
    public void describe() {
        System.out.println("Una silla moderna.");
    }
}
```

3.2 Patrones Creacionales

ABSTRACT FACTORY

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/b_Patron_AbstractFactory




```java
/**
 * Interfaz principal para la fábrica de muebles.
 * Define métodos para crear distintos tipos de muebles.
 */
public interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
    CoffeeTable createCoffeeTable();
}
```

```java
class VictorianFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new VictorianChair();
    }
    public Sofa createSofa() {
        return new VictorianSofa();
    }
    public CoffeeTable createCoffeeTable() {
        return new VictorianCoffeeTable();
    }
}
```

```java
/**
 * Fábrica concreta para crear muebles de estilo moderno.
 */
class ModernFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ModernChair();
    }
    public Sofa createSofa() {
        return new ModernSofa();
    }
    public CoffeeTable createCoffeeTable() {
        return new ModernCoffeeTable();
    }
}
```

3.2 Patrones Creacionales
ABSTRACT FACTORY

El patrón Abstract Factory es útil en diferentes contextos de aplicaciones reales, especialmente en situaciones donde es necesario crear familias de objetos relacionados sin especificar sus clases concretas. Algunos ejemplos de aplicaciones reales donde se recomienda el uso del patrón Abstract Factory incluyen:

- **Desarrollo de Interfaces de Usuario Multiplataforma:** En aplicaciones que deben funcionar en diferentes plataformas (como Windows, macOS, Linux), un Abstract Factory puede ser utilizado para crear elementos de la interfaz de usuario que son específicos de cada plataforma, como ventanas, botones y cuadros de diálogo, sin modificar el código principal de la aplicación.

- **Creación de Conjuntos de Objetos que Comparten un Tema Común:** Este patrón es ideal para situaciones en las que se necesitan conjuntos de objetos que comparten un tema común o estilo, como en aplicaciones de diseño gráfico o videojuegos, donde diferentes temas pueden requerir diferentes representaciones gráficas o comportamientos de los objetos.

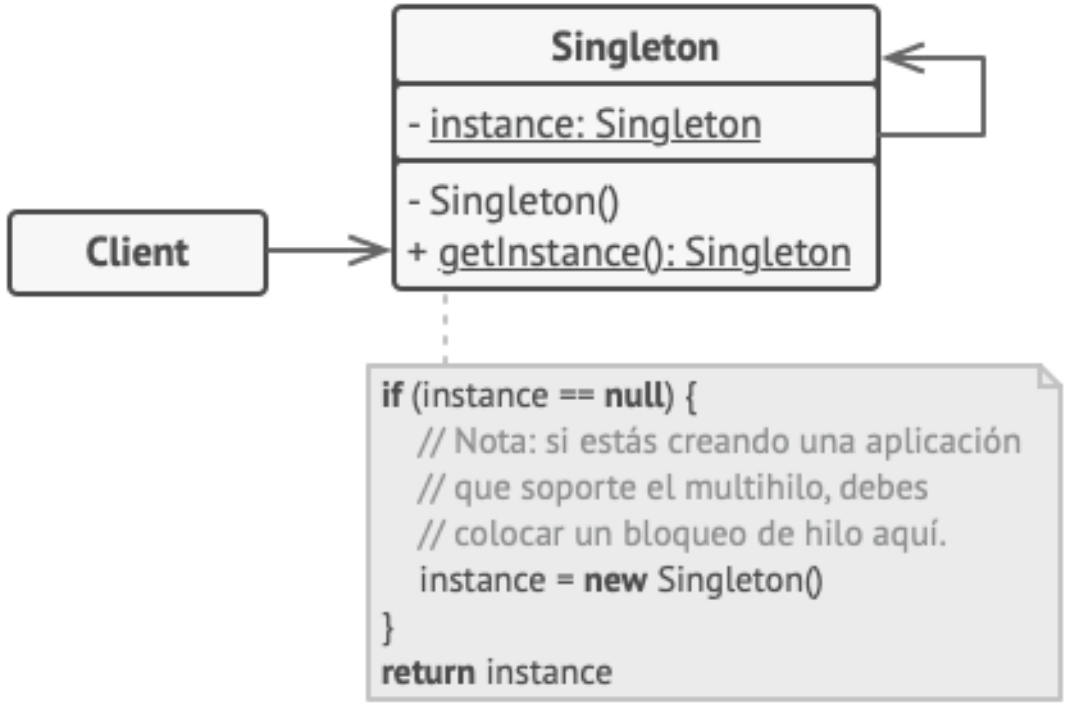
- **Desarrollo de Aplicaciones con Múltiples Variantes de Producto:** En situaciones donde una aplicación necesita soportar múltiples variantes del mismo producto, que podrían incluir diferentes combinaciones de subcomponentes, el Abstract Factory permite gestionar estas variantes de manera más ordenada y modular.

- **Creación de Objetos en Juegos o Simulaciones:** En el desarrollo de juegos o simulaciones, donde diferentes escenarios o ambientes requieren diferentes tipos de objetos con comportamientos o apariencias distintas, como diferentes tipos de enemigos, herramientas, o ambientes.

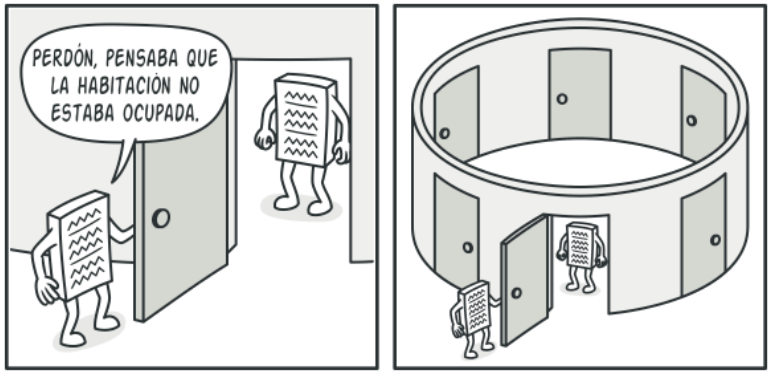
3.2 Patrones Creacionales

SINGLETON

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



Este patrón es útil cuando necesitas asegurarte de que solo exista una instancia de una clase a lo largo de toda la aplicación, como en el caso de un controlador de configuración o una conexión a una base de datos.



https://refactoring.guru/es/design-patterns/singleton

3.2 Patrones Creacionales

SINGLETON

https://github.com/aalonsopuig/Java_Ejemplos_3/tree/main/src/c_Patron_Singleton

Cómo implementarlo

1. Añade un campo estático privado a la clase para almacenar la instancia Singleton.
2. Declara un método de creación estático público para obtener la instancia Singleton.
3. Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.
4. Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.
5. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

```java
/**
 * Clase MainClass para probar el patrón Singleton.
 */
public class MainClass {
    public static void main(String[] args) {
        Singleton miSingleton = Singleton.obtenerInstancia();
        miSingleton.hacerAlgo();
    }
}
```

```java
public class Singleton {

    // La única instancia de Singleton que será creada.
    private static Singleton instancia;

    /**
     * Constructor privado para evitar la instanciación externa.
     */
    private Singleton() {
        // Inicialización de la instancia, si es necesario.
    }

    /**
     * Método para obtener la única instancia de la clase.
     * @return la única instancia de Singleton.
     */
    public static synchronized Singleton obtenerInstancia() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }

    /**
     * Método de ejemplo para demostrar funcionalidad.
     */
    public void hacerAlgo() {
        System.out.println("Haciendo algo...");
    }
}
```

El patrón Singleton es útil en varias situaciones en aplicaciones reales, especialmente cuando es necesario asegurar que una clase tenga una única instancia a lo largo de toda la aplicación. Algunos ejemplos de uso del patrón Singleton en aplicaciones reales incluyen:

1. **Controladores de Configuración:** En aplicaciones donde se necesita un único punto de control para la configuración global, el Singleton puede ser utilizado para garantizar que exista una única instancia de la configuración, accesible globalmente.

2. **Conexiones a Bases de Datos:** Para gestionar conexiones a bases de datos, el Singleton puede ser útil para evitar la sobrecarga de abrir y cerrar conexiones múltiples veces. Una única instancia de un gestor de conexiones puede ser reutilizada a lo largo de la aplicación.

3. **Manejo de Recursos del Sistema:** En situaciones donde se requiere un manejo centralizado de recursos del sistema, como archivos de log, conexiones a servidores o gestores de caché, el Singleton asegura un único punto de acceso y control.

4. **Acceso a Interfaces de Hardware o Servicios Externos:** Para el acceso a hardware específico o servicios externos (como impresoras, dispositivos de entrada, servicios web), el Singleton puede ser utilizado para gestionar el acceso y asegurar un uso coherente y controlado.

Ejercicios

3.2 Patrones Creacionales

Ejercicios

Enunciado del Ejercicio: **Fábrica de Reproductores Multimedia**

Desarrollar un programa que utilice el patrón **Abstract Factory** para crear diferentes tipos de reproductores multimedia. Los reproductores pueden ser de audio o de video, y cada tipo tendrá dos variantes: Básico y Avanzado.

Pasos a seguir:

1. Definir Interfaces de Productos:
   - Crear una interfaz AudioPlayer con un método playAudio().
   - Crear una interfaz VideoPlayer con un método playVideo().
2. Implementar Productos Concretos:
   - Desarrollar clases concretas BasicAudioPlayer y AdvancedAudioPlayer que implementen AudioPlayer.
   - Desarrollar clases concretas BasicVideoPlayer y AdvancedVideoPlayer que implementen VideoPlayer.
3. Definir la Interfaz Abstracta de la Fábrica:
   - Crear una interfaz MediaPlayerFactory con métodos para crear instancias de AudioPlayer y VideoPlayer.
4. Implementar Fábricas Concretas:
   - Desarrollar la clase BasicMediaPlayerFactory que cree reproductores de audio y video básicos.
   - Desarrollar la clase AdvancedMediaPlayerFactory que cree reproductores de audio y video avanzados.
5. Demostrar el Uso de las Fábricas:
   - Escribir una clase MediaPlayerTest que utilice ambas fábricas (BasicMediaPlayerFactory y AdvancedMediaPlayerFactory) para demostrar la creación y el uso de los diferentes tipos de reproductores multimedia.

3.2 Patrones Creacionales

Ejercicios

Enunciado del Ejercicio: **Fábrica de Bebidas**

Desarrollar un programa que utilice el patrón **Factory Method** para crear diferentes tipos de bebidas. Las bebidas pueden ser de diferentes tipos, como café, té o chocolate caliente, y cada tipo de bebida se preparará de manera diferente.

Pasos a seguir:

1. Crear una interfaz **Beverage** que defina los métodos **prepare**() y **serve**().
2. Implementar diferentes tipos de bebidas como clases concretas (**Coffee**, **Tea**, **HotChocolate**), cada una implementando la interfaz **Beverage**.
3. Desarrollar una clase abstracta **BeverageFactory** con un método factory **createBeverage**().
4. Implementar fábricas concretas para cada tipo de bebida (**CoffeeFactory**, **TeaFactory**, **HotChocolateFactory**).
5. Escribir una clase de prueba que demuestre el uso de las fábricas para crear diferentes tipos de bebidas.

Bibliografía

- Gamma, E., Johnson, R., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

- **Sumérgete en los patrones de diseño.** V2021-1.7. Alexander Shvets.

  https://refactoring.guru/es/design-patterns/book

  Versión online: https://refactoring.guru/es/design-patterns/catalog

- Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc.".

- Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, *1*(34), 597.

Técnicas de Programación Avanzada

```
exit(); //Gracias!
```