UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Sesión 14: Introducción a Patrones GoF

Recordando:

¿Que vimos la clase pasada?

Universidad Tecnológica del Perú

Logro de aprendizaje

Al finalizar la sesión, el estudiante soluciona problemas aplicando Patrones GoF usando Java en la resolución de ejercicios.

Universidad Tecnológica del Perú

Utilidad

Los Patrones GoF son herramientas poderosas que no solo facilitan la implementación de diseños robustos y eficientes, sino que también mejoran la comunicación y la colaboración entre desarrolladores, promoviendo la creación de software de alta calidad y mantenible.

Agenda

- ¿Qué son los Patrones de Diseño?
- Patrones GoF: ¿Qué son y quiénes son GoF?
- Tipos de Patrones GoF

Universidad Tecnológica del Perú

Saberes Previos

- Conceptos básicos de programación orientada a objetos.
- Familiaridad con el lenguaje de programación (preferiblemente Java).
- Comprender la importancia de la reutilización de código y la mantenibilidad del software.

Definición de la importancia del diseño de software eficiente y reutilizable

En el mundo de la tecnología, la creación de software eficiente, confiable y mantenible es esencial.

El diseño de software no es solo escribir código, sino también planificar cuidadosamente cómo se estructura y organiza ese código.

Esto es crucial para evitar problemas futuros y permitir la expansión y modificación del software de manera eficiente.

Introducción a la idea de los Patrones de Diseño como soluciones probadas para problemas comunes en programación:

Los Patrones de Diseño son soluciones probadas y documentadas para problemas comunes en programación.

Son como recetas que los desarrolladores pueden seguir para abordar problemas específicos de diseño de software.

Esto ayuda a estandarizar y simplificar el proceso de desarrollo, ya que no es necesario reinventar la rueda cada vez que se enfrenta a un problema conocido.

Los Patrones GoF son un conjunto de patrones de diseño ampliamente aceptados y documentados

Estos patrones son conocidos como los "Gang of Four" y son ampliamente aceptados y respetados en la comunidad de desarrollo de software.
A lo largo de la presentación, exploraremos estos patrones en detalle y comprenderemos por qué son tan influyentes en la industria.

¿Qué son los Patrones de Diseño?

Los Patrones de Diseño son soluciones probadas y documentadas para problemas comunes en el diseño de software.
Son enfoques y estrategias que los desarrolladores de software han ideado a lo largo de los años para abordar desafíos recurrentes. Estos patrones son como recetas que puedes seguir para resolver problemas específicos de diseño en tu código.
Utilizarlos no solo te ayuda a resolver esos problemas, sino que también te permite escribir software más eficiente, mantenible y reutilizable.

¿Qué son los Patrones de Diseño?

https://www.youtube.com/watch?v=qHuI_IWUkfA

Universidad Tecnológica del Perú

Importancia de los Patrones de Diseño

- **Reutilización de Soluciones Probadas:**

En lugar de reinventar la rueda cada vez que te encuentras con un problema común, puedes recurrir a un Patrón de Diseño probado y documentado. Esto ahorra tiempo y esfuerzo, ya que no tienes que diseñar una solución desde cero.

- **Mantenibilidad del Software:**

Los Patrones de Diseño promueven la creación de código limpio y estructurado. Esto facilita la comprensión y el mantenimiento del software a lo largo del tiempo, lo que es esencial para proyectos a largo plazo.

Importancia de los Patrones de Diseño

- **Comunicación Efectiva:**

Los Patrones de Diseño proporcionan un lenguaje común entre los desarrolladores. Cuando todos están familiarizados con los mismos patrones, la comunicación y la colaboración se vuelven más efectivas.

- **Escalabilidad y Flexibilidad:**

Los Patrones de Diseño ayudan a diseñar software que puede adaptarse y crecer con el tiempo.

Esto es crucial en un mundo donde los requisitos del software pueden cambiar constantemente.

Ejemplos de Problemas Resueltos por Patrones de Diseño

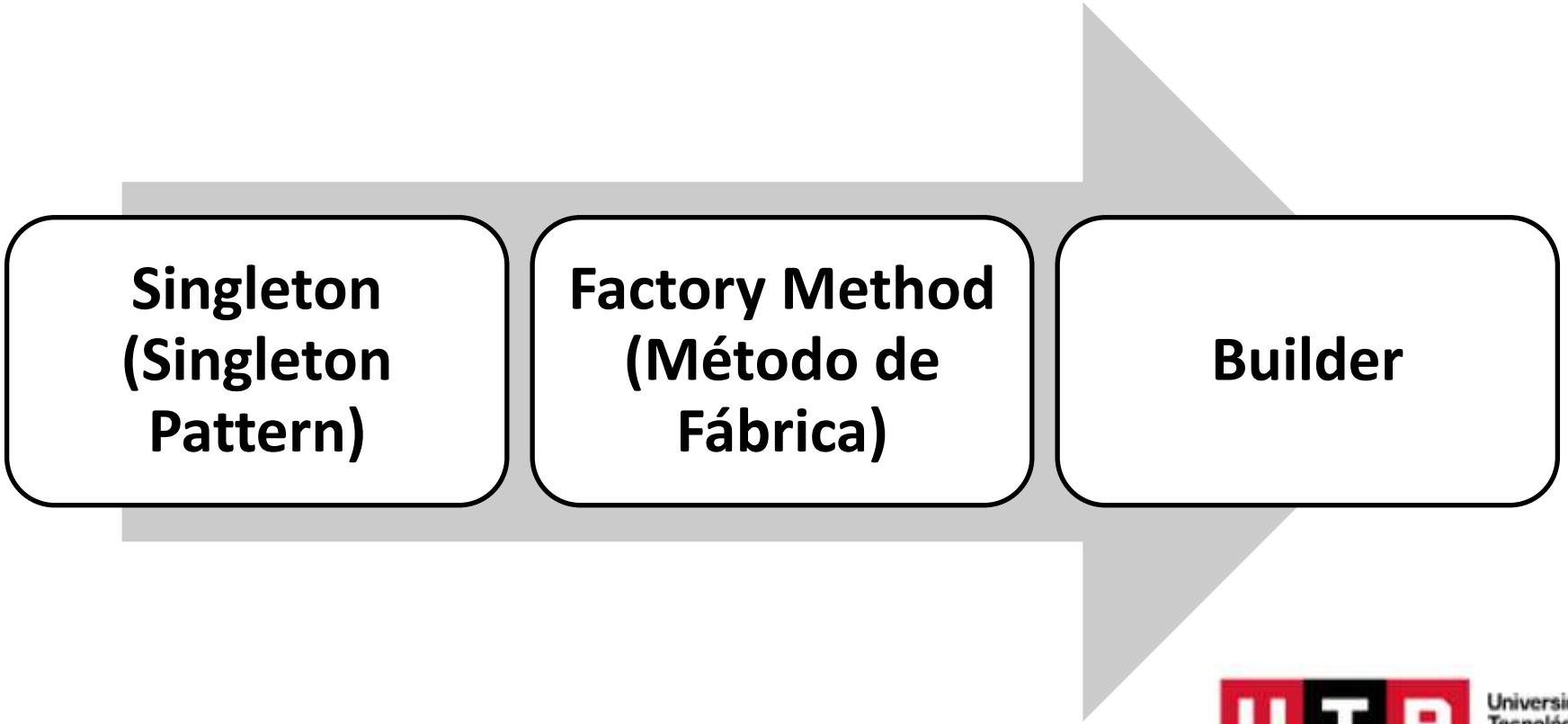
Para ilustrar mejor la utilidad de los Patrones de Diseño, aquí hay algunos ejemplos de problemas comunes que pueden resolverse mediante el uso de patrones:

- **Singleton:** Utilizado cuando necesitas garantizar que una clase tenga una única instancia y proporcionar un punto de acceso global a esa instancia.
- **Factory Method:** Útil cuando deseas delegar la creación de objetos a sus subclases, permitiendo que una clase principal sea independiente de las clases concretas que crea.

Ejemplos de Problemas Resueltos por Patrones de Diseño

- **Observer:** Ideal para implementar una relación uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos que dependen de él sean notificados y actualizados automáticamente.

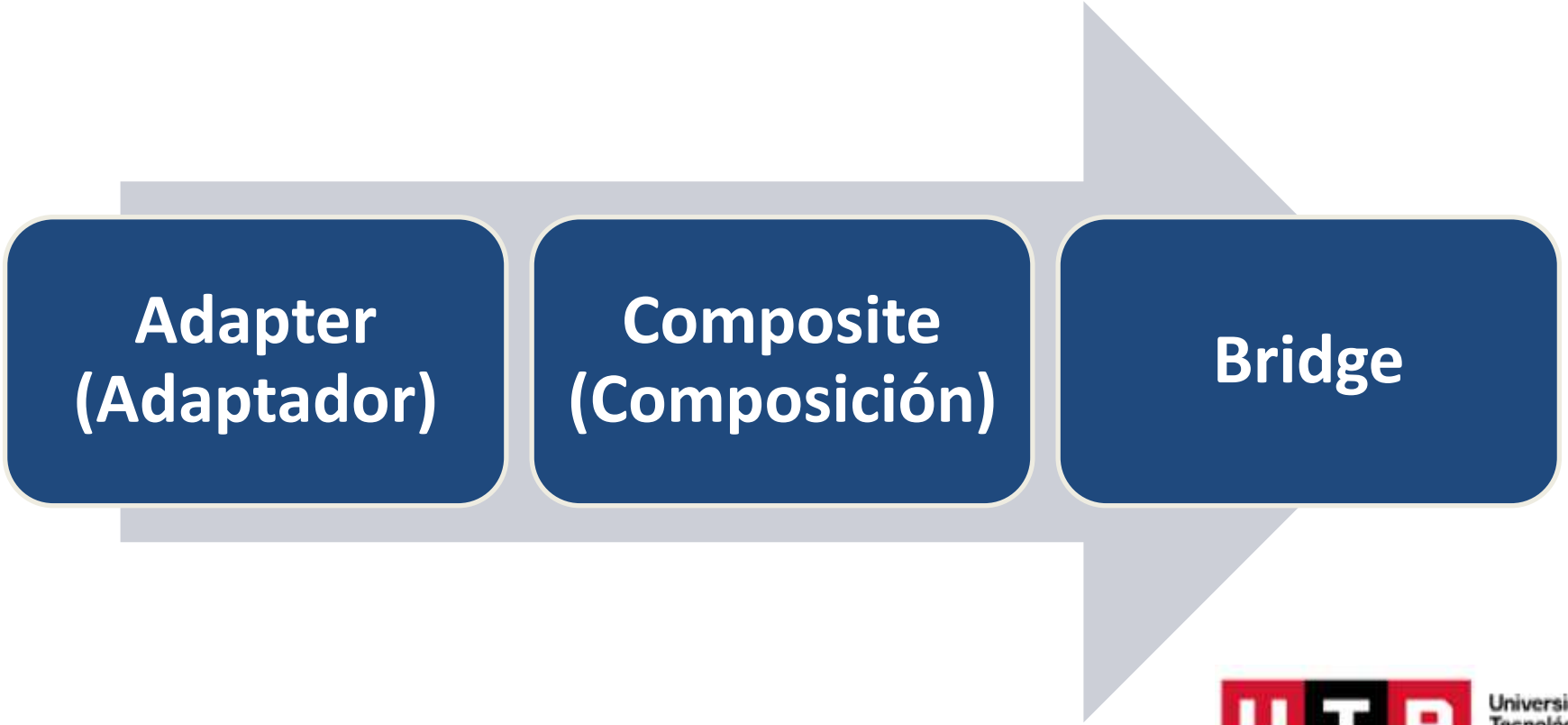
Tipos de Patrones GoF-Patrones Creacionales



Tipos de Patrones GoF-Patrones Creacionales

- **Singleton (Singleton Pattern):** Este patrón garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Es útil cuando se necesita compartir una única instancia de una clase en todo el sistema, como un registro central o una conexión a una base de datos.
- **Factory Method (Método de Fábrica):** El patrón Factory Method delega la creación de objetos a sus subclases, permitiendo que una clase principal sea independiente de las clases concretas que crea. Esto es valioso cuando se necesita crear objetos sin especificar la clase exacta de objeto que se creará.
- **Builder:** El patrón permite construir objetos complejos paso a paso. Se utiliza para construir un objeto compuesto asegurando que el proceso de construcción sea independiente de las partes que componen el objeto y de su representación.

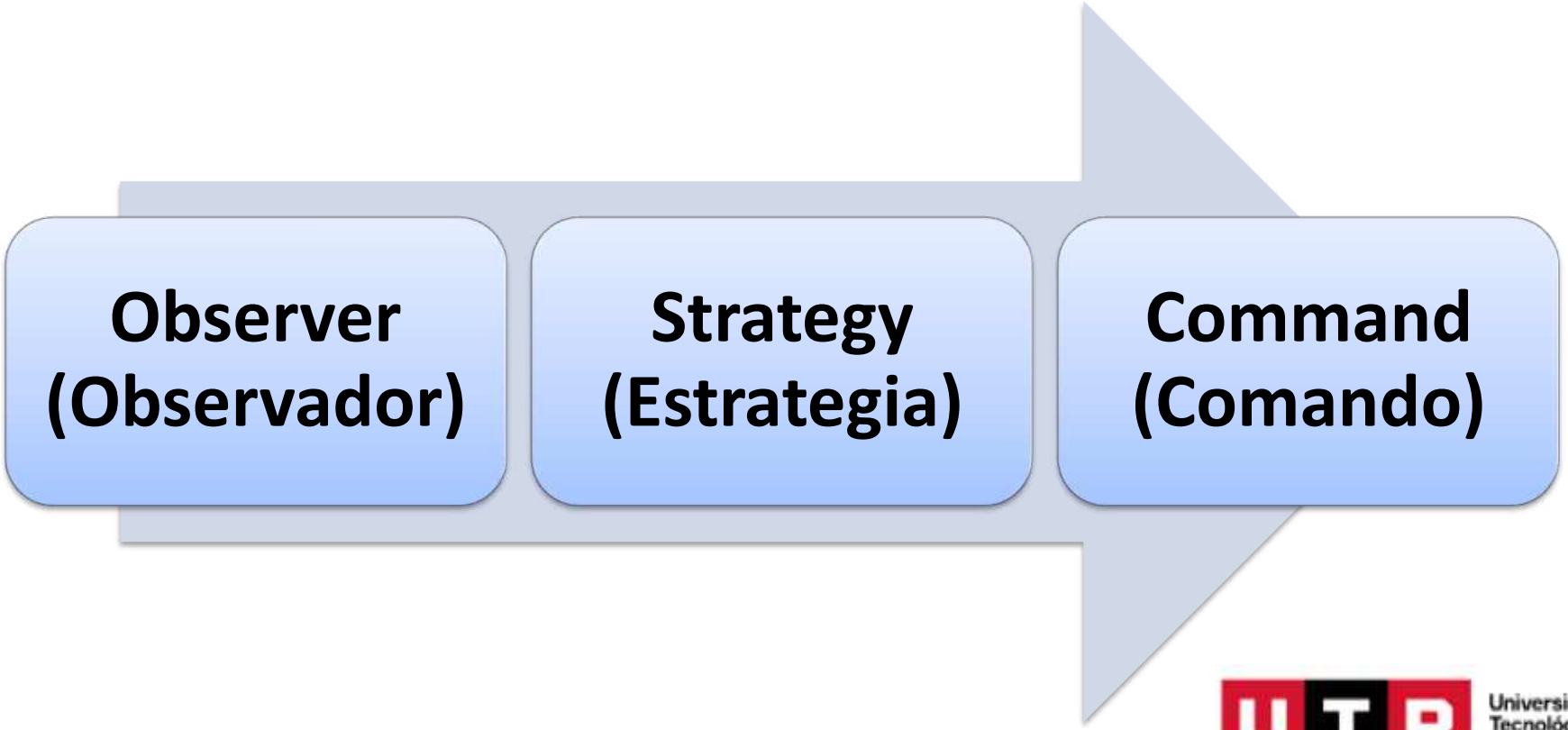
Tipos de Patrones GoF-Patrones Estructurales



Tipos de Patrones GoF-Patrones Estructurales

- **Adapter (Adaptador):** Este patrón permite que objetos con interfaces incompatibles trabajen juntos. Se utiliza cuando se necesita conectar dos sistemas que no pueden interactuar directamente debido a diferencias en sus interfaces.
- **Composite (Composición):** El patrón Composite se utiliza para componer objetos en estructuras de árbol para representar jerarquías de objetos. Es útil cuando se necesita tratar tanto objetos individuales como composiciones de objetos de manera uniforme.
- **Bridge:** El patrón desacopla una abstracción de su implementación, de modo que ambas pueden variar independientemente. Es especialmente útil cuando se necesita extender clases en varias dimensiones.

Tipos de Patrones GoF-Patrones Comportamiento

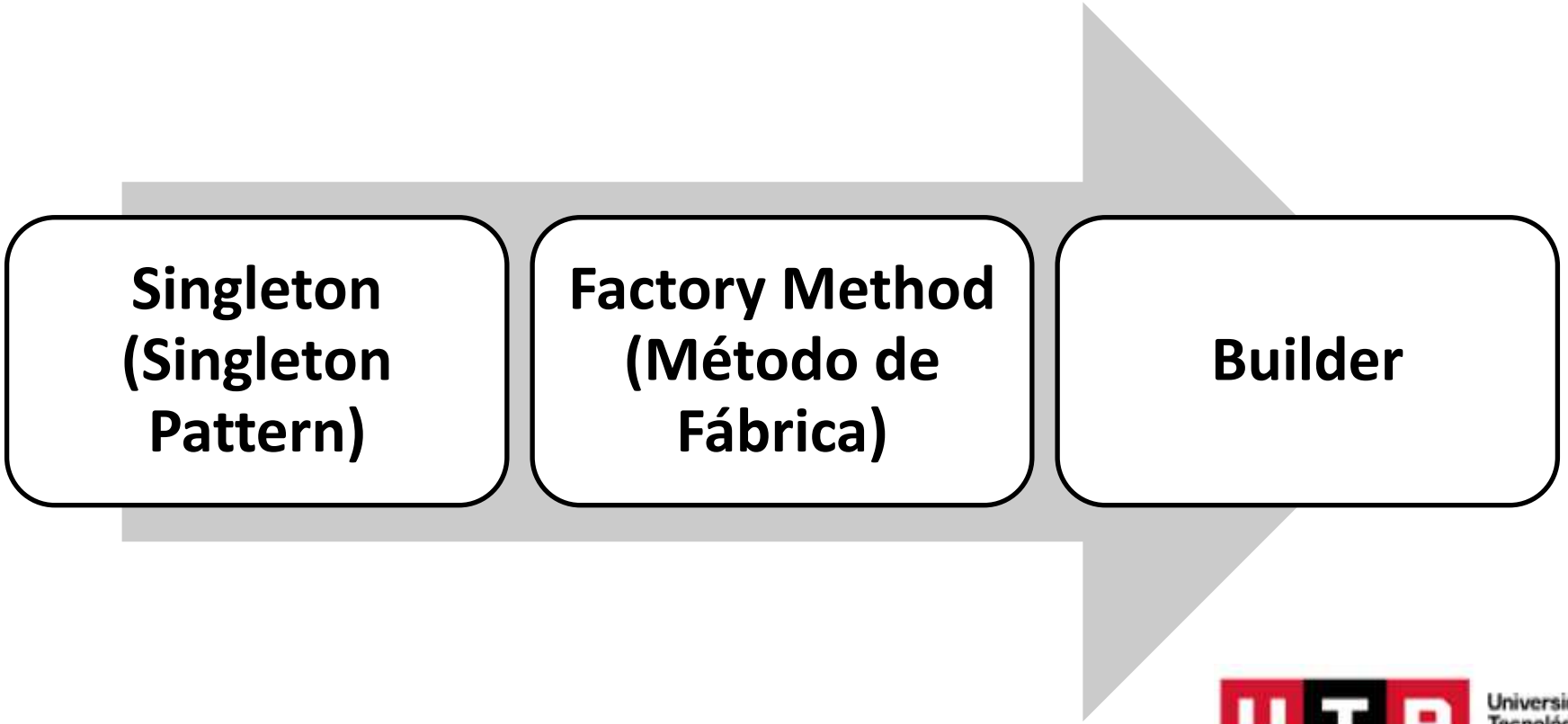


Tipos de Patrones GoF-Patrones Comportamiento

- **Observer (Observador):** Este patrón establece una relación uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos que dependen de él sean notificados y actualizados automáticamente.
- **Strategy (Estrategia):** El patrón Strategy define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Esto permite que el cliente elija el algoritmo que se utilizará en tiempo de ejecución.
- **Command (Comando):** El patrón Command encapsula una solicitud como un objeto, lo que permite parametrizar clientes con operaciones, encolar solicitudes y respaldar operaciones deshacer. Se utiliza para desacoplar a quien emite una solicitud de quien la procesa.

Detalle de cada uno de los Patrones GoF Creacionales

Patrones GoF-Patrones Creacionales



Patrón Singleton:

El patrón Singleton es uno de los patrones de diseño más conocidos y se utiliza para garantizar que una clase tenga solo una instancia y proporcionar un punto de acceso global a esa instancia.

Propósito: Asegurarse de que una clase tenga una única instancia y proporcionar un punto de acceso global a ella.

Uso común: Se utiliza en escenarios donde se necesita controlar el acceso a recursos compartidos, como gestores de configuración, conexiones a bases de datos o registros (logs).

Patrón Singleton:

https://www.youtube.com/watch?v=uB3FFZsdx3w

Universidad Tecnológica del Perú

Patrón Singleton:

```
public class Singleton {
    // Instancia privada y estática de la clase Singleton
    private static Singleton uniqueInstance;
    // Constructor privado para evitar la instanciación externa
    private Singleton() {}
    // Método público y estático para obtener la instancia
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

Cómo funciona:

- El constructor de la clase Singleton es privado. Esto garantiza que ninguna otra clase pueda instanciar un objeto Singleton directamente.
- La única forma de obtener una instancia de Singleton es a través del método getInstance().
- La primera vez que se llama a getInstance(), se crea una nueva instancia de Singleton. Las llamadas subsiguientes a getInstance()
- simplemente devolverán la instancia ya creada.

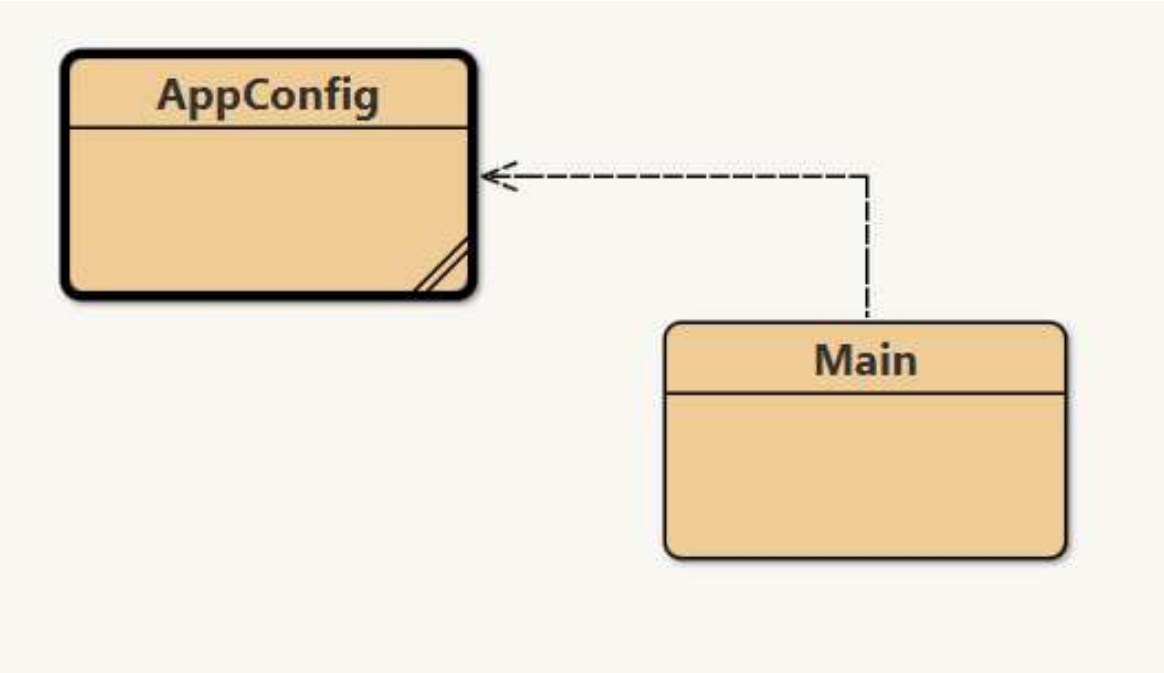
Patrón Singleton:

Nota:

- La implementación básica anterior no es segura para entornos multihilo. Si múltiples hilos intentan accedera ella al mismo tiempo, podrían crear múltiples instancias.
- Para hacerlo seguro en un entorno multihilo, se pueden utilizar mecanismos de sincronización.
- Es importante mencionar que el patrón Singleton debe usarse con precaución, ya que puede introducir un estado global en una aplicación y hacer que el código sea menos predecible. Es esencial evaluar si realmente se necesita un Singleton antes de implementarlo.

Patrón Singleton:

Implementar un Singleton para manejar la configuración de una aplicación.



```
public class AppConfig {
    private static AppConfig instance;
    private String configValue;

    private AppConfig() {
        // Cargar configuración desde archivo o base de datos
        configValue = "default_value";
    }

    public static AppConfig getInstance() {
        if (instance == null) {
            instance = new AppConfig();
        }
        return instance;
    }

    public String getConfigValue() {
        return configValue;
    }

    public void setConfigValue(String configValue) {
        this.configValue = configValue;
    }
}
```

Patrón Singleton:

Implementar un Singleton para manejar la configuración de una aplicación.

Patrón Singleton:

```
public class Main {
    public static void main(String[] args) {
        AppConfig config = AppConfig.getInstance();

        // Acceder y modificar configuración
        System.out.println("Valor de configuración: " + config.getConfigValue());
        config.setConfigValue("new_value");
        System.out.println("Valor de configuración actualizado: " + config.getConfigValue());

        // Otra instancia devuelve la misma configuración
        AppConfig anotherConfig = AppConfig.getInstance();
        System.out.println("Valor de configuración de otra instancia: " + anotherConfig.getConfigValue());
    }
}
```

Patrón Singleton:

Como funciona:

Atributos:

private static AppConfig instance;: Esta es la única instancia de la clase AppConfig que será compartida en toda la aplicación.

private String configValue;: Esta es una propiedad de configuración que se puede obtener y modificar.

Constructor privado:

```
private AppConfig() {
    // Load configuration from file or database
    configValue = "default_value";
}
```

El constructor es privado para prevenir la creación de instancias desde fuera de la clase. Esto asegura que la única forma de obtener una instancia de AppConfig es a través del método getInstance().

Patrón Singleton:

```
public static AppConfig getInstance() {
    if (instance == null) {
        instance = new AppConfig();
    }
    return instance;
}
```

Como funciona:

Método estático getInstance():

Este método retorna la instancia única de AppConfig. Si la instancia aún no ha sido creada (instance == null), se crea una nueva. En caso contrario, se retorna la instancia ya existente.

Patrón Singleton:

Métodos de acceso y modificación de configValue:

```
public String getConfigValue() {
    return configValue;
}

public void setConfigValue(String configValue) {
    this.configValue = configValue;
}
```

- **getConfigValue():** Retorna el valor actual de configValue.
- **setConfigValue(String configValue):** Permite establecer un nuevo valor para configValue.

Patrón Singleton: main

Obtención de la instancia única de AppConfig:

```
AppConfig config = AppConfig.getInstance();
```

Se obtiene la instancia única de AppConfig usando el método getInstance().

Acceso y modificación de la configuración:

```
System.out.println("Config value: " + config.getConfigValue());
config.setConfigValue("new_value");
System.out.println("Updated config value: " + config.getConfigValue());
```

- Se imprime el valor actual de configValue.
- Se cambia el valor de configValue a "new_value" y se imprime el nuevo valor.

Patrón Singleton:

Verificación del patrón Singleton:

```
AppConfig anotherConfig = AppConfig.getInstance();
System.out.println("Config value from another instance: " + anotherConfig.getConfigValue());
```

- Se obtiene otra referencia a la instancia única de AppConfig.
- Se verifica que anotherConfig tiene el mismo configValue que config, demostrando que ambas referencias apuntan a la misma instancia.

Resumen

El patrón Singleton garantiza que solo una instancia de AppConfig sea creada y compartida en toda la aplicación. Esto es útil cuando se necesita un único punto de acceso para cierta información o configuración a lo largo de la vida de la aplicación.

Patrón Factory:

El patrón Factory es un patrón creacional que proporciona una interfaz para crear objetos en una superclase, pero permite a las subclases alterar el tipo de objetos que se crearán.

Propósito: Definir una interfaz para crear un objeto, pero permitir que las subclases decidan qué clase instanciar. El Factory Method permite a una clase delegar la responsabilidad de instanciar sus objetos a sus subclases.

Uso común: Se utiliza cuando hay una superclase con múltiples subclases y, en función de la entrada, se tiene que devolver una clase de una de las subclases.

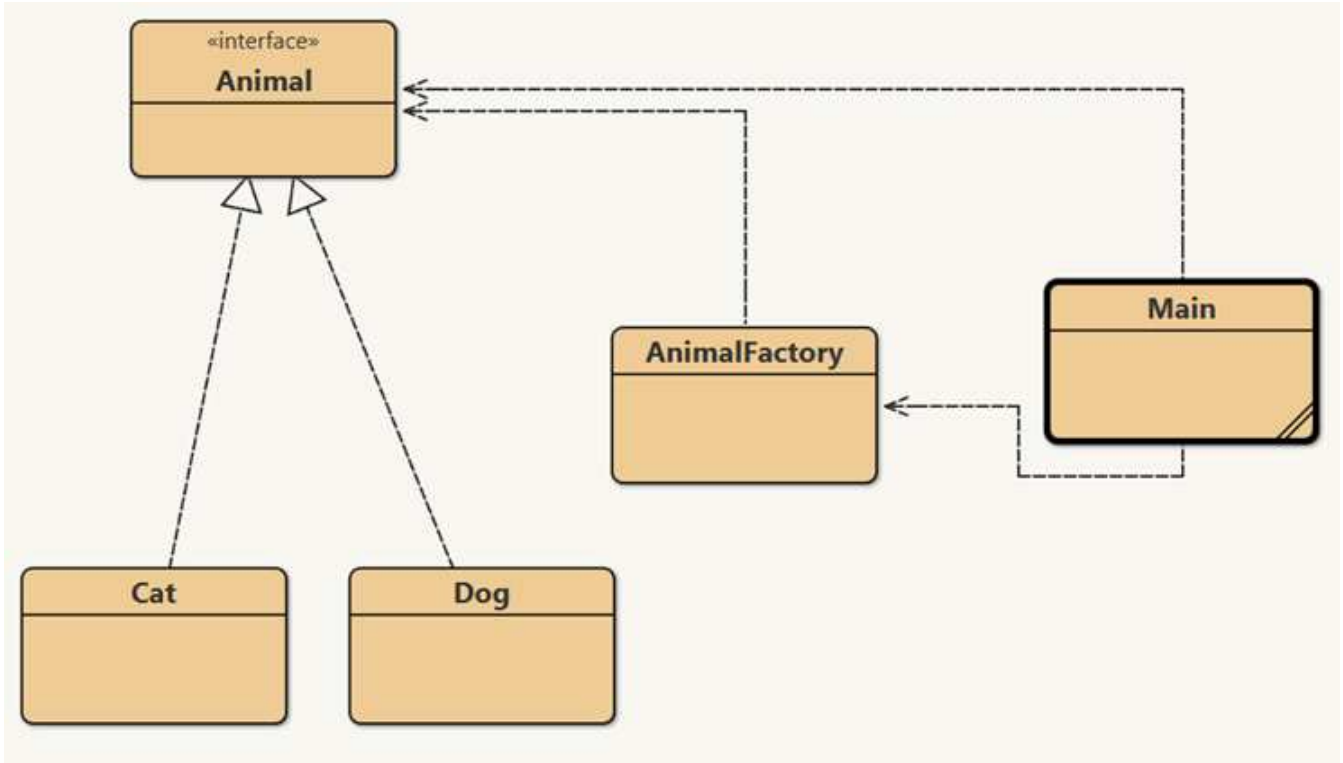
Patrón Factory:

https://www.youtube.com/watch?v=v-NdSDNVb4U&list=PL1YXwF4Lvrn1c9yrsoG3i0aKuYmIf0FjU&index=2

Universidad Tecnológica del Perú

Patrón Factory:

Supongamos que tenemos una interfaz Animal y varias clases que implementan esta interfaz, como Dog y Cat.



Patrón Factory:

```
// Interfaz Animal
public interface Animal {
    void speak();
}
```

```
// Clase Cat que implementa Animal
public class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("Meow!");
    }
}
```

```
public class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("Woof!");
    }
}
```

Implementación básica en Java:
Supongamos que tenemos una interfaz Animal y varias clases que implementan esta interfaz, como Dog y Cat.

Patrón Factory:

```
public class AnimalFactory {
    // Método Factory para crear instancias de Animal
    public Animal getAnimal(String type) {
        if ("Dog".equalsIgnoreCase(type)) {
            return new Dog();
        } else if ("Cat".equalsIgnoreCase(type)) {
            return new Cat();
        }
        return null;
    }
}
```

Ahora, implementaremos el Factory para crear instancias de Dog o Cat basadas en la entrada:

Patrón Factory:

```
// Clase Main para probar la AnimalFactory
public class Main {
    public static void main(String[] args) {
        AnimalFactory factory = new AnimalFactory();
        Animal dog = factory.getAnimal("Dog");
        dog.speak();  // Salida: Woof!
        Animal cat = factory.getAnimal("Cat");
        cat.speak();  // Salida: Meow!
    }
}
```

Patrón Factory:

El patrón Factory es útil para aislar la lógica de creación de objetos y promover la cohesión y la modularidad en el código.

Cómo funciona:

- La interfaz Animal define un contrato que todas las subclases deben seguir.
- Las clases Dog y Cat implementan la interfaz Animal y proporcionan sus propias implementaciones del método speak().
- AnimalFactory es la clase Factory que crea y devuelve instancias de Dog o Cat basadas en la entrada proporcionada.
- En el método main, se utiliza el Factory para crear instancias de Dog y Cat y se invoca el método speak() en cada instancia.

Patrón Builder:

El patrón Builder es un patrón creacional que permite construir objetos complejos paso a paso. Se utiliza para construir un objeto compuesto asegurando que el proceso de construcción sea independiente de las partes que componen el objeto y de su representación.

Propósito: Separar la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción pueda crear diferentes representaciones.

Uso común: Se utiliza cuando un objeto necesita ser creado con muchas opciones posibles, y muchas de estas opciones pueden ser opcionales.

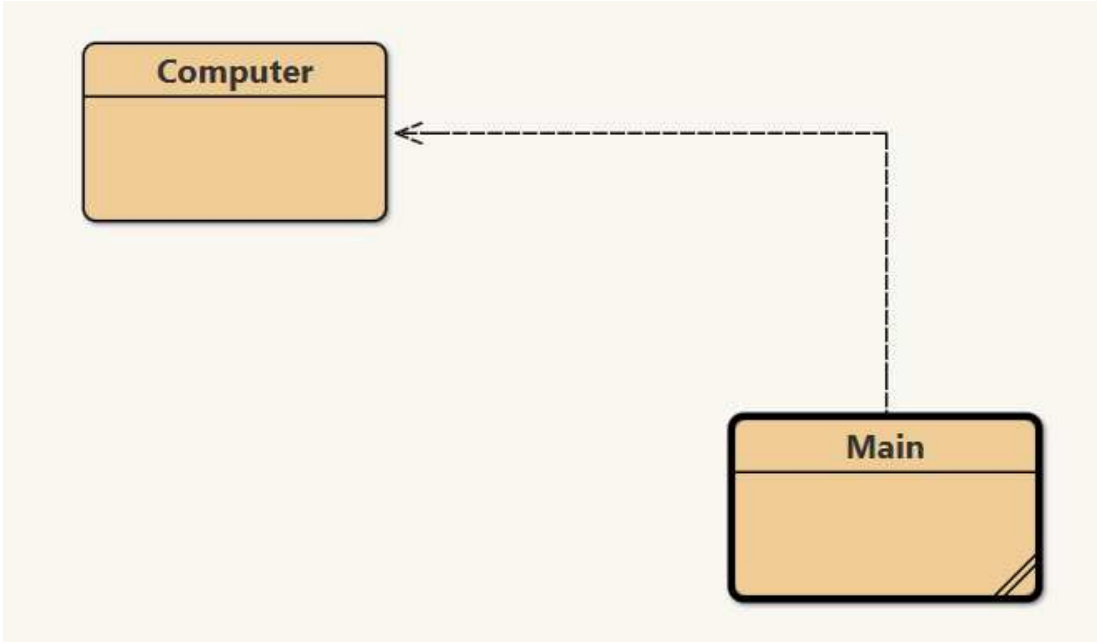
Patrón Builder:

https://www.youtube.com/watch?v=MRoZCgtQX1E&list=PL1YXwF4Lvrn1c9yrsoG3i0aKuYmIf0FjU&index=5

Universidad Tecnológica del Perú

Patrón Builder:

Supongamos que queremos construir un objeto Computer que tiene varios componentes como CPU, RAM, disco duro, tarjeta gráfica, etc.



Patrón Builder:

Implementación básica en Java: Supongamos que queremos construir un objeto Computer que tiene varios componentes como CPU, RAM, disco duro, tarjeta gráfica, etc.

```
public class Computer {
    private String CPU;
    private String RAM;
    private String hardDrive;
    private String graphicsCard;

    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.hardDrive = builder.hardDrive;
        this.graphicsCard = builder.graphicsCard;
    }

    public void printDetails() {
        System.out.println("CPU: " + CPU);
        System.out.println("RAM: " + RAM);
        System.out.println("Hard Drive: " + hardDrive);
        System.out.println("Graphics Card: " +
graphicsCard);
        System.out.println();
    }
```

```
public static class Builder {
        private String CPU;
        private String RAM;
        private String hardDrive;
        private String graphicsCard;

        public Builder(String CPU, String RAM) {
            this.CPU = CPU;
            this.RAM = RAM;
        }

        public Builder setHardDrive(String hardDrive) {
            this.hardDrive = hardDrive;
            return this;
        }

        public Builder setGraphicsCard(String graphicsCard) {
            this.graphicsCard = graphicsCard;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}
```

Patrón Builder

Patrón Builder:

Cómo funciona:

- El objeto Computer tiene varios componentes, algunos de los cuales son obligatorios (CPU, RAM) y otros opcionales (hardDrive, graphicsCard).
- **La clase Builder** interna permite configurar las opciones deseadas para el objeto Computer.
- El método build() de la clase Builder crea y devuelve una instancia de Computer con las opciones configuradas.
- En el método main, se utiliza el Builder para crear dos diferentes configuraciones de Computer.

```
public class Main {
    public static void main(String[] args) {
        Computer gamingComputer = new
Computer.Builder("Intel i9", "32GB RAM")
                .setGraphicsCard("NVIDIA RTX 3090")
                .setHardDrive("2TB SSD")
                .build();

        Computer officeComputer = new
Computer.Builder("Intel i5", "16GB RAM")
                .setHardDrive("1TB HDD")
                .build();

        gamingComputer.printDetails();
        officeComputer.printDetails();
    }
}
```

Patrón Builder:

El patrón Builder es útil para garantizar que un objeto se construya correctamente, especialmente cuando tiene múltiples opciones de configuración.
También mejora la legibilidad y organización del código al aislar la lógica de construcción.

Practica

Implementa los ejercicios vistos.

Conclusiones:

- Cada uno de estos patrones GoF aborda problemas de diseño específicos y proporciona soluciones probadas.
- Los desarrolladores pueden seleccionar y aplicar estos patrones según las necesidades de sus proyectos, lo que contribuye a un diseño de software más flexible, mantenible y escalable.

Universidad Tecnológica del Perú

Conclusiones:

- Al comprender y aplicar estos patrones, los desarrolladores pueden tomar decisiones informadas sobre cómo abordar los problemas de diseño en sus proyectos y mejorar la calidad de su código.
- Estos patrones han resistido la prueba del tiempo y se han convertido en herramientas esenciales en el kit de herramientas de cualquier desarrollador de software.

Universidad Tecnológica del Perú

Cierre

¿Qué hemos aprendido hoy?

Universidad Tecnológica del Perú

Bibliografía

- MORENO PÉREZ, J. “Programación orientada a objetos”. RA-MA Editorial. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=31933
- Vélez Serrano, José. “Diseñar y programar, todo es empezar: una introducción a la Programación Orientada a Objetos usando UML y Java”. Dykinson. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=36368

Universidad Tecnológica del Perú