Trabajo 1 – Patrones y diseño de software

Objetivo: Demostrar la aplicación de principios SOLID y patrones creaciones.

**Punto 1  
  
Parte A:  
.** Definición del problema que se desean abordar (25%)

a) Describir uno o varios problemas donde se puedan implementar tres patrones creacionales distintos, justificando porqué la aplicación del patrón da solución a los problemas planteados.

Los problemas pueden estar relacionados o ser independientes entre sí, la temática es libre. (Se esperan en total tres patrones creacionales justificados).

**Solución Parte A:**

1. **Object pool:**

***Problema:***

En el desarrollo de video juegos, crear y destruir objetos constantemente puede consumir muchos recursos, así mismo se puede presentar un problema cuando se instancia y se destruye los objetos, debido a que la memoria se comienza a fragmentar lo que inevitablemente va a afectar el rendimiento general del juego.

***Solución***: Usar object pool

Con el patrón **object pool** se busca reciclar los objetos, es decir se crea una vez los objetos, se utilizan y luego se liberan los objetos, para ser nuevamente utilizados desde el pool cuando sean necesarios.

En este caso el patrón es útil, puesto que no se está creando y destruyendo en memoria con cada objeto nuevo, si no que se reutilizara los objetos disponibles preexistentes, reduciendo la instanciación de los objetos y mejorando su proceso. Así mismo se pueden establecer limites en la cantidad de objetos a crear, lo que evitara desbordamiento de memoria.

1. Patron Factory Method

***Problema:***

La complejidad en la creación y gestión de múltiples tipos de dispositivos electrónicos en una tienda, cada uno con sus propias características, pero que comparten atributos comunes. Sin el uso del patron Factory, la lógica para instanciar cada dispositivo estaría dispersa en el código. Con sentencias **if** que deciden que clase crear según el tipo de producto. Generando alto acoplamiento, falta de escalabilidad y problemas en la reutilización y extensión.

***Solución***: Usar ***Factory Method***

Implementaremos el patrón **Factory Method** para, la gestión de la tienda de productos electrónicos que vende dispositivos como teléfonos móviles, computadoras y tabletas. Cada dispositivo tiene características específicas, pero comparten atributos comunes como nombre, precio y la capacidad de mostrar sus especificaciones. El patrón Factory encapsula la lógica de creación en una una fábrica que decide que objeto se va a instanciar. Esto mejora la extensión, la mantenibilidad y la organización del código de una manera sencilla. La adición de un nuevo dispositivo se traduce en la creación de una nueva subclase y la actualización de la fábrica, sin alterar el código principal de la aplicación.

1. **Patrón Builder**

***Problema:***

Imagina que administras una fábrica de dispositivos electrónicos donde cada cliente puede encargar un teléfono inteligente o una computadora con diferentes configuraciones. Por ejemplo, algunos clientes quieren 4 GB de RAM con un procesador básico y una cámara de calidad media, mientras que otros desean 16 GB de RAM con un procesador de última generación y cámara de alta resolución. Además, en el caso de las computadoras, hay quienes requieren un teclado mecánico o incluso distintas capacidades de almacenamiento.

La dificultad surge cuando intentas crear todos estos dispositivos con múltiples parámetros que varían de cliente en cliente. Usar un único constructor con muchos argumentos (RAM, procesador, almacenamiento, cámara, teclado, etc.) se vuelve inmanejable y poco claro, sin mencionar la gran posibilidad de cometer errores al instanciar un dispositivo con tantos parámetros opcionales o combinaciones distintas.

***Solución***: Usar ***Builder***

Con builder, se separa la lógica para armar un dispositivo, ya sea una computadora o un teléfono. Así mismo es fácil cambiar la configuración de cada dispositivo, sin modificar el código principal ni crear múltiples constructores. También la existencia del DeviceDirector, hace que, en un futuro al añadir nuevos componentes, se debería actualizar el builder y el director sin necesidad de afectar el resto del sistema.

**Parte B. Diagrama UML de Clases (25%)**

Para cada uno de los problemas definidos en el punto 1, elaborar el diagrama UML donde se evidencie la estructura del patrón de diseño, explicando la correspondencia con los participantes.

Es importante que los nombres de las clases se adapten al dominio, es decir, no dejar nombres de clases genéricas o igual a como se presentan en el material de estudio.

Se esperan tres diagramas diferentes o un único diagrama que implemente los tres patrones seleccionados.

**Solución parte B:**

1. **Object pool**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

1. **Patrón Factory Method**

**Interfaz de usuario gráfica, Diagrama, Aplicación

El contenido generado por IA puede ser incorrecto.**

1. **Patron Builder:**

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

**Parte C. Implementación de los patrones diseño (25%).**

Implementar en el lenguaje de su elección las clases y los objetos que hacen parte de los tres patrones creacionales, evidenciando un correcto uso de estos y siguiendo principios SOLID.

No es necesario implementar funcionalidades, es suficiente con mostrar mensajes por consola donde se evidencie el funcionamiento del patrón.

1. [**Patron Object pool**](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesCreacionales/tree/main/src/main/java/org/example/object_pool)**<Link>**

La clase principal es objectPoolExample.java.

Se deja claro que el patron ha sido usado invocando sus métodos desde la clase main, solo para efectos de visualizar la correcta liberación de memoria y se entiende que el uso completo de la clase Bullet y su ciclo de vida, debe ser controlado directamente desde BulletPool

1. [**Patrón Factory Method**](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesCreacionales/tree/main/src/main/java/org/example/factory_method)**<Link>**

La clase principal de entrada es main.py.

1. [**Patron Builder<Link>:**](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesCreacionales/tree/main/src/main/java/org/example/builder)

La clase principal es BuilderCliente.java

**Punto 2 (25%).**

Se requiere modelar un sistema para construir documentos en formato PDF, asegurando la flexibilidad en la creación de los documentos entre lo que se incluye la generación de documento a partir de plantillas.

La aplicación deberá permitir al usuario seleccionar el tipo de formato de origen con el que desea trabajar (por ejemplo, XLSX, DOCX, o XML) y posteriormente generar documentos complejos de manera programática, incluyendo adición de múltiples columnas, tablas, títulos, estilos, entre otros.

**Requisitos:**

1. El sistema debe permitir seleccionar el tipo de documento con el que se desea trabajar. El sistema debe funcionar independiente del formato.
2. Los documentos deben ser construidos de manera flexible y permitir agregar columnas, tablas, enunciados, estilos, entre otros, en tiempo de ejecución.
3. La solución debe ser fácilmente extensible para soportar nuevos formatos en el futuro sin modificar el código existente.
4. El código debe estar organizado de forma que el cliente de la aplicación no necesite conocer los detalles de lectura y escritura de los formatos, ni de la generación del código fuente.

Entregables:

* Diagrama de Clases de UML de la solución en el que se evidencie el uso de al menos dos patrones.

Interfaz de usuario gráfica, Aplicación, Teams

El contenido generado por IA puede ser incorrecto.

* Explicación de cómo se abordaría este problema, justificando la elección del diseño.

***Explicación:***

Teniendo en cuenta los requisitos iniciales planteados en el problema, se toma la decisión técnica de implementar dos patrones en el diseño de la solución. Las razones para la implementación de estos se dejan a continuación plasmada:

El **Factory Method** es una elección natural para este tipo de problema, ya que permite que la aplicación sea flexible respecto a los formatos de entrada (XLSX, DOCX, XML, etc.). sin tener que modificar considerablemente el código cuando se agregan nuevos formatos. Así mismo, el cliente no tiene que preocuparse por los detalles internos de lectura y escritura de cada formato.

El **Builder** es ideal para construir objetos complejos (con tablas, columnas, títulos…) y de manera flexible, sin exponer la complejidad de su construcción al cliente. En este caso, se utiliza para construir el documento PDF de manera modular y ordenada. Sin que el cliente deba manejar directamente la lógica de construccion paso a paso.

Se cumple el principio de **open/closed** de SOLID, ya que, para añadir nuevos formatos o nuevas formas de construir un documento, se extiende el sistema con nuevas clases sin alterar las existes. Así mismo se cumple con principio de single responsability.

El diseño adoptado combina patrones de diseño orientados a objetos que favorecen la **flexibilidad**, la **extensibilidad, el encapsulamiento de la lógica interna** y la **separación de responsabilidades**. Los patrones **Factory Method** y **Builder** permiten que el sistema sea fácilmente adaptable a futuros cambios, como la adición de nuevos formatos o la personalización de la construcción de documentos, sin afectar el código existente. La solución está diseñada para ser simple de entender y fácil de mantener a largo plazo, cumpliendo con los requisitos planteados en el problema.

* Código fuente de la aplicación donde se evidencie la interacción de los patrones, no es necesario implementar las funcionalidades.

[Código fuente punto 2.](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesCreacionales/tree/main/src/main/java/org/example/punto2)<Link>

**Plazo de entrega:**

Sábado 22 de marzo.

El trabajo deberá ser enviado a través de Teams, pueden usar repositorios como github, gitlab o Azure Repos para el código, en el documento deberá aparecer el enlace al repositorio, el repositorio deberá ser público. El trabajo será presentado en horario de clase.

El documento debe contener el nombre de los integrantes del grupo. Máximo 3 personas por grupo, los grupos son de conformación libre.