**ESCUELA SUPERIOR POLITECNICA DEL LITORAL**

**Diseño de Software**

**Taller #4: Patrones**

**Integrantes:**

Andrés Layedra

Xavier Camacho

Carlos Ronquillo

Luis Rivera

Índice de Contenidos

[Sección A 3](#_Toc181803194)

[1. Factory Method 3](#_Toc181803195)

[2. Decorator 4](#_Toc181803196)

[3. Adapter 5](#_Toc181803197)

[Sección B 6](#_Toc181803198)

# Sección A

## Factory Method

**Motivación de uso:**

El patrón Factory Method es adecuado para este sistema debido a que permite crear objetos de diferentes tipos (en este caso, los reportes en diferentes formatos) sin especificar la clase exacta que se debe instanciar. En este escenario, cada tipo de reporte (PDF, Excel, Word) comparte una estructura común, pero varía en la forma en que se genera y se visualiza. El uso de un método de fábrica permite centralizar la lógica de creación de los objetos, garantizando que, a pesar de las diferencias en los formatos de los reportes, el proceso de generación sea consistente y escalable.

**Consecuencias de la decisión:**

Entre las ventajas de usar el patrón Factory Method se encuentran la capacidad de agregar nuevos tipos de reportes en el futuro sin afectar el código que utiliza los reportes existentes, lo que facilita la extensibilidad del sistema. Además, favorece el principio abierto/cerrado de SOLID, ya que el sistema está abierto a la extensión (por la adición de nuevos tipos de reporte) pero cerrado a la modificación (no es necesario cambiar el código existente para incorporar nuevos tipos de reportes). Por otro lado, las desventajas son el aumento en la complejidad del sistema debido a la necesidad de crear una jerarquía de clases para los diferentes tipos de reportes, así como la introducción de más clases y métodos que pueden hacer que el código sea más difícil de entender al principio.

**Relación con principios SOLID:**

El patrón Factory Method está alineado con varios principios de SOLID, principalmente con el principio de abierto/cerrado, como se mencionó anteriormente. Asimismo, el patrón promueve el Single Responsibility Principle al centralizar la creación de los objetos en un único método, lo que mejora la cohesión del sistema. También facilita el principio de inversión de dependencias, ya que permite que el código cliente dependa de abstracciones (interfaces o clases base) en lugar de implementaciones concretas, lo que mejora la flexibilidad y reduce el acoplamiento.

**Asunciones realizadas:**

Se asume que cada tipo de reporte tiene un comportamiento diferenciado, pero sigue un proceso común de generación que puede ser encapsulado de manera flexible.

## Decorator

**Motivación de uso:**

El patrón Decorator es ideal para este caso porque permite añadir funcionalidades adicionales (como personalización de estilo, fuentes, colores, etc.) a los objetos de manera dinámica sin alterar su estructura interna. En este sistema, dicho patrón permite agregar de forma flexible las opciones de personalización a los informes de acuerdo con los requerimientos específicos de cada usuario, sin necesidad de modificar las clases base de los reportes. Aquello es crucial, ya que diferentes informes pueden tener diferentes niveles de personalización, y el uso de decoradores permite construir estos informes de manera incremental, adaptándose a las preferencias del usuario sin complicar la lógica central de generación de los reportes.

**Consecuencias de la decisión:**

La ventaja de implementar el patrón Decorator es una gran flexibilidad para modificar la apariencia y características de los informes de forma modular y extendida, permitiendo la adición de funcionalidades sin modificar las clases base de los informes. Además de tal, otra ventaja es que se facilita el Single Responsibility Principle al momento de delegar las modificaciones visuales en objetos decoradores específicos, manteniendo la generación de los informes organizada y separada de las personalizaciones. Lamentablemente, posee desventajas, entres las que se incluyen la posible complejidad añadida debido a la creación de múltiples clases decoradoras, lo que podría generar una jerarquía de objetos más difícil de entender o gestionar.

**Relación con principios SOLID:**

Este patrón se relaciona estrechamente con varios principios SOLID. En primer lugar, apoya el principio abierto/cerrado, ya que las funcionalidades adicionales pueden añadirse a los objetos sin modificar las clases existentes. También promueve el Single Responsibility Principle, ya que cada decorador tiene una única responsabilidad (modificar un aspecto específico del informe) y se puede combinar con otros decoradores para crear configuraciones complejas de manera modular. Además, favorece el principio de inversión de dependencias al permitir que el sistema dependa de abstracciones y no de implementaciones concretas, lo que facilita la extensión del sistema sin acoplarse a implementaciones específicas de informes.

**Asunciones realizadas:**

Se asume que las personalizaciones de los informes no son fijas y pueden variar significativamente entre diferentes informes y entre diferentes usuarios.

## Adapter

**Motivación de uso:**

El patrón Adapter es la solución ideal para integrar diferentes sistemas de notificación (como correo electrónico, WhatsApp y Telegram) que tienen interfaces incompatibles. El propósito del patrón es proporcionar una interfaz común para interactuar con todos estos servicios, permitiendo que el sistema de generación de informes pueda enviar notificaciones de manera transparente, independientemente del medio seleccionado. Cada servicio externo tiene su propia implementación, pero al utilizar un adaptador para cada uno, se puede hacer que todos ellos "encajen" en una interfaz unificada que el sistema principal pueda usar sin necesidad de preocuparse por las diferencias de implementación entre los diversos servicios.

**Consecuencias de la decisión:**

Como ventajas de usar el patrón Adapter se incluye la capacidad de integrar fácilmente múltiples servicios externos sin tener que modificar el sistema principal ni reescribir las lógicas de envío de notificaciones. Los adaptadores permiten mantener la flexibilidad para agregar más servicios de notificación en el futuro sin afectar el código existente. Asimismo, tal patrón permite un bajo acoplamiento entre el sistema central y las bibliotecas o servicios externos, lo que mejora la mantenibilidad y escalabilidad del sistema. Por otra parte, entre las desventajas se podrían considerar la sobrecarga de crear y gestionar múltiples clases adaptadoras para cada servicio externo, lo que puede aumentar la complejidad del código.

**Relación con principios SOLID:**

Este patrón Adapter está alineado con el principio abierto/cerrado, ya que permite que el sistema sea extendido para soportar nuevos servicios de notificación sin necesidad de modificar el código existente. También facilita el principio de inversión de dependencias, ya que el sistema central depende de interfaces comunes (siendo estas, abstracciones) para interactuar con los servicios de notificación, en lugar de depender de implementaciones concretas. De esta manera, se mantiene el código más flexible y menos acoplado. Además, el patrón promueve el Single Responsibility Principle, ya que cada adaptador se encarga únicamente de convertir las interfaces externas a una interfaz común que el sistema puede entender.

**Asunciones realizadas:**

Se presupone que los servicios de notificación (correo electrónico, WhatsApp, Telegram, etc.) no comparten una forma común de implementación, lo que hace necesario el uso de adaptadores para transformar sus interfaces a un formato común que el sistema pueda utilizar.

# Sección B