Trabajo 3 – Patrones y diseño de software

Xavier De Jesús Lozano Figueroa.

Miguel Ángel Blanco López.

Marlon David Peñuela Pardo.

Facultad de ingenierías, Instituto Tecnológico Metropolitano (ITM).

Demostrar la aplicación de principios SOLID y patrones de comportamiento.

**Punto 1**

**Parte A:**

. Definición del problema que se desean abordar (33%)

a) Definir tres problemas donde se puedan implementar patrones de comportamiento distintos, justificando porqué la aplicación del patrón da solución al problema planteado.

Los problemas pueden estar relacionados o ser independientes entre sí, la temática es libre.

1. **Patrón 1: Patrón visitor**

**Problema cotidiano y solución con el patrón visitante**

**Problema:**

Un programa de gestión de documentos maneja diferentes tipos de archivos (PDF, Word, Excel, imágenes) y necesita realizar múltiples operaciones con ellos (imprimir, exportar, validar, comprimir) sin modificar las clases de los archivos cuando se añaden nuevas operaciones.

**Solución:**

Se implementa el patrón Visitor donde:

* Los diferentes tipos de archivo son los "Elementos" (clases concretas que no cambiarán)
* Las operaciones a realizar son los "Visitantes" (imprimir, exportar, validar)
* Cada visitante implementa su lógica específica para cada tipo de archivo
* Se pueden añadir nuevas operaciones (como cifrar o analizar) creando nuevos visitantes sin modificar las clases de los archivos
* El sistema puede aplicar cualquier operación a una estructura de archivos heterogénea sin usar condicionales por cada tipo

Esta implementación separa claramente las operaciones de los objetos sobre los que actúan, permitiendo añadir nuevas funcionalidades sin modificar las clases existentes de los archivos.

**Problemas del enfoque estándar:**

* Problemas de extensibilidad: Si necesitamos añadir una nueva operación (como cifrar), debemos modificar TODAS las clases existentes (PDF, Excel, Word, etc.) para añadir el nuevo método.
* Violación del principio Open/Closed: Las clases no están "cerradas para modificación" ya que cada nueva funcionalidad requiere cambiar todas las clases existentes.
* Código disperso: La lógica relacionada con una operación (como imprimir) está dispersa en múltiples clases, lo que dificulta su mantenimiento.
* Cohesión baja: Cada clase contiene código para muchas operaciones diferentes, cuando lo ideal sería agrupar el código relacionado.
* Difícil reutilización: Es complicado reutilizar la lógica de una operación específica en otro contexto.

1. **Patrón 2: Template Method**

Un sistema GPS necesita calcular el tiempo de viaje de acuerdo a la distancia que comprende la ruta y teniendo en cuenta el contexto y restricciones propias de cada medio de transporte. Para solucionar esto podemos usar el patrón template method ya que nos facilita modificar pequeñas partes del algoritmo de cálculo de la distancia sin necesidad de involucrar ciclos de control que pueden agregar más complejidad de la necesaria al código.

1. **Patrón 3: Patrón Observer**

Necesitamos implementar un sistema de notificaciones para una aplicación donde diferentes tipos de suscriptores (por correo electrónico, SMS y notificaciones push) deben recibir actualizaciones cuando ocurren eventos en el sistema.

**Ventajas:**

* Establece relaciones entre objetos en tiempo de ejecución
* Proporciona bajo acoplamiento entre Subject y Observers
* Soporta comunicación broadcast
* Facilita la adición de nuevos observadores sin cambiar el código existente

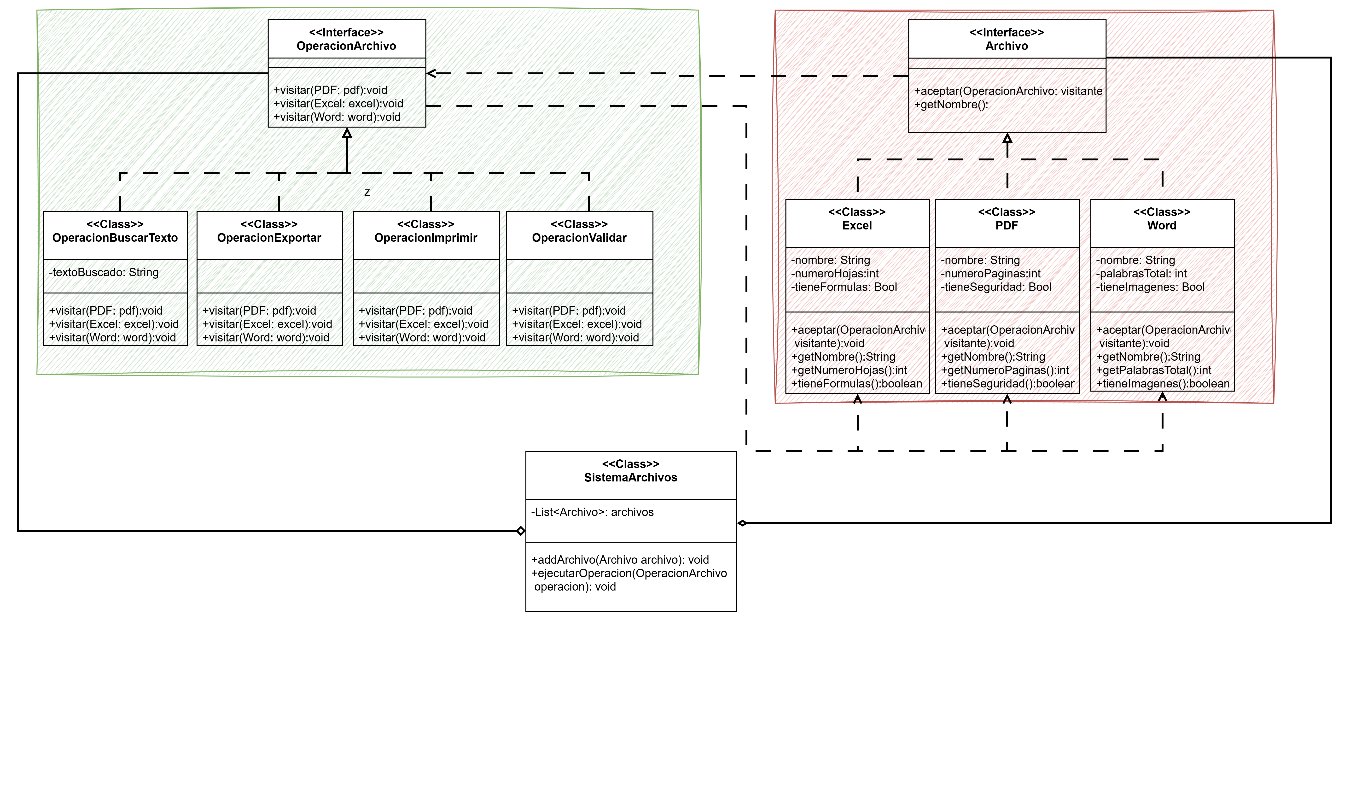
**Parte B. Diagrama UML de Clases (33%):**

Para cada uno de los problemas definidos en la parte A, 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 dos patrones seleccionados.

1. **Patrón 1:**



**Estructura del patrón**

El patrón Visitor consta de cuatro elementos principales:

* **Interfaz Elemento (Archivo)**: Define la operación aceptar() que permite a los visitantes procesar el elemento.
* **Elementos Concretos (PDF, Excel, Word)**: Implementan la interfaz Archivo y contienen los datos específicos de cada tipo de documento.
* **Interfaz Visitante (OperacionArchivo)**: Define métodos visitar() específicos para cada tipo de elemento.
* **Visitantes Concretos (OperacionImprimir, OperacionExportar, etc.)**: Implementan la interfaz Visitante con operaciones específicas para cada tipo de elemento.

**Beneficios clave demostrados**

* **Separación de responsabilidades**: Las clases de archivos solo contienen datos y la lógica de aceptar visitantes, mientras que las operaciones están en clases separadas.
* **Extensibilidad flexible**: Podemos añadir nuevas operaciones (como OperacionBuscarTexto) sin modificar las clases de archivos existentes.
* **Código cohesivo**: Toda la lógica relacionada con una operación está en una sola clase.

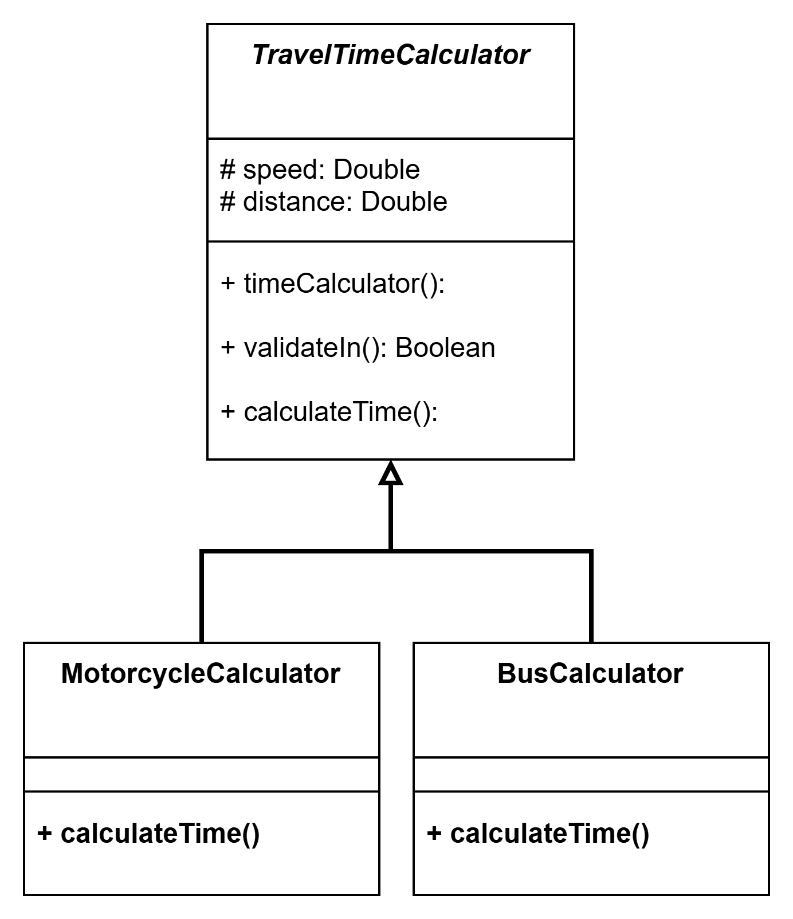
**¿Por qué es mejor que la solución "normal"?**

En un enfoque tradicional, tendríamos que añadir métodos como imprimir(), exportar(), validar() a cada clase de archivo. Esto presenta varios problemas:

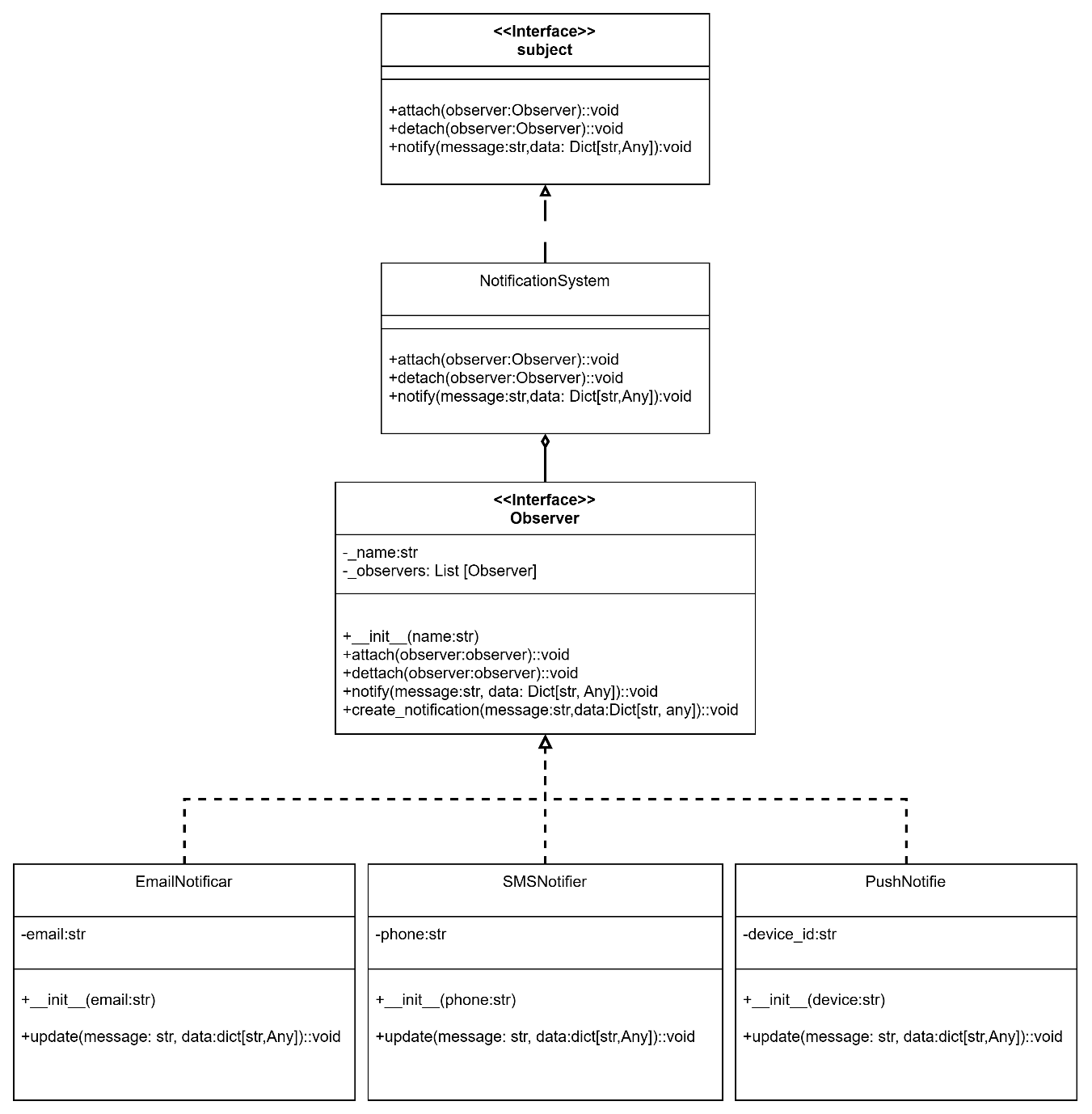
* Cada vez que añadimos una nueva operación, debemos modificar todas las clases de archivos.
* El código relacionado con cada operación queda disperso entre varias clases.
* Las clases de archivos acaban conteniendo mucha lógica no relacionada con su propósito principal.

Con el patrón Visitor, podemos añadir nuevas operaciones simplemente creando nuevas clases de visitantes, manteniendo las clases de archivos intactas.

1. **Patrón 2: Template**



1. **Patrón 3: Observer**



**Estructura del patrón.**

* La interfaz Observer define el método update() que todas las clases observadoras deben implementar.
* La interfaz Subject define los métodos para gestionar observadores y notificarles.
* NotificationSystem implementa la interfaz Subject y mantiene una colección de observadores.
* Las clases EmailNotifier, SMSNotifier y PushNotifier implementan la interfaz Observer y definen comportamientos específicos en respuesta a las notificaciones.

**Parte C. Implementación de los patrones diseño (34%).**

Implementar en el lenguaje de su elección las clases y los objetos que hacen parte de los tres patrones estructurales, 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. **Patrón 1: <**[Link](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesDeComportamiento/tree/main/src/main/java/org/example/visitor)**>**
2. **Patrón 2: <**[Link](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesDeComportamiento/tree/main/src/main/java/org/example/template_method)**>**
3. **Patrón 3: <**Link**> ------>** Ejecutar en el link python3 \_\_main\_\_.py
4. **Diagramas: <**[Link](https://app.diagrams.net/#G1GKdxyZUuqW_Va2EgfhkSQe9YcvUZtnlS)**><**[Link2](https://github.com/Especializacion-Ingenieria-Software/PrincipiosSolidPatronesDeComportamiento/blob/main/PatronesComportamentales.drawio)**>**

**Plazo de entrega:**

10 de mayo a las 08:00am.

El trabajo deberá ser cargado a través Teams, pueden usar repositorios como github, gitlab o Azure Repos para el código fuente, en el documento deberá aparecer el enlace al repositorio.  El trabajo será expuesto en el horario de clase. El documento debe contener el nombre de los integrantes del grupo. Máximo 4 personas por grupo, los grupos son de conformación libre.