**Cumplimiento de principios SOLID**

A continuación, se muestra evidencia de como el diseño y la implementación cumplen con los principios S.O.L.I.D. Cada uno cuenta con una imagen con partes del diseño U.M.L, segmentos de código y una explicación de las estrategias o consideraciones que se tomaron en cuenta para el cumplimiento del principio.

**Single Responsability Principle (S.R.P)**

**Diseño UML:**

Texto

Descripción generada automáticamente

**Segmentos de código:**

Texto

Descripción generada automáticamente

**Estrategia tomada:** Para cumplir con este principio se optó por crear una clase con una sola razón para cambiar, cómo se ve en la imagen la clase Texto está diseñada exclusivamente para representar un texto y gestionar sus atributos y funcionalidades asociadas. Esta decisión se tomó con el objetivo de asegurar un código mantenible, escalable, comprensible y reutilizable, que tenga una única razón para cambiar.

**Open and Close Principle (O.C.P)**

**Diseño UML:**

**Diagrama

Descripción generada automáticamente  
Segmento de código:**

**Super clase abstracta:**

Texto

Descripción generada automáticamente

**Subclases concretas:**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Estrategia tomada:** Para cumplir con el principio de sustitución de OCP, se decidió crear una superclase que actúa como un esqueleto común para todas las subclases relacionadas. Esta superclase contiene métodos abstractos que representan las funcionalidades esenciales que deben ser implementadas por cada subclase, pero no define la implementación específica de cada método. Esto permite que cada subclase adapte y personalice la funcionalidad según sus necesidades particulares, mientras aún se adhiere al contrato establecido por la superclase. Si se requiere añadir un nuevo formato de bitácora no es necesario modificar el código existe, se puede extender una nueva subclase que implemente los métodos de la superclase.

**Liskov Substitution Principle (L.S.P)**

**Diseño UML:**

**Diagrama

Descripción generada automáticamente  
Segmento de código:**

**Super clase abstracta:**

Texto

Descripción generada automáticamente

**Subclases concretas:**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Estrategia tomada:** Paracumplir con este principio se creo una superclase que le hereda métodos a sus subclases, tanto las subclases con la superclase tienen la misma interfaz, además estas clases están abiertas a extensión, pero no a modificación del código existente.

**Interface Segregation Principle (I.S.P)**

**Diseño UML:**

Diagrama

Descripción generada automáticamente

**Segmentos de código:**

Texto

Descripción generada automáticamente Texto

Descripción generada automáticamente

**Estrategia tomada:** Para cumplir con este principio se realizaron interfaces específicas, cómo se ve en las imágenes, se creó una interfaz ITematica enfocada solo en objetos de tipo temática y una interfaz ITexto enfocada solo en objetos tipo texto. Esto cumpliendo con el principio de Interface Segregation, el cual expresa que es mejor tener varias interfaces específicas a tener una sola a nivel general, cuidando la alta cohesión de la interface creada asignándole una sola razón para cambiar a cada interface.

**Dependency Inversion Principle (D.I.P)**

**Diseño UML:**

Diagrama

Descripción generada automáticamente

**Segmentos de código:**

Texto

Descripción generada automáticamente Texto

Descripción generada automáticamente

**Estrategia tomada:** Con el fin de cumplir este principio se optó por evitar vincular directamente clases concretas, esto se realizó por medio de una capa de abstracción que se incorporó en el diseño, específicamente una interface. Esta interface actúa como un contrato que define el comportamiento que debe tener cualquier clase que la implemente, permitiendo así desacoplar las clases concretas de los detalles de implementación.

**Separación de responsabilidades**

A continuación, se muestra evidencia de como el diseño y la implementación cumplen con la separación de responsabilidad desde el punto de vista M.V.C. Cada uno cuenta con una imagen con partes del diseño U.M.L, segmentos de código y una explicación de las estrategias o consideraciones que se tomaron en cuenta para el cumplimiento del SoC🡪M.V.C.

**Diseño UML:**

Diagrama

Descripción generada automáticamente

Diagrama, Esquemático

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

**Segmentos de código:**

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

**Estrategia tomada:** Se optó por la utilización de varios paquetes para dividir las responsabilidades de manera más efectiva. En este diseño, el paquete "repository" se encarga de todo lo relacionado con la base de datos, mientras que el "service" consume los servicios del "repository" y del "model". Por su parte, el "controller" actúa como un intermediario entre el front-end y el back-end, consumiendo los servicios del "service" para que la vista pueda utilizarlos. Esta estructura sigue el principio del Modelo-Vista-Controlador (MVC), donde la capa de modelo permanece independiente de la capa de vista, asegurando así una separación clara de responsabilidades y cumpliendo con el principio de Separation of Concerns (SoC).

**Patrones de diseño**

**Patrón de diseño creacional:** **Singleton**

**UML:**

Diagrama

Descripción generada automáticamente

**Código:**

Texto

Descripción generada automáticamente

**Clase que utiliza el patrón:**

Captura de pantalla de computadora

Descripción generada automáticamente

**Estrategia tomada:**

Se opto por este patrón ya que así se garantiza que la clase ConexionIA tenga una única instancia, proporcionando un punto de acceso global a esa instancia. Se utiliza para controlar el acceso a recursos compartidos, como bases de datos o archivos de configuración, asegurando consistencia global y evitando problemas de concurrencia. Además, reduce la sobrecarga de recursos al reutilizar una sola instancia y facilita el acceso a componentes críticos del sistema, simplificando el diseño general y evitando la creación accidental de múltiples instancias.

**Patrón de comportamiento: Observador**

**UML:**

**Diagrama

Descripción generada automáticamente**

**Código:**

**Super clase abstracta:**

Texto

Descripción generada automáticamente

**Observadores**

Observador 1:

Texto

Descripción generada automáticamente

Observador 2:

Texto

Descripción generada automáticamente

Observador 3:

Texto

Descripción generada automáticamente

**Clase que utiliza el patrón:**

Texto

Descripción generada automáticamente

**Estrategia:** Ya que el patrón Observador es una estrategia de diseño de software utilizada cuando se desea que un objeto (sujeto) notifique automáticamente a un conjunto de objetos dependientes (observadores) sobre cualquier cambio en su estado. Se opta la utilización de este patrón ya que se requería de realizar bitácoras en tres formatos distintos esos tres formatos son observadores suscritos a un objeto que es cambiante y cada vez que realice un cambio es necesario que sea notificado a los observadores para que sea agregado a las bitácoras.

**Patrón de diseño estructural: decorador**

**UML:**

**Diagrama

Descripción generada automáticamente**

**Código**

**Interface original:**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Clase que posee una relación de realización con la interface original:**

Texto

Descripción generada automáticamente

**Decorador de idioma:**

Captura de pantalla de un celular

Descripción generada automáticamente

**Estrategia:** Ya que este es un patrón que se utiliza a la hora de dar mantenimiento y se requería enviar un PDF en otro idioma como parte de un mantenimiento. Aquí es donde entra en juego el patrón decorador. En lugar de modificar directamente el código existente para manejar el PDF en otro idioma, se optó por aplicar el patrón decorador. Esto significa que se creó una clase decoradora que se encarga de añadir la funcionalidad de generar el PDF en el idioma requerido, sin modificar el código original de la clase que genera el PDF en el idioma predeterminado. La ventaja clave de utilizar el patrón decorador en este contexto es que facilita la extensión y el mantenimiento del sistema en el futuro. Si en algún momento se necesitan agregar más funcionalidades relacionadas con la generación de PDF, se pueden crear nuevas clases decoradoras sin afectar el código existente. Esto hace que el sistema sea más flexible, escalable y fácil de mantener a largo plazo.