**Principios de Diseño**

1. **Single Responsability Principle (S)**
2. **Open/Close Principle (O)**
3. **Liskov Substitution Principle (L)**
4. **Interface Segregation Principle (I)**
5. **Dependency Inversion Principle (D)**
6. **Encapsular la variabilidad**
7. **Diseñar hacia interfaces, no hacia implementaciones**
8. **Favorecer composición antes que herencia**
9. **Ley Demeter (No hables con extraños)**
10. **Hollywood**
11. **Una y solo una regla**
12. **Modularidad, cohesión y acoplamiento**

**SINGLE RESPONSABILITY**

* Una clase o función de una clase deben tener una única razón para cambiar.
* Cuando tenemos una clase o una función, debemos analizar si existe más de una posible razón que puedan generar cambios a la misma, si es así, puede ser que esta se esté dedicando a más de un tipo de responsabilidad, con lo cual, será más probable que cambie a menudo, y será potencialmente menos reutilizable.

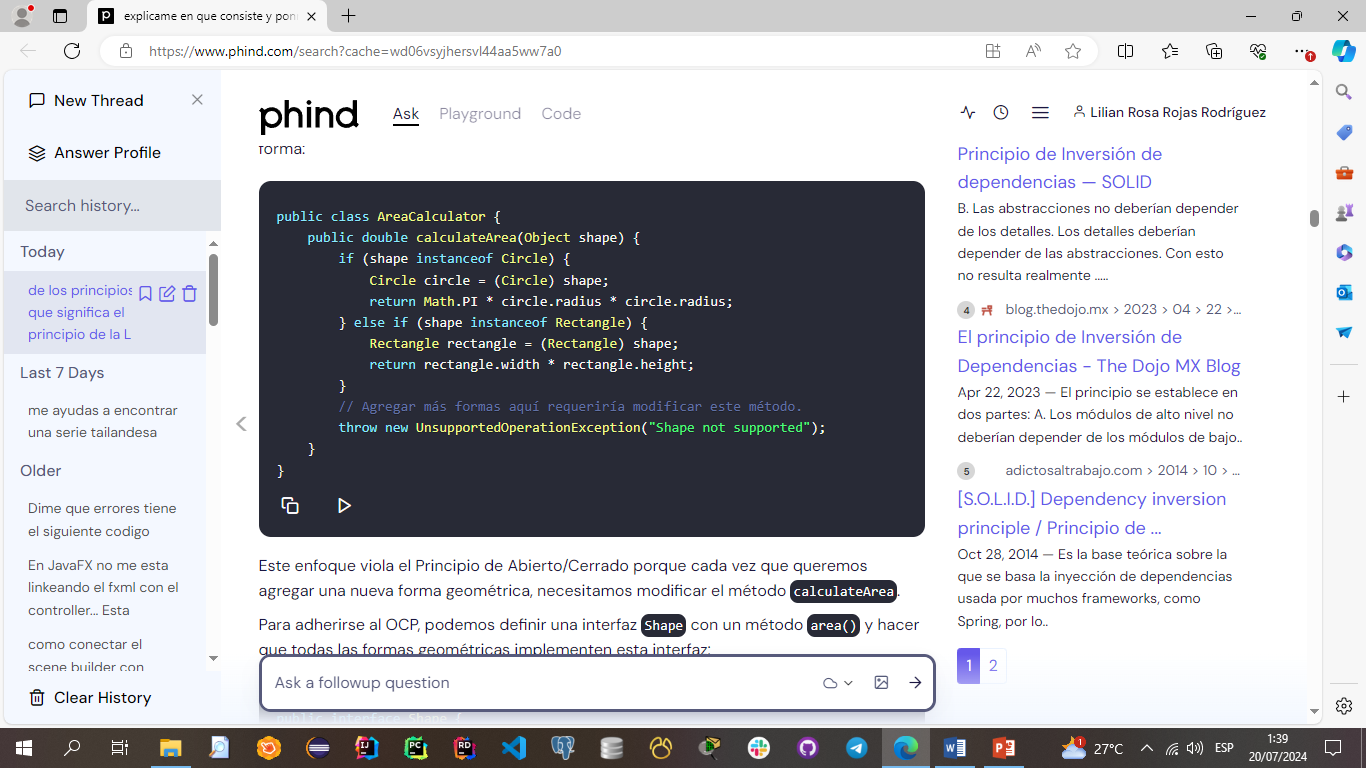
**Ejemplo**

Consideremos una clase Student que inicialmente maneja tres responsabilidades diferentes: registrar estudiantes, calcular sus resultados y enviar correos electrónicos a los estudiantes. Esta clase viola el Principio de Responsabilidad Única ya que tiene múltiples responsabilidades. Para adherirse al SRP, deberíamos dividir esta clase en tres clases separadas, cada una manejando una sola responsabilidad. Al hacer esto, cada clase tiene ahora una única responsabilidad, lo que facilita su mantenimiento y modificación sin afectar otras partes del código.

**OPEN/CLOSE**

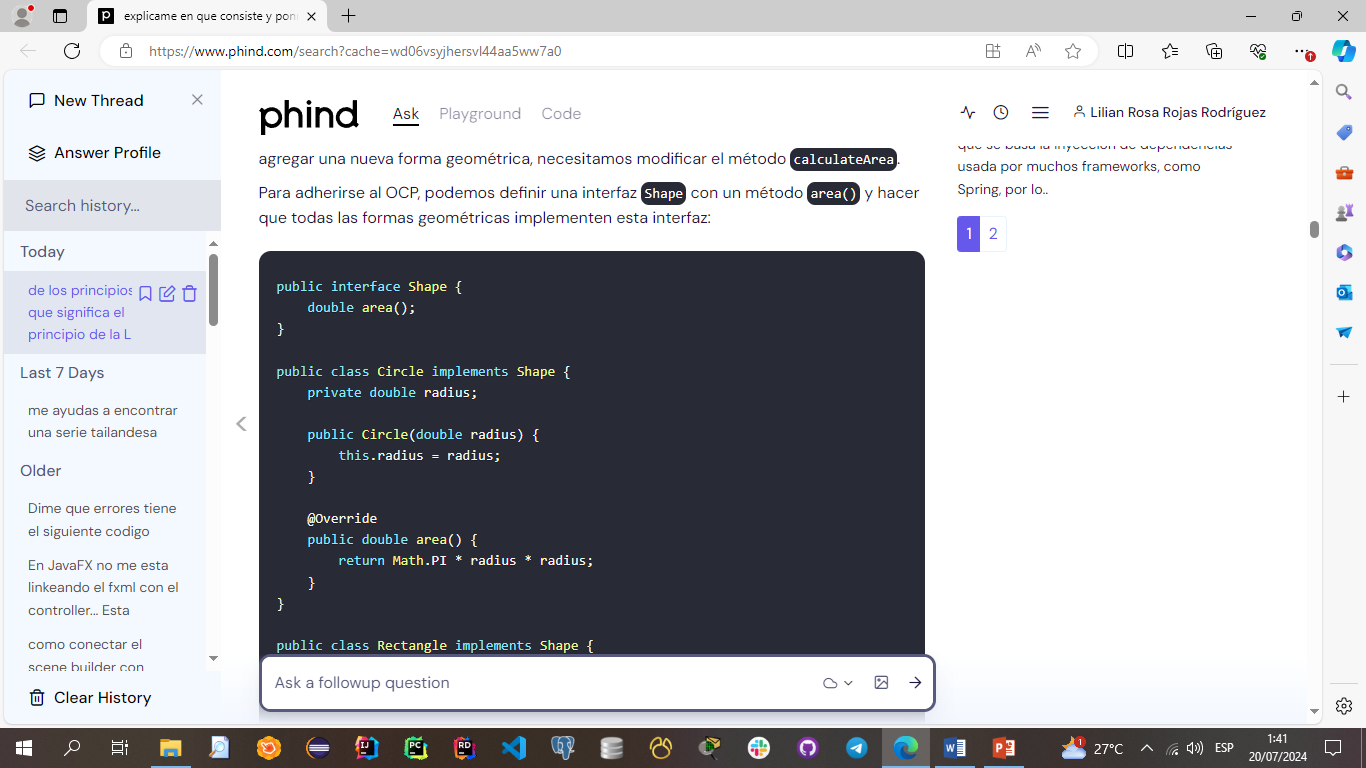
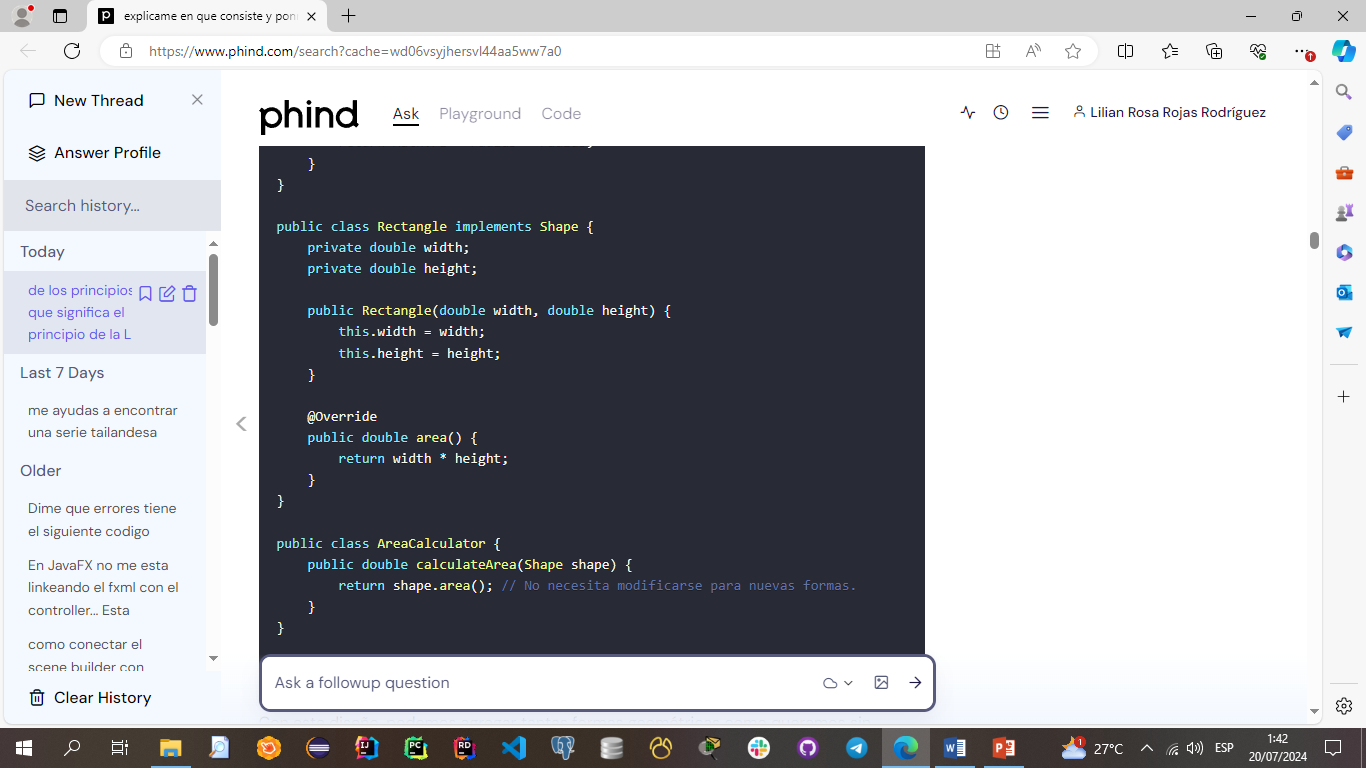
* El diseño debe quedar abierto para extensiones y cerrado para modificaciones.
* Identificar los cambios probables y dejar puntos de extensión necesarios abiertos, mientras deja cerrada la solución para las modificaciones más probables.
* Esto se logra generalmente mediante el uso de abstracciones y polimorfismo, permitiendo que nuevas funcionalidades se añadan mediante la creación de nuevas clases que extienden o implementan las abstracciones existentes, en lugar de modificar las clases ya existentes.

**Ejemplo**

Consideremos un sistema de cálculo de áreas para diferentes formas geométricas. Sin aplicar el OCP, podríamos tener una función que calcule el área basándose en el tipo de forma:

Este enfoque viola el Principio de Abierto/Cerrado porque cada vez que queremos agregar una nueva forma geométrica, necesitamos modificar el método calculateArea.

Para adherirse al OCP, podemos definir una interfaz Shape con un método area() y hacer que todas las formas geométricas implementen esta interfaz:



Con este diseño, podemos agregar tantas formas geométricas como queramos sin necesidad de modificar el código existente del AreaCalculator, cumpliendo así con el Principio de Abierto/Cerrado.

**LISKOV SUBSTITUTION**

El Principio de Sustitución de Liskov asegura que una subclase pueda asumir el lugar de su superclase sin afectar la funcionalidad esperada del programa. Para cumplir con este principio, es necesario que:

* Los objetos de una superclase puedan ser reemplazados por objetos de una subclase sin afectar la corrección del programa.
* Las subclases deben ser substituibles por sus superclases sin alterar las propiedades deseables del programa (tales como la corrección).

**Ejemplo**

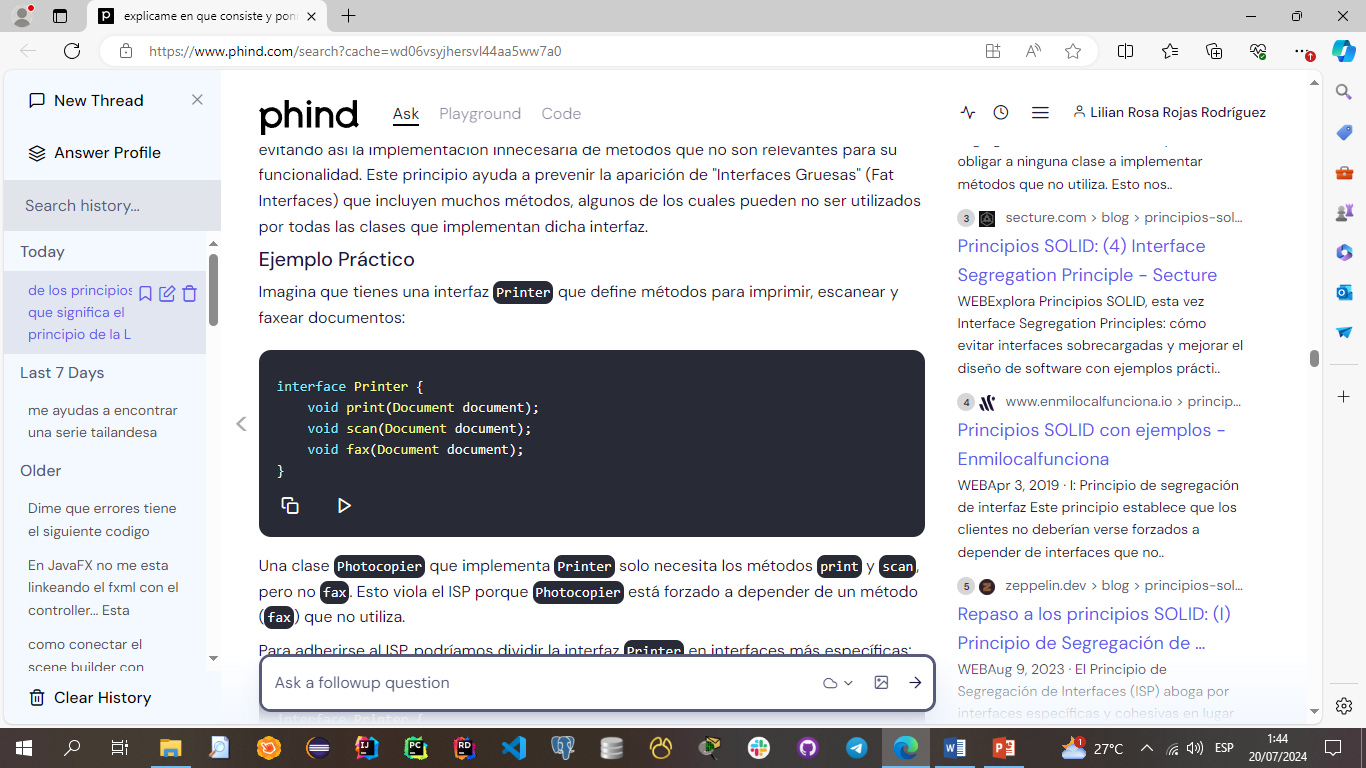
Imagina que tienes una clase Bird (Ave) con un método fly() (volar). Si creas una subclase Penguin (Pingüino) que hereda de Bird, pero los pingüinos no pueden volar, esto violaría el Principio de Sustitución de Liskov porque no puedes reemplazar un objeto Bird con un objeto Penguin sin alterar la funcionalidad esperada del programa (en este caso, la capacidad de volar).

Para adherirse al LSP, podrías refactorizar tus clases para que la capacidad de volar no esté implícita en la superclase Bird. Una solución podría ser tener una interfaz FlyingBird (AveVoladora) que tenga el método fly(), y hacer que solo aquellas aves que pueden volar implementen esta interfaz. De esta manera, Penguin no implementaría FlyingBird, y el programa podría tratar adecuadamente con ambos tipos de aves sin violar el LSP

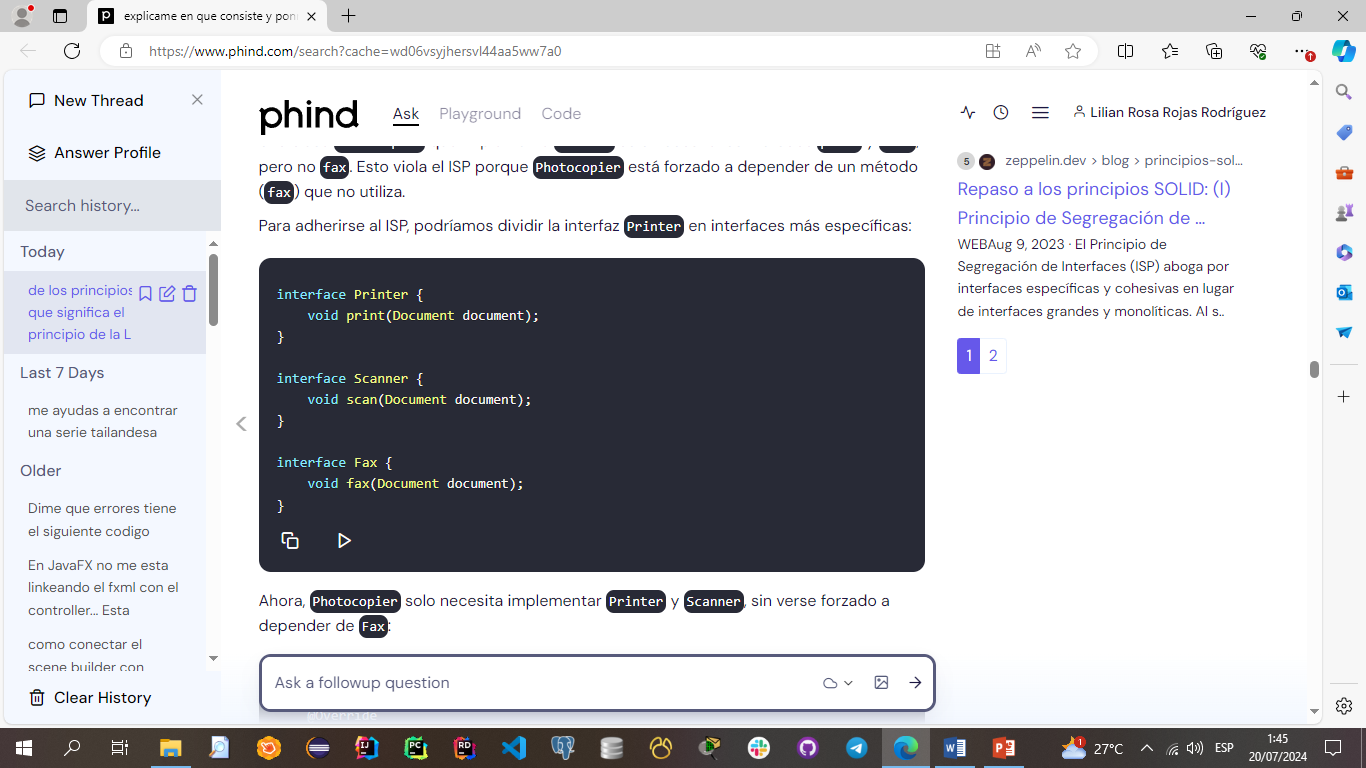
**INTERFACE SEGREGATION**

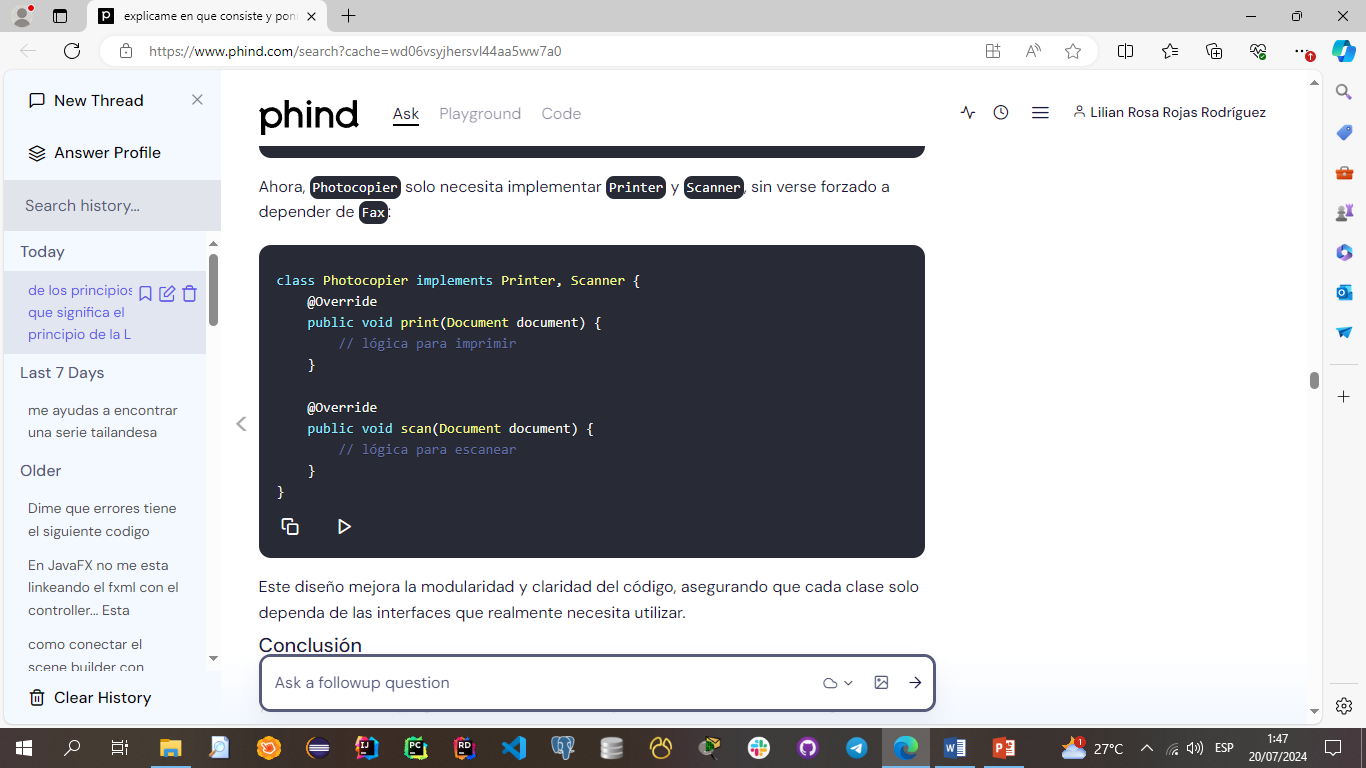
* Principio de subdivisión de Interfaces: Los clientes no deberían depender de interfaces que no emplean.
* Las interfaces deberían ser tan pequeñas como sea posible, de modo que mantengan la cohesión de las funciones que declaran.
* Si una misma interfaz es empleada por más de un tipo de cliente, y cada uno tiene sus necesidades separadas, posiblemente la solución será dividir esta interfaz “gruesa” en dos para hacer cada una de las nuevas interfaces ajustada a las necesidades de su cliente.

**Ejemplo**

Imagina que tienes una interfaz Printer que define métodos para imprimir, escanear y faxear documentos:

Una clase Photocopier que implementa Printer solo necesita los métodos print y scan, pero no fax. Esto viola el ISP porque Photocopier está forzado a depender de un método (fax) que no utiliza.

Para adherirse al ISP, podríamos dividir la interfaz Printer en interfaces más específicas:

Ahora, Photocopier solo necesita implementar Printer y Scanner, sin verse forzado a depender de Fax:

**DEPENDENCY INVERSION**

Este principio establece dos reglas fundamentales:

* Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de abstracciones.
* Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

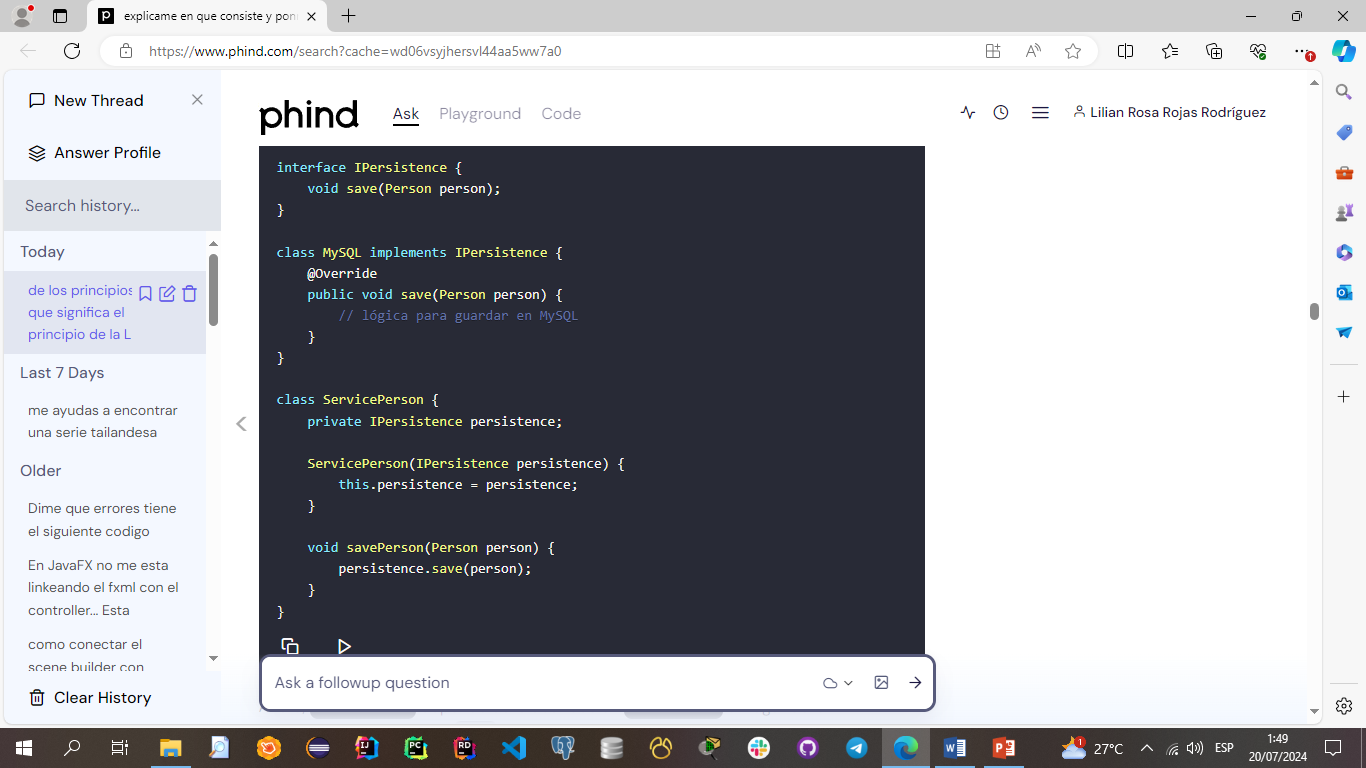
El objetivo del Principio de Inversión de Dependencias es reducir el acoplamiento entre las clases de alto nivel (aquellas que contienen la lógica de negocio) y las clases de bajo nivel (aquellas que realizan tareas específicas, como acceso a datos), haciendo que ambas dependan de abstracciones en lugar de depender directamente unas de otras. Esto se logra a través del uso de interfaces o clases abstractas que definen las operaciones que las clases de bajo nivel deben implementar, permitiendo así que las clases de alto nivel interactúen con estas abstracciones en lugar de con las implementaciones concretas.

**Ejemplo**

Supongamos que tenemos una aplicación que guarda información de personas en una base de datos MySQL. Tradicionalmente, podríamos tener una clase ServicePerson que crea una instancia de una clase MySQL para guardar la información:

Este diseño viola el Principio de Inversión de Dependencias porque ServicePerson (una clase de alto nivel) depende directamente de MySQL (una clase de bajo nivel). Si decidimos cambiar el sistema de almacenamiento a otro diferente a MySQL, tendríamos que modificar ServicePerson.

Para adherirse al DIP, creamos una interfaz IPersistence que abstrae la operación de guardar:



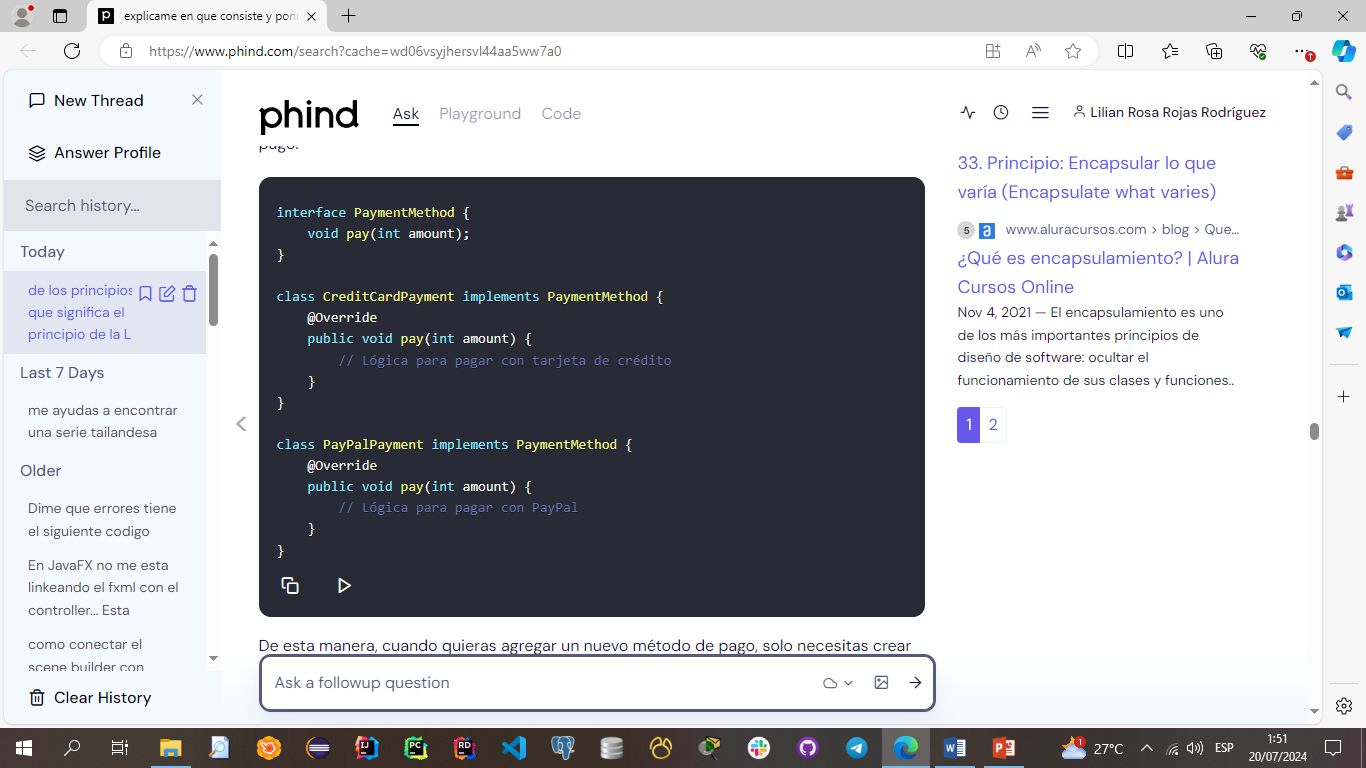
Ahora, ServicePerson depende de la abstracción IPersistence en lugar de la implementación concreta MySQL. Esto permite cambiar fácilmente el mecanismo de persistencia sin modificar ServicePerson, simplemente pasando una implementación diferente de IPersistence al constructor de ServicePerson.

**ENCAPSULAR LA VARIABILIDAD**

* Al diseñar, se deben reconocer las partes que son variables, de aquellas que no varían, y separarse entre sí.
* Las partes variables deben ser encapsuladas en piezas separadas de modo que las partes que no varían puedan estabilizarse rápidamente.
* Esto favorece también los casos en que las partes variables, pueden incrementarse o cambiarse en el tiempo.

**Ejemplo**

Imagina que estás desarrollando un sistema de comercio electrónico que actualmente soporta pagos a través de tarjetas de crédito. Sabes que en el futuro podrías querer agregar más métodos de pago, como PayPal o transferencias bancarias.

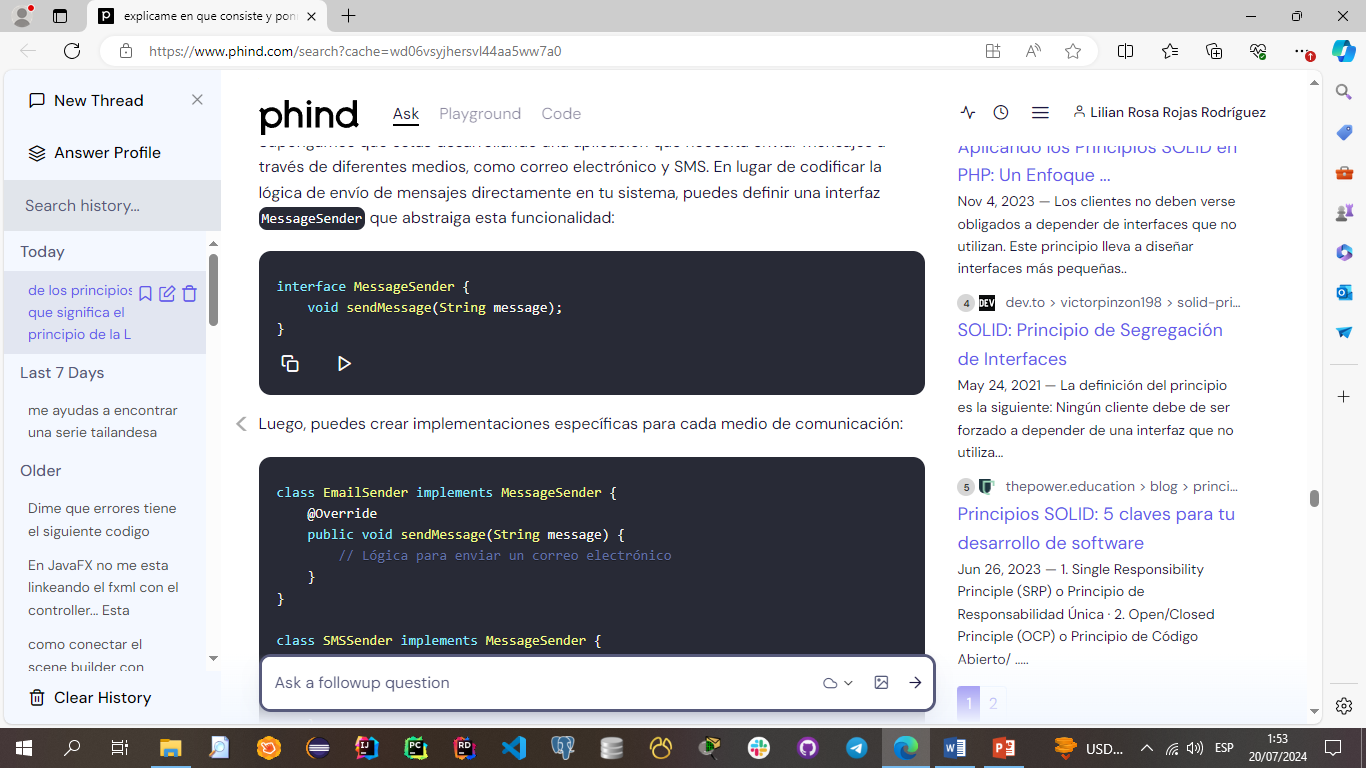
En lugar de codificar la lógica de pago directamente en tu sistema, puedes encapsular la variabilidad creando una interfaz PaymentMethod y clases concretas para cada método de pago:

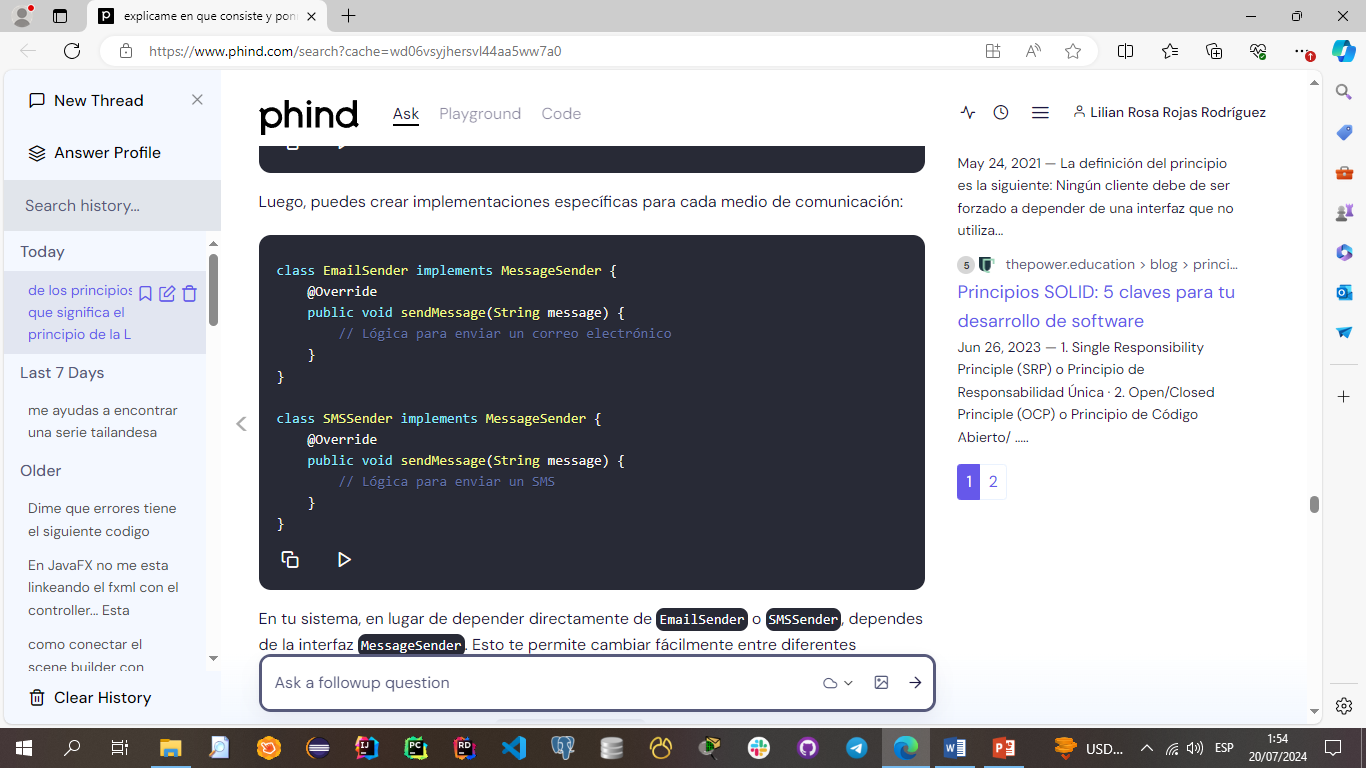
De esta manera, cuando quieras agregar un nuevo método de pago, solo necesitas crear una nueva clase que implemente PaymentMethod, sin tener que modificar el código existente que utiliza este método de pago.

**DISEÑAR HACIA INTERFACES, NO HACIA IMPLEMENTACIONES**

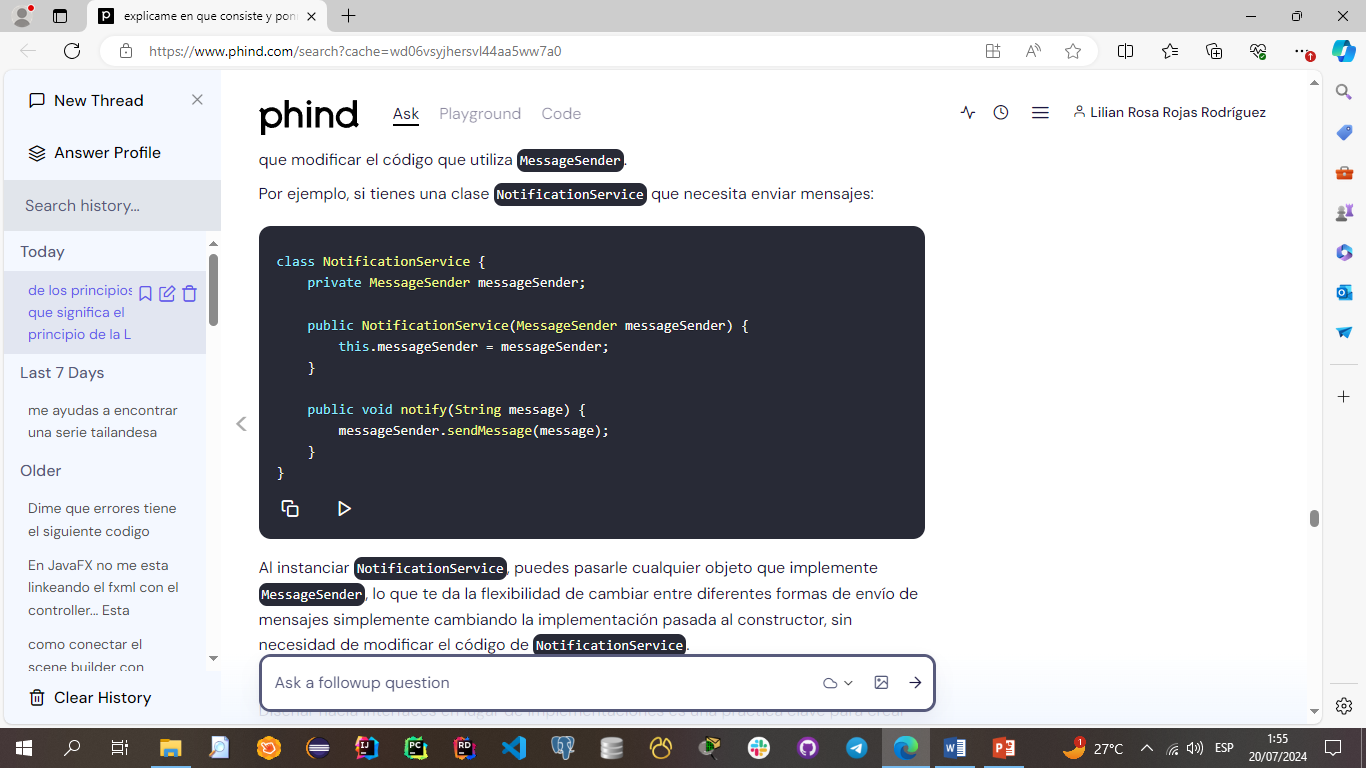
* Este principio reconoce la separación de la interfaz abstracta de su implementación concreta.
* Parte de la observación de que mientras las interfaces tienden a permanecer estables, las implementaciones pueden variar mucho, aparecer nuevas, modificarse formas existentes para ganar en una cualidad u otra como puede ser consumo de memoria o tiempo de respuesta.

**Ejemplo**

Supongamos que estás desarrollando una aplicación que necesita enviar mensajes a través de diferentes medios, como correo electrónico y SMS. En lugar de codificar la lógica de envío de mensajes directamente en tu sistema, puedes definir una interfaz MessageSender que abstraiga esta funcionalidad:

Luego, puedes crear implementaciones específicas para cada medio de comunicación:

En tu sistema, en lugar de depender directamente de EmailSender o SMSSender, dependes de la interfaz MessageSender. Esto te permite cambiar fácilmente entre diferentes métodos de envío de mensajes o incluso agregar nuevos métodos en el futuro sin tener que modificar el código que utiliza MessageSender.

Por ejemplo, si tienes una clase NotificationService que necesita enviar mensajes:

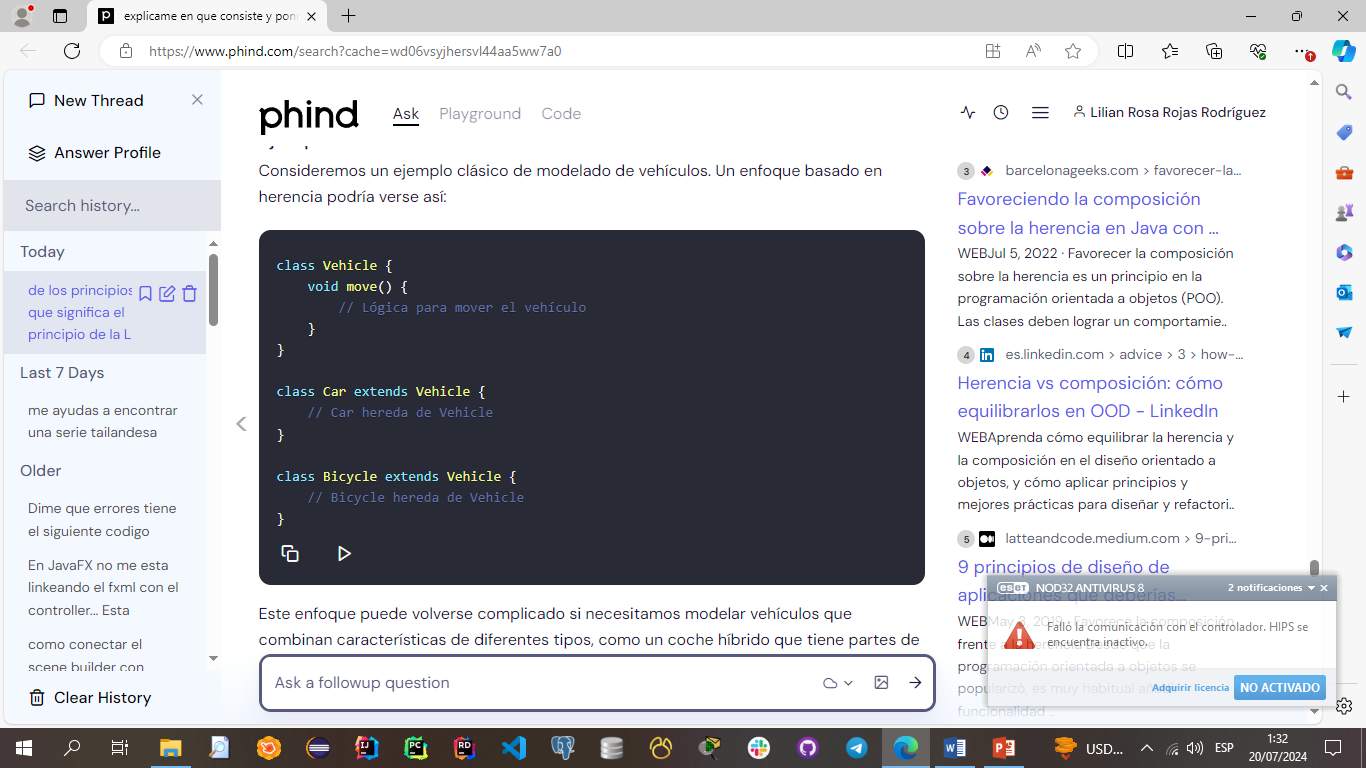
l instanciar NotificationService, puedes pasarle cualquier objeto que implemente MessageSender, lo que te da la flexibilidad de cambiar entre diferentes formas de envío de mensajes simplemente cambiando la implementación pasada al constructor, sin necesidad de modificar el código de NotificationService.

**FAVORECER COMPOSICIÓN ANTES QUE HERENCIA**

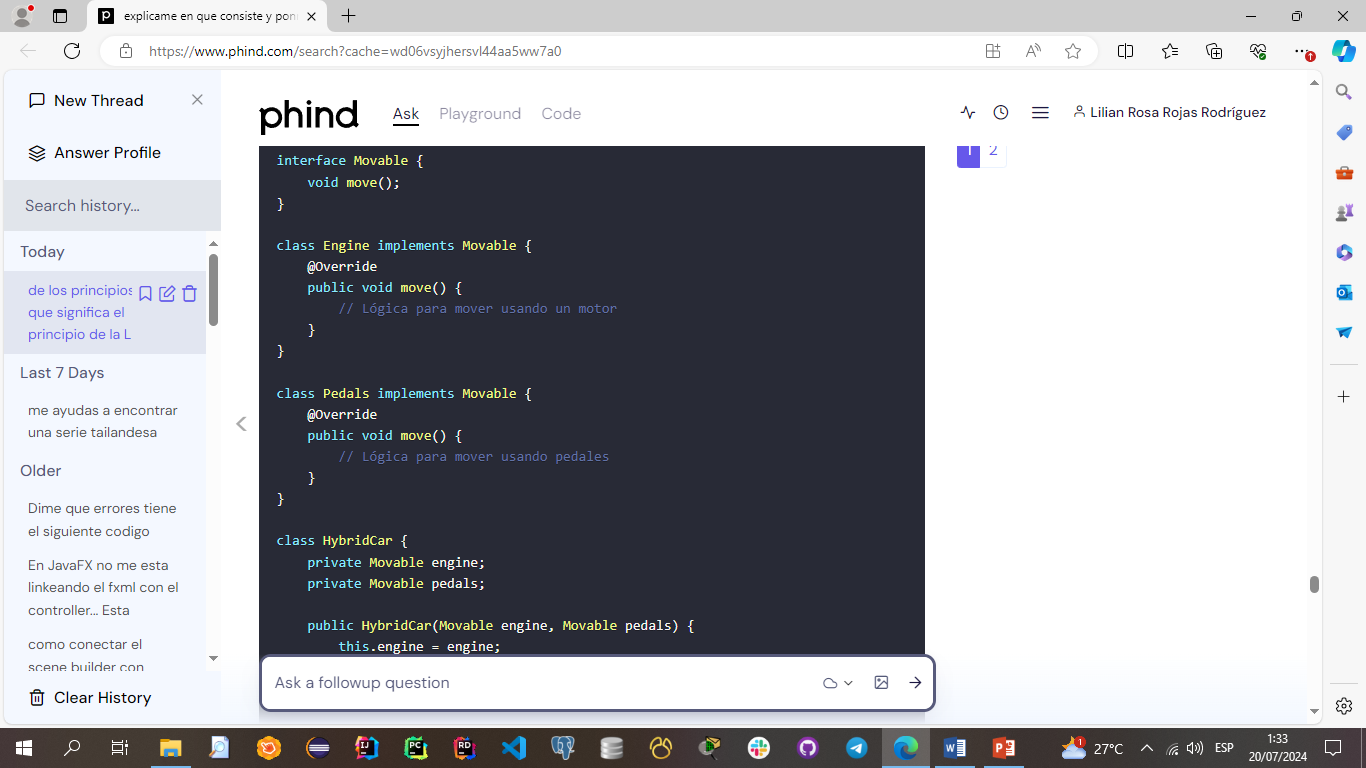
La composición permite una mayor flexibilidad ya que facilita la modificación y extensión del comportamiento de los objetos en tiempo de ejecución, simplemente cambiando las instancias de los objetos contenidos.

El principio de diseño de software "Favorecer la composición antes que la herencia" sugiere que es preferible construir relaciones entre objetos mediante la composición de objetos más pequeños con comportamientos específicos, en lugar de utilizar la herencia para extender funcionalidades de una clase base o padre. Este enfoque promueve un diseño más flexible y mantenible, permitiendo reutilizar y combinar comportamientos de manera más efectiva sin estar atado a una jerarquía rígida de clases.

**Ejemplo**

Consideremos un ejemplo clásico de modelado de vehículos. Un enfoque basado en herencia podría verse así:

Este enfoque puede volverse complicado si necesitamos modelar vehículos que combinan características de diferentes tipos, como un coche híbrido que tiene partes de un automóvil y partes de una bicicleta.

Un enfoque basado en composición sería diferente:



En este caso, HybridCar utiliza composición para incluir tanto un Engine como Pedals, ambos implementando la interfaz Movable. Esto le da a HybridCar la capacidad de moverse de múltiples maneras sin necesidad de heredar de múltiples clases, lo cual no es posible en lenguajes como Java que no soportan herencia múltiple. Además, este diseño es más flexible y fácil de modificar o extender, ya que puedes cambiar cómo se mueve el HybridCar simplemente pasando diferentes implementaciones de Movable al constructor.

**NO HABLES CON EXTRAÑOS O LEY DEMETER**

Las piezas de código no deben revelar sus particularidades, y no deben interactuar con demasiadas otras “piezas”.

Dividir el código en “células” (clases, módulos, subsistemas) dependiendo de la complejidad, y mantener al mínimo posible las interacciones entre estas subdivisiones, de modo que si una de ellas requiere ser sustituida o reparada, el número de afectaciones se mantenga bajo.

Las operaciones de una clase, solamente deberían efectuar llamadas a objetos que cumplan una de las 4 características siguientes:

* 1. **A sí mismos.**
  2. **A objetos pasados como argumento.**
  3. **A objetos creados por ella.**
  4. **A objetos con los cuales tiene relaciones directas.**

**Diseñar componentes tímidos (shy), que respetan la Ley de Demeter**, ayuda a obtener bajo acoplamiento.

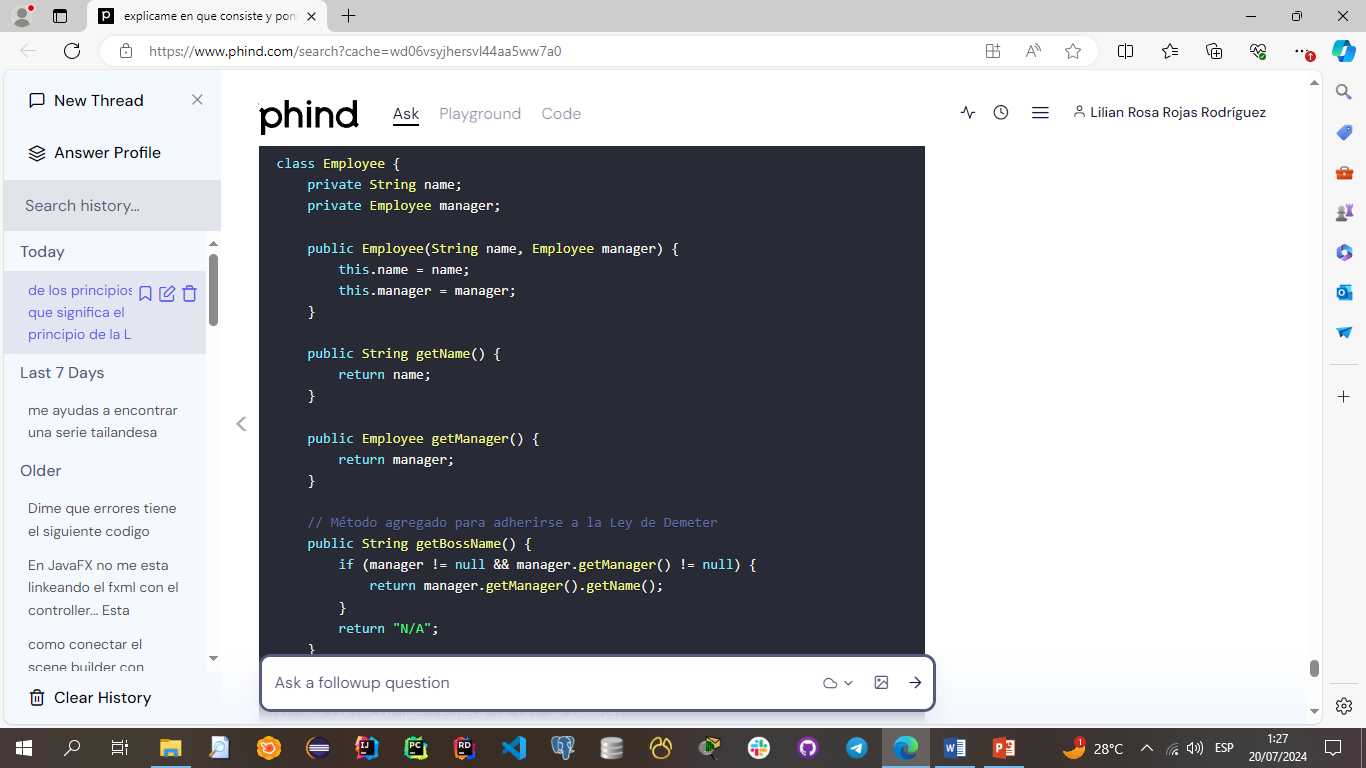
**Ejemplo**

Consideremos un sistema de gestión de empleados donde cada empleado tiene un jefe. Un enfoque que violaría la Ley de Demeter sería intentar obtener el nombre del jefe del jefe de un empleado, accediendo a propiedades anidadas de manera encadenada:



Este código viola la Ley de Demeter porque employee está accediendo a la propiedad manager de su manager, lo cual es considerado hablar con un "extraño".

Para adherirse a la Ley de Demeter, podríamos refactorizar el código para que cada objeto solo interactúe con sus colaboradores directos. Una forma de hacerlo sería agregar un método en la clase Employee que encapsule la lógica de obtener el nombre del jefe del jefe:



En este caso, getBossName encapsula la lógica de acceder al nombre del jefe del jefe, manteniendo la interacción dentro de los límites establecidos por la Ley de Demeter. Esto reduce el acoplamiento entre objetos y hace que el código sea más fácil de mantener y entender, ya que cada objeto solo interactúa con sus colaboradores directos.

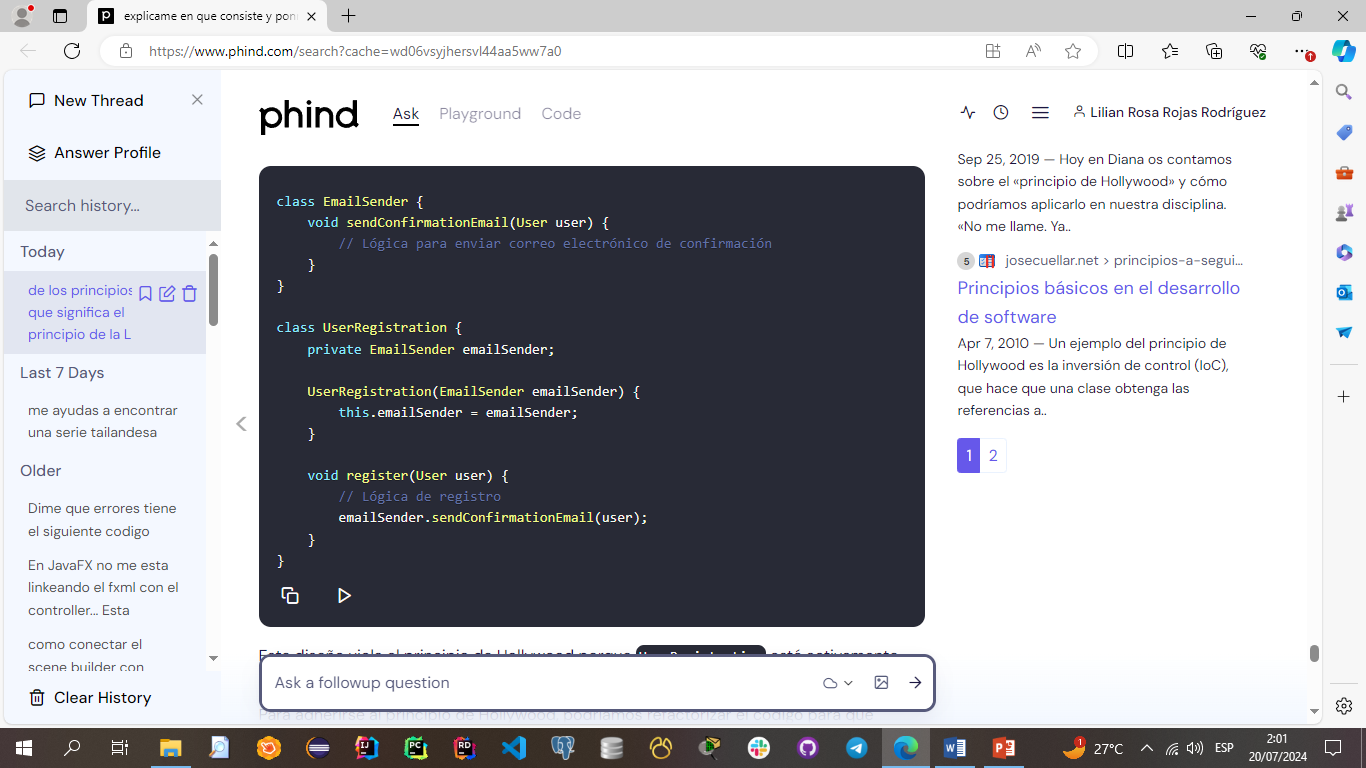
**HOLLYWOOD**

* Al diseñar piezas de alto nivel que interactuarán con piezas de más bajo nivel, se debe evitar las llamadas de estas últimas a las primeras, las piezas de más alto nivel deben mantenerse como coordinadoras, y no viceversa.
* La programación de una aplicación en la que se adicionan plugins debe seguir ese principio, el plugin no debe interactuar con su contenedor, sino al revés.

El principio de Hollywood, también conocido como el principio de "No nos llame, nosotros le llamaremos", es un concepto en el diseño de software que se relaciona estrechamente con el principio de inversión de control (IoC) y el principio de inversión de dependencias (uno de los cinco principios SOLID). Este principio sugiere que las clases de alto nivel no deberían depender directamente de las clases de bajo nivel, sino que deberían ser las clases de alto nivel las que determinen cuándo y cómo se invocan las clases de bajo nivel. Esto se logra a menudo a través de la inyección de dependencias, donde las dependencias necesarias son suministradas a una clase desde fuera, en lugar de que la clase las busque activamente.

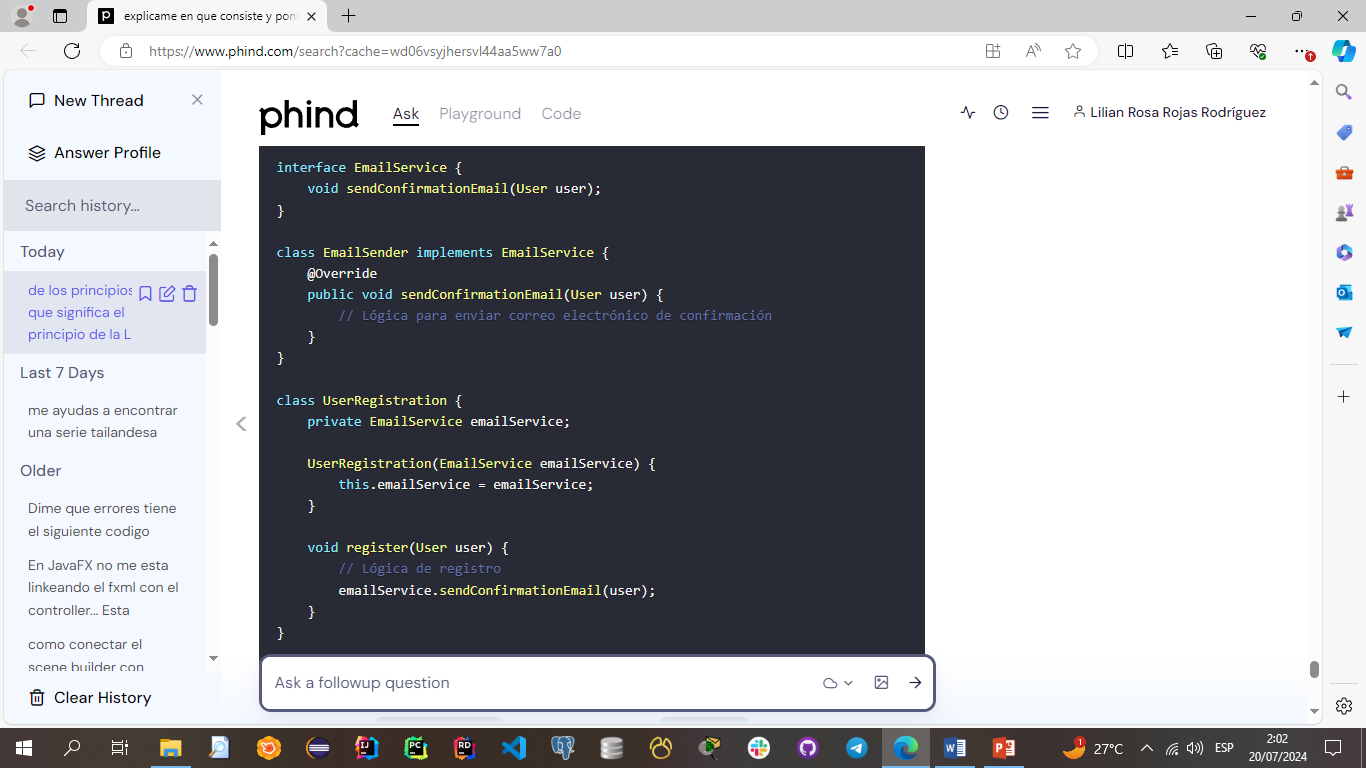
El principio de Hollywood promueve un diseño de software donde las decisiones sobre qué componentes interactúan y cómo se realizan estas interacciones son tomadas por componentes de alto nivel o frameworks, en lugar de por los componentes individuales de bajo nivel. Esto ayuda a mantener un bajo acoplamiento entre componentes, facilita la reutilización de código y mejora la flexibilidad y la escalabilidad del sistema.

**Ejemplo**

Consideremos un sistema simple de registro de usuarios donde necesitamos enviar un correo electrónico de confirmación después de que un usuario se registra. Sin aplicar el principio de Hollywood, podríamos tener una clase UserRegistration que depende directamente de una clase EmailSender para enviar el correo electrónico:

Este diseño viola el principio de Hollywood porque UserRegistration está activamente buscando y utilizando EmailSender.

Para adherirse al principio de Hollywood, podríamos refactorizar el código para que UserRegistration no dependa directamente de EmailSender, sino que reciba una referencia a un servicio de envío de correos electrónicos a través de la inyección de dependencias:



En este diseño, UserRegistration no sabe nada sobre EmailSender; solo sabe que necesita un EmailService para enviar correos electrónicos. La decisión de qué implementación específica de EmailService usar se toma en algún punto de alto nivel en la aplicación, posiblemente en el momento de la configuración del sistema, y se inyecta en UserRegistration a través del constructor. Esto permite cambiar fácilmente la implementación de EmailService sin modificar UserRegistration, siguiendo el principio de Hollywood.

**UNA Y SOLO UNA REGLA**

* Este principio establece la no repetición de una misma responsabilidad en más de una clase o módulo.
* En la medida en que una función es repetida en un mayor número de lugares, a la hora de modificar esa función, se pueden quedar varias de las piezas afectadas sin actualizar el cambio.

**Ejemplo**

Imaginemos que estamos diseñando una API RESTful para un sistema de gestión de tareas. Podríamos enfrentarnos a varias decisiones de diseño, como cómo manejar la paginación de resultados, cómo validar los datos entrantes y cómo manejar los errores.

Si aplicamos el principio de "Una y solo una regla", decidimos que nuestra API seguirá una única estrategia para cada uno de estos aspectos. Por ejemplo, para la paginación, podemos elegir implementar una paginación basada en cursor, donde cada respuesta incluye un cursor que apunta al siguiente conjunto de resultados. Esta es nuestra única regla para la paginación.

Para la validación de datos, podemos decidir que todos los campos requeridos deben ser validados en el servidor antes de procesar cualquier solicitud. Esta es nuestra única regla para la validación de datos.

Finalmente, para el manejo de errores, podemos decidir que toda la API devolverá códigos de estado HTTP estándar y mensajes de error descriptivos en formato JSON. Esta es nuestra única regla para el manejo de errores.

Aplicar el principio de "Una y solo una regla" en nuestro diseño garantiza que cada decisión de diseño esté claramente definida y que no haya ambigüedad sobre cómo se deben manejar diferentes aspectos de la API. Esto facilita la comprensión del sistema por parte de los desarrolladores y usuarios, y simplifica el proceso de mantenimiento y extensión del sistema en el futuro.

