**Patrones de Diseño GRASP**

**Patrones GRASP**

1. **Experto**
2. **Creador**
3. **Controlador**
4. **Indirección**
5. **Bajo Acoplamiento**
6. **Alta Cohesión**
7. **Polimorfismo**
8. **Fabricación Pura**
9. **No hables con extraños**

**EXPERTO**

**Otros Nombres:**

* Experto en Información
* Expert

**Objetivo:**

El objetivo principal del patrón Experto es asignar responsabilidades a las clases que tienen la información necesaria para cumplir con esas responsabilidades. Esto promueve la cohesión y reduce el acoplamiento, facilitando así el mantenimiento y la evolución del sistema.

**Aplicaciones:**

El patrón Experto se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara asignación de responsabilidades y una buena modularización. Ejemplos incluyen sistemas empresariales, aplicaciones de gestión, y cualquier sistema donde la información y las operaciones estén estrechamente relacionadas.

**Estructura:**

**Participantes:**

* **Experto**: La clase que tiene la información necesaria para realizar una tarea específica.

**Consecuencias:**

* **Ventajas**:
  + **Cohesión alta**: Las clases tienen responsabilidades claras y bien definidas.
  + **Acoplamiento bajo**: Reduce la dependencia entre clases.
  + **Mantenimiento y evolución facilitados**: Cambios en una parte del sistema tienen menos impacto en otras partes.
* **Desventajas**:
  + Puede llevar a la creación de muchas clases pequeñas, lo que puede complicar la estructura del sistema si no se gestiona adecuadamente.

**Usos conocidos:**

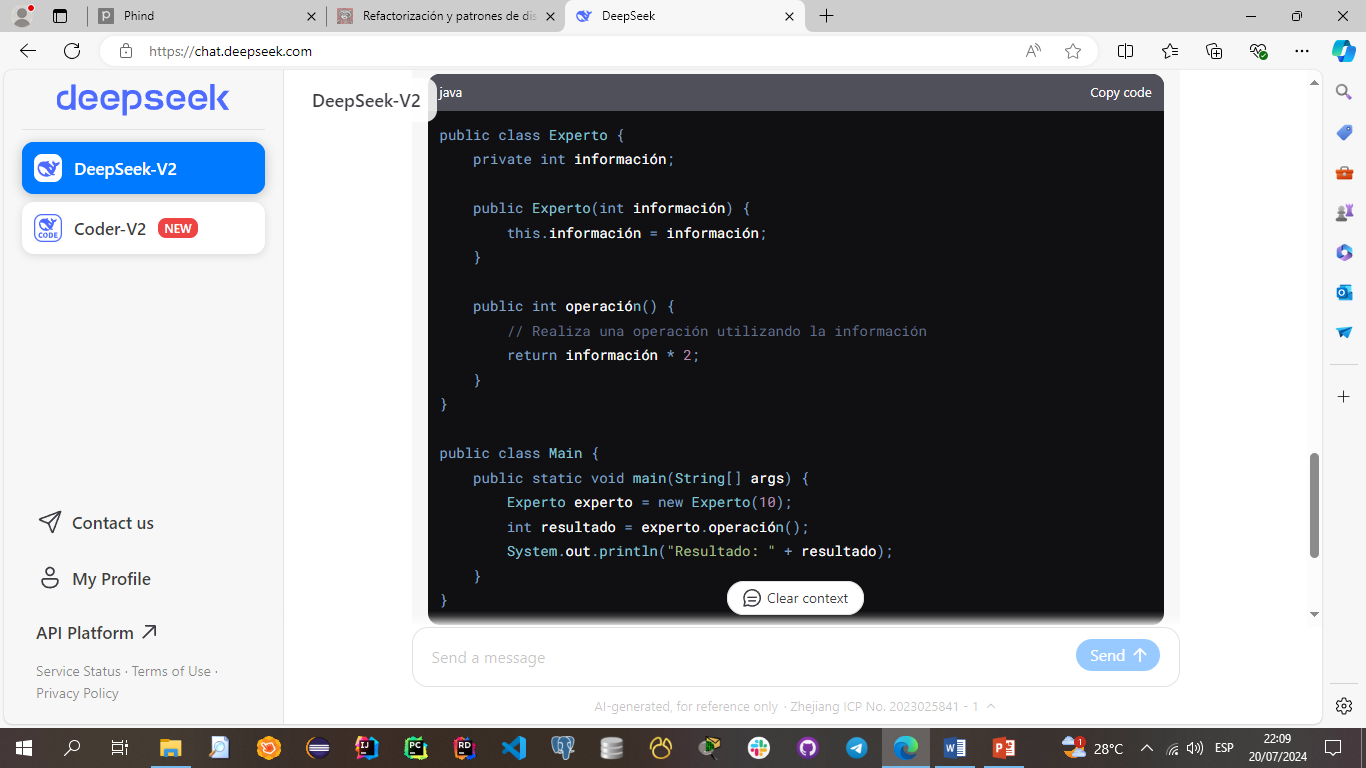
El patrón Experto es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de transacciones y seguridad.
* **Spring Framework**: En la inyección de dependencias.
* **.NET Framework**: En la gestión de datos y lógica de negocio.

**Patrones Relacionados:**

* **Patrón de Diseño Singleton**: Para asegurar que una clase tenga una única instancia.
* **Patrón de Diseño Factory**: Para la creación de objetos.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.

**Código de ejemplo:**



**CREADOR**

**Otros Nombres:**

* Creador
* Creator

**Objetivo:**

El objetivo principal del patrón Creador es asignar la responsabilidad de la creación de objetos a la clase que tiene la información necesaria para hacerlo de manera eficiente y coherente. Esto promueve la cohesión y reduce el acoplamiento, facilitando así el mantenimiento y la evolución del sistema.

**Aplicaciones:**

El patrón Creador se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara asignación de responsabilidades de creación de objetos. Ejemplos incluyen sistemas empresariales, aplicaciones de gestión, y cualquier sistema donde la creación de objetos esté estrechamente relacionada con la lógica de negocio.

**Estructura:**

**Participantes:**

* **Creador**: La clase que tiene la responsabilidad de crear instancias de otra clase.

**Consecuencias:**

**Consecuencias que trae su uso**

* **Ventajas**:
  + **Cohesión alta**: Las clases tienen responsabilidades claras y bien definidas.
  + **Acoplamiento bajo**: Reduce la dependencia entre clases.
  + **Mantenimiento y evolución facilitados**: Cambios en la creación de objetos tienen menos impacto en otras partes del sistema.
* **Desventajas**:
  + Puede llevar a la creación de muchas clases pequeñas, lo que puede complicar la estructura del sistema si no se gestiona adecuadamente.

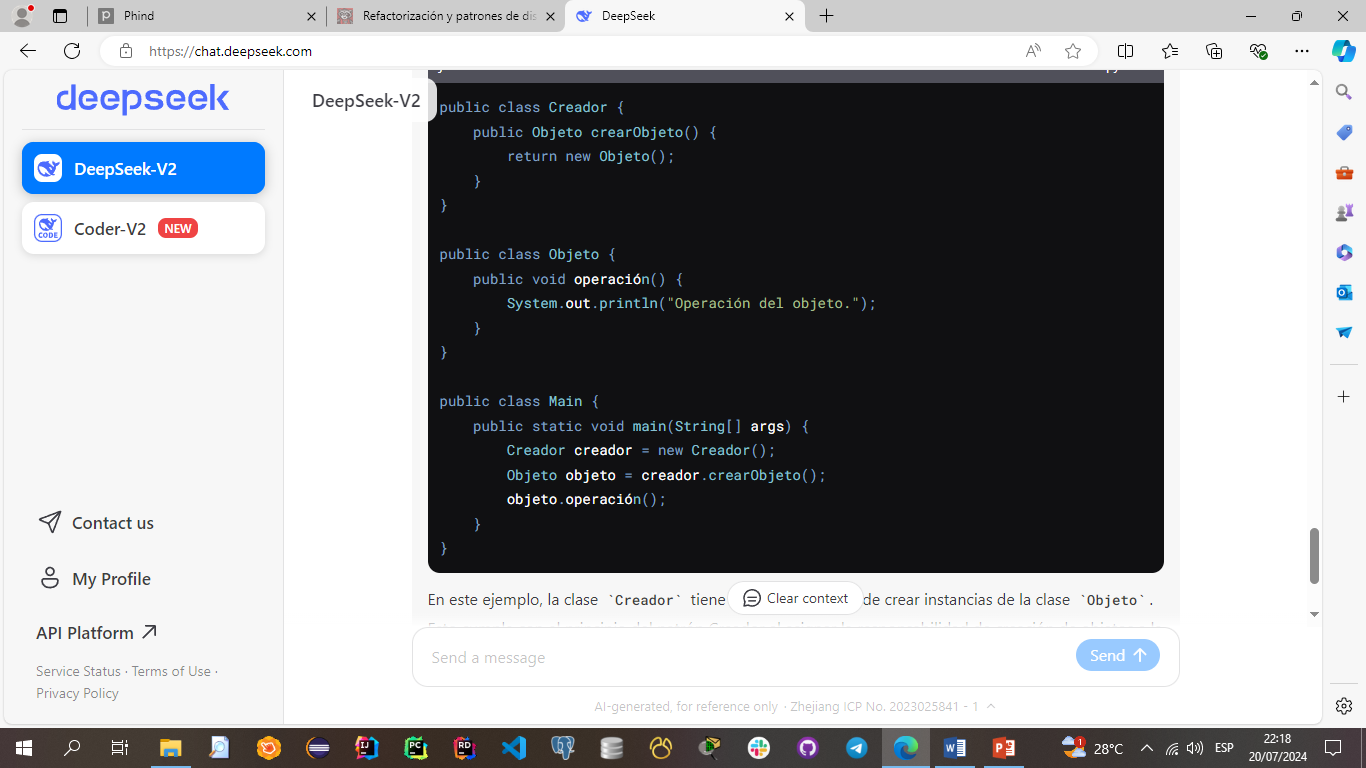
**Usos conocidos:**

El patrón Creador es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la creación de beans gestionados.
* **Spring Framework**: En la inyección de dependencias y la creación de beans.
* **.NET Framework**: En la creación de objetos en la capa de negocio.

**Patrones Relacionados:**

* **Patrón de Diseño Singleton**: Para asegurar que una clase tenga una única instancia.
* **Patrón de Diseño Factory Method**: Para la creación de objetos mediante un método de fábrica.
* **Patrón de Diseño Abstract Factory**: Para la creación de familias de objetos relacionados.

**Código de ejemplo:**

**CONTROLADOR**

**Otros Nombres:**

* Controlador
* Controller

**Objetivo:**

El objetivo principal del patrón Controlador es asignar la responsabilidad de manejar eventos y coordinar operaciones a una clase que actúa como intermediario entre la interfaz de usuario y el resto del sistema. Esto promueve la separación de preocupaciones, facilitando así el mantenimiento y la evolución del sistema.

**Aplicaciones:**

El patrón Controlador se utiliza en diversos sistemas de software, especialmente en aplicaciones que siguen la arquitectura Model-View-Controller (MVC). Ejemplos incluyen aplicaciones web, aplicaciones de escritorio, y cualquier sistema donde sea necesario separar la lógica de negocio de la interfaz de usuario.

**Estructura:**

**Participantes:**

* **Controlador**: La clase que tiene la responsabilidad de manejar eventos y coordinar operaciones.

**Consecuencias:**

* **Ventajas**:
  + **Separación de preocupaciones**: La lógica de negocio está separada de la interfaz de usuario.
  + **Reutilización**: El controlador puede ser reutilizado en diferentes contextos.
  + **Mantenimiento y evolución facilitados**: Cambios en la lógica de negocio tienen menos impacto en la interfaz de usuario y viceversa.
* **Desventajas**:
  + Puede llevar a la creación de controladores grandes y complejos si no se gestiona adecuadamente.

**Usos conocidos:**

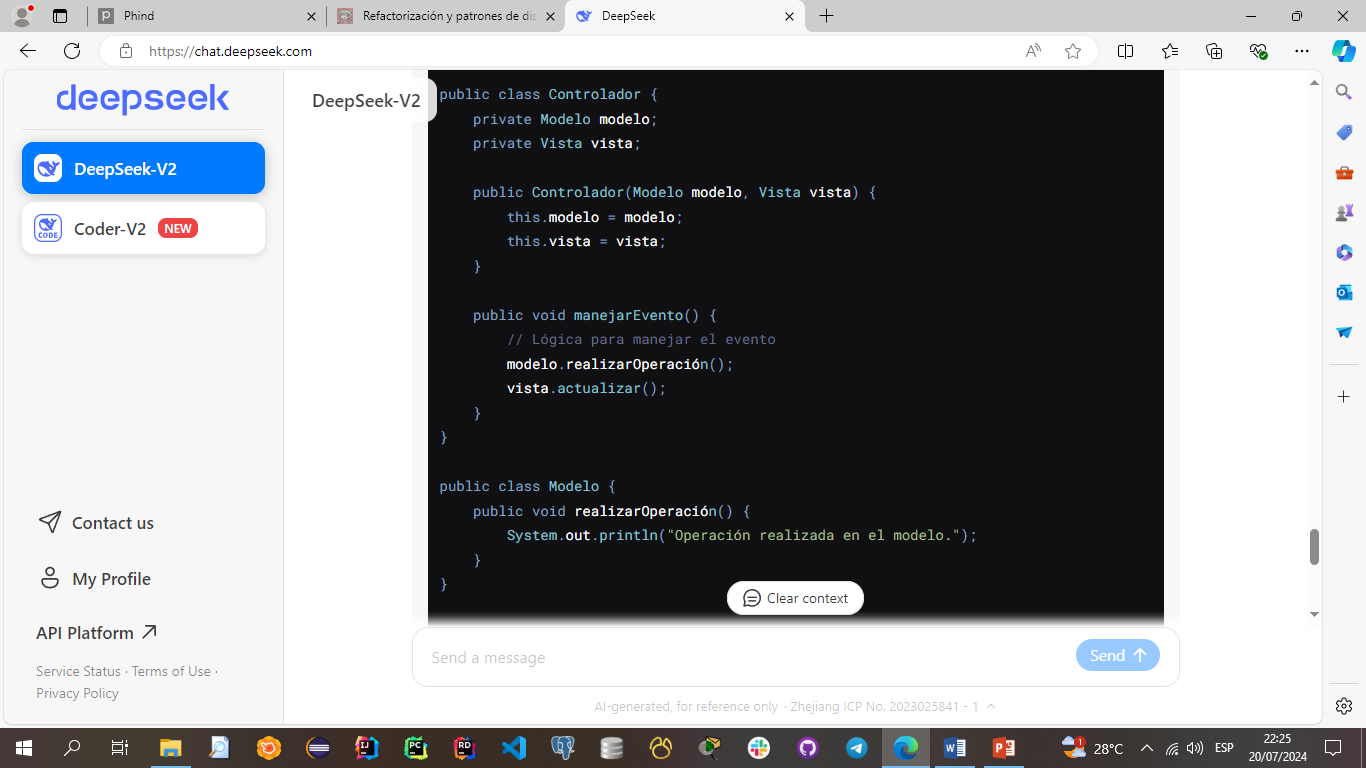
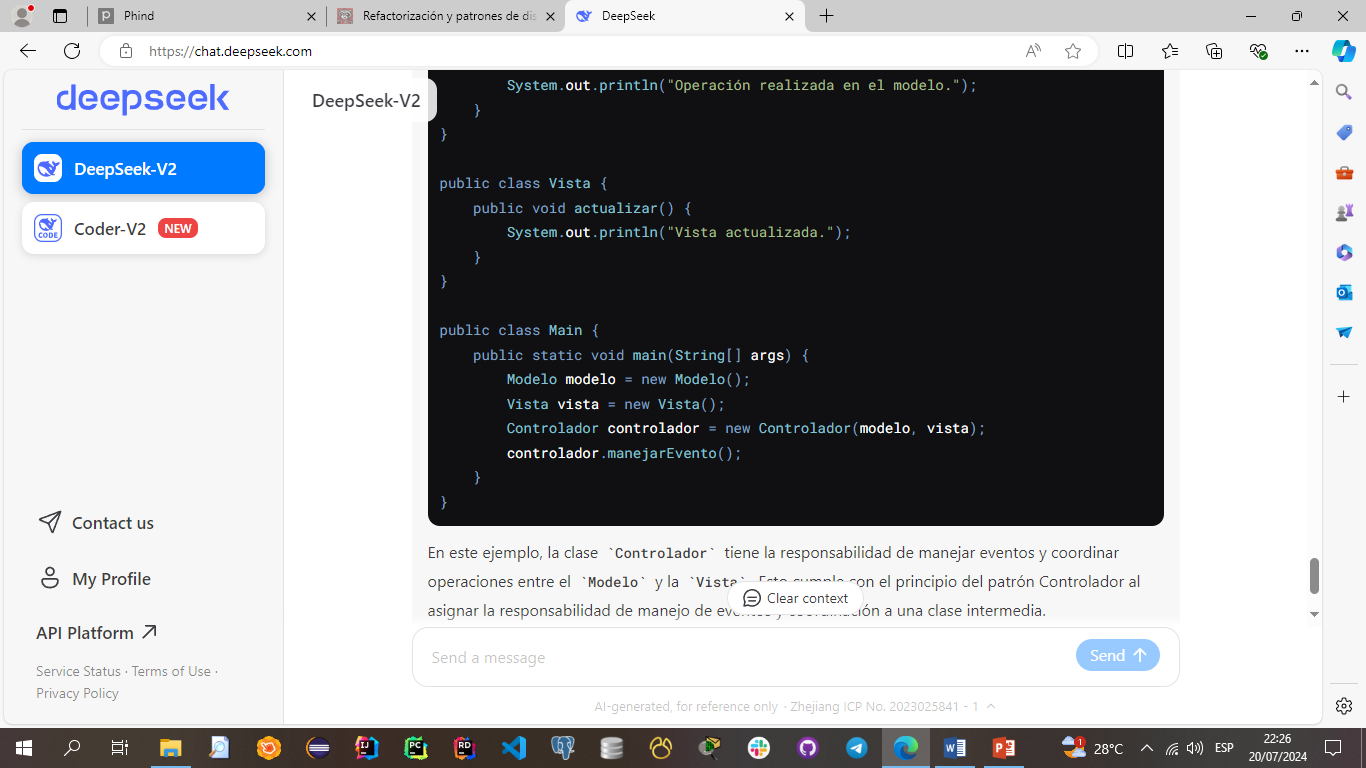
El patrón Controlador es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Spring MVC**: En la gestión de solicitudes HTTP.
* **ASP.NET MVC**: En la gestión de solicitudes web.
* **JavaFX**: En la gestión de eventos de la interfaz de usuario.

**Patrones Relacionados:**

* **Patrón de Diseño MVC (Model-View-Controller)**: Para separar la lógica de negocio, la interfaz de usuario y el control de flujo.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.
* **Patrón de Diseño Command**: Para encapsular solicitudes como objetos.

**Código de ejemplo:**



**INDIRECCIÓN**

**Otros Nombres:**

* Indirección
* Indirection

**Objetivo:**

El objetivo principal del patrón Indirección es introducir un intermediario entre dos componentes o sistemas para reducir su acoplamiento directo. Esto promueve la flexibilidad, la modularidad y la facilidad de mantenimiento del sistema.

**Aplicaciones:**

El patrón Indirección se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara separación de componentes y desacoplamiento. Ejemplos incluyen sistemas empresariales, aplicaciones distribuidas, y cualquier sistema donde sea necesario desacoplar componentes para facilitar cambios y evoluciones.

**Estructura:**

**Participantes:**

* **Intermediario**: La clase que actúa como intermediario entre dos componentes o sistemas.

**Consecuencias:**

* **Ventajas**:
  + **Desacoplamiento**: Reduce el acoplamiento directo entre componentes.
  + **Flexibilidad**: Facilita la modificación y evolución del sistema.
  + **Mantenimiento y evolución facilitados**: Cambios en un componente tienen menos impacto en otros componentes.
* **Desventajas**:
  + Puede introducir una capa adicional de complejidad si no se gestiona adecuadamente.

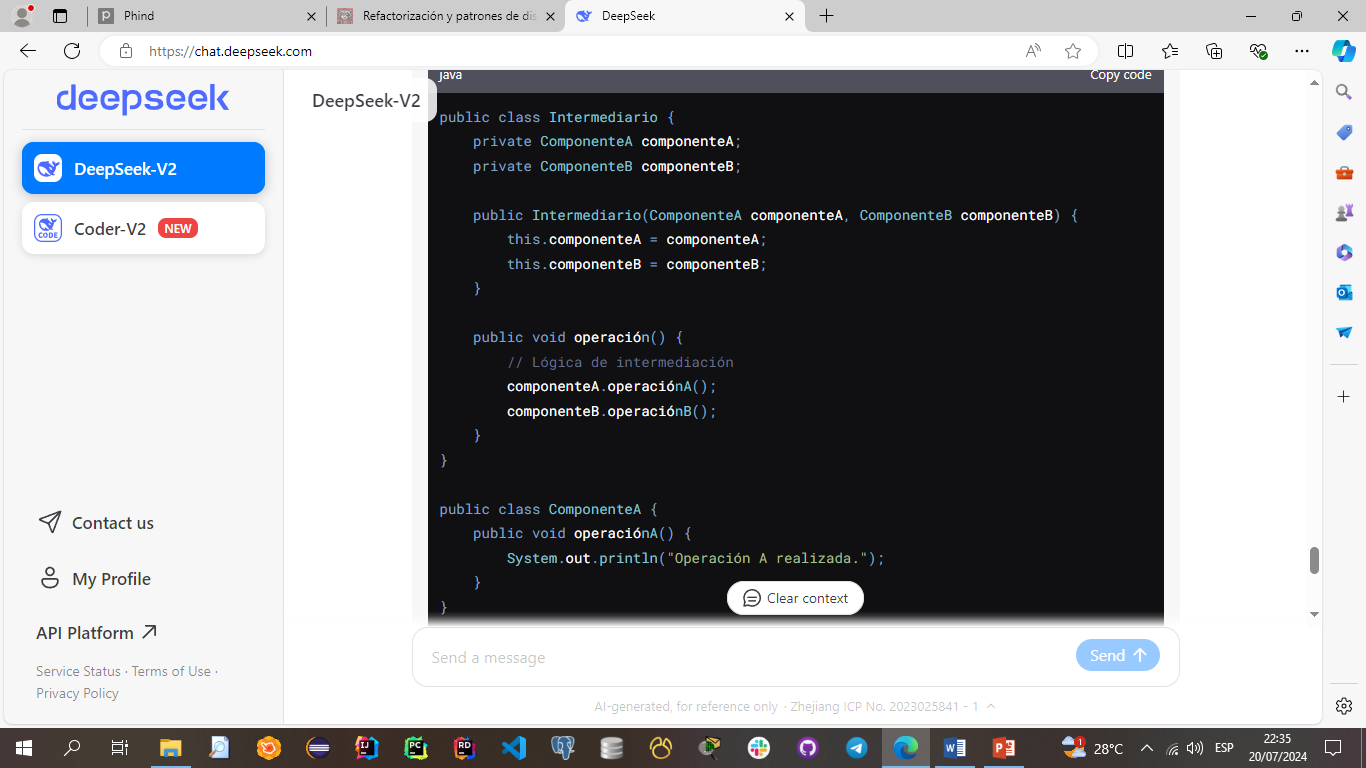
**Usos conocidos:**

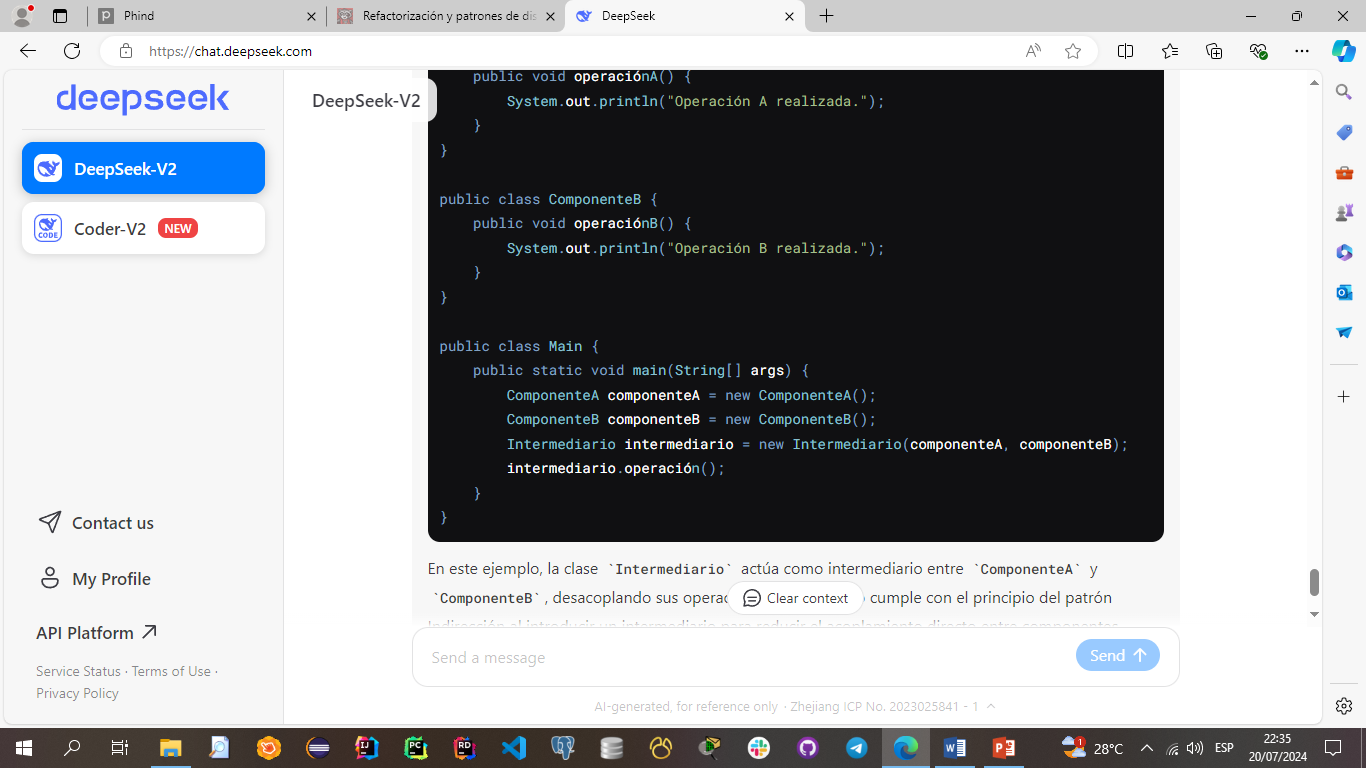
El patrón Indirección es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de servicios y componentes.
* **Spring Framework**: En la inyección de dependencias y la gestión de beans.
* **.NET Framework**: En la gestión de servicios y componentes.

**Patrones Relacionados:**

* **Patrón de Diseño Proxy**: Para proporcionar un intermediario para otro objeto.
* **Patrón de Diseño Mediator**: Para definir un objeto que encapsula cómo interactúan un conjunto de objetos.
* **Patrón de Diseño Adapter**: Para convertir la interfaz de una clase en otra interfaz que los clientes esperan.

**Código de ejemplo:**



**BAJO ACOPLAMIENTO**

**Otros Nombres:**

* Bajo acoplamiento
* Low coupling

**Objetivo:**

El objetivo principal del patrón Bajo Acoplamiento es diseñar componentes o módulos de tal manera que su dependencia mutua sea mínima. Esto promueve la flexibilidad, la reutilización y la facilidad de mantenimiento del sistema.

**Aplicaciones:**

El patrón Bajo Acoplamiento se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara separación de responsabilidades y desacoplamiento. Ejemplos incluyen sistemas empresariales, aplicaciones distribuidas, y cualquier sistema donde sea necesario desacoplar componentes para facilitar cambios y evoluciones.

**Estructura:**

**Participantes:**

* **ComponenteA**: Una clase o módulo con baja dependencia de otros componentes.
* **ComponenteB**: Otra clase o módulo con baja dependencia de otros componentes.

**Consecuencias:**

* **Ventajas**:
  + **Flexibilidad**: Facilita la modificación y evolución del sistema.
  + **Reutilización**: Los componentes pueden ser reutilizados en diferentes contextos.
  + **Mantenimiento y evolución facilitados**: Cambios en un componente tienen menos impacto en otros componentes.
* **Desventajas**:
  + Puede requerir un diseño más cuidadoso y una mayor inversión inicial.

**Usos conocidos:**

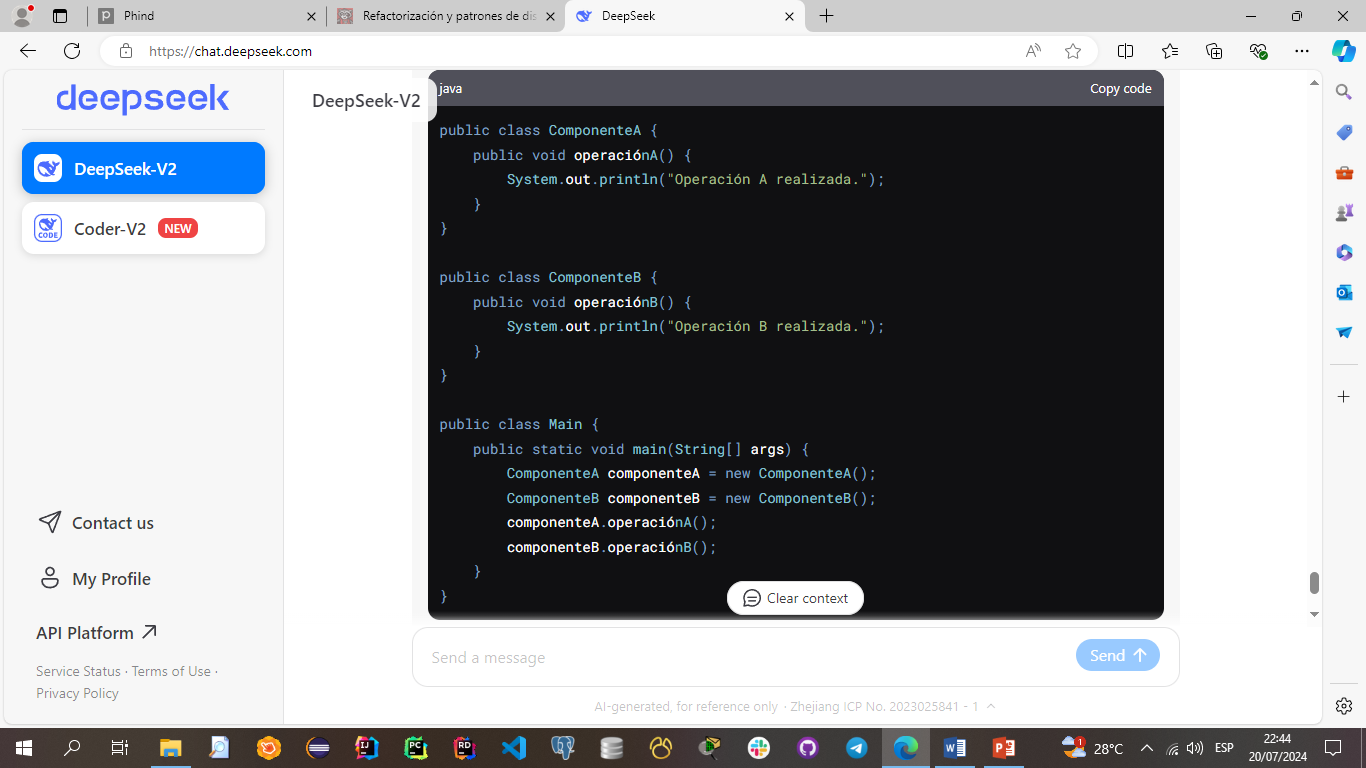
El patrón Bajo Acoplamiento es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de servicios y componentes.
* **Spring Framework**: En la inyección de dependencias y la gestión de beans.
* **.NET Framework**: En la gestión de servicios y componentes.

**Patrones Relacionados:**

* **Patrón de Diseño Singleton**: Para asegurar que una clase tenga una única instancia.
* **Patrón de Diseño Factory**: Para la creación de objetos.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.

**Código de ejemplo:**



**ALTA COHESIÓN**

**Otros Nombres:**

* Alta Cohesión
* High Cohesion

**Objetivo:**

El objetivo principal del patrón Alta Cohesión es diseñar clases o módulos de tal manera que sus responsabilidades estén estrechamente relacionadas y enfocadas. Esto promueve la claridad, la mantenibilidad y la reutilización del código.

**Aplicaciones:**

El patrón Alta Cohesión se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara asignación de responsabilidades y un enfoque modular. Ejemplos incluyen sistemas empresariales, aplicaciones de gestión, y cualquier sistema donde sea necesario mantener las responsabilidades de una clase o módulo bien definidas.

**Estructura:**

**Participantes:**

* **Componente**: Una clase o módulo con responsabilidades estrechamente relacionadas.

**Consecuencias:**

* **Ventajas**:
  + **Claridad**: Las responsabilidades de una clase o módulo están bien definidas y fáciles de entender.
  + **Mantenibilidad**: Cambios en una parte del sistema tienen menos impacto en otras partes.
  + **Reutilización**: Los componentes cohesivos pueden ser reutilizados en diferentes contextos.
* **Desventajas**:
  + Puede requerir un diseño más cuidadoso y una mayor inversión inicial.

**Usos conocidos:**

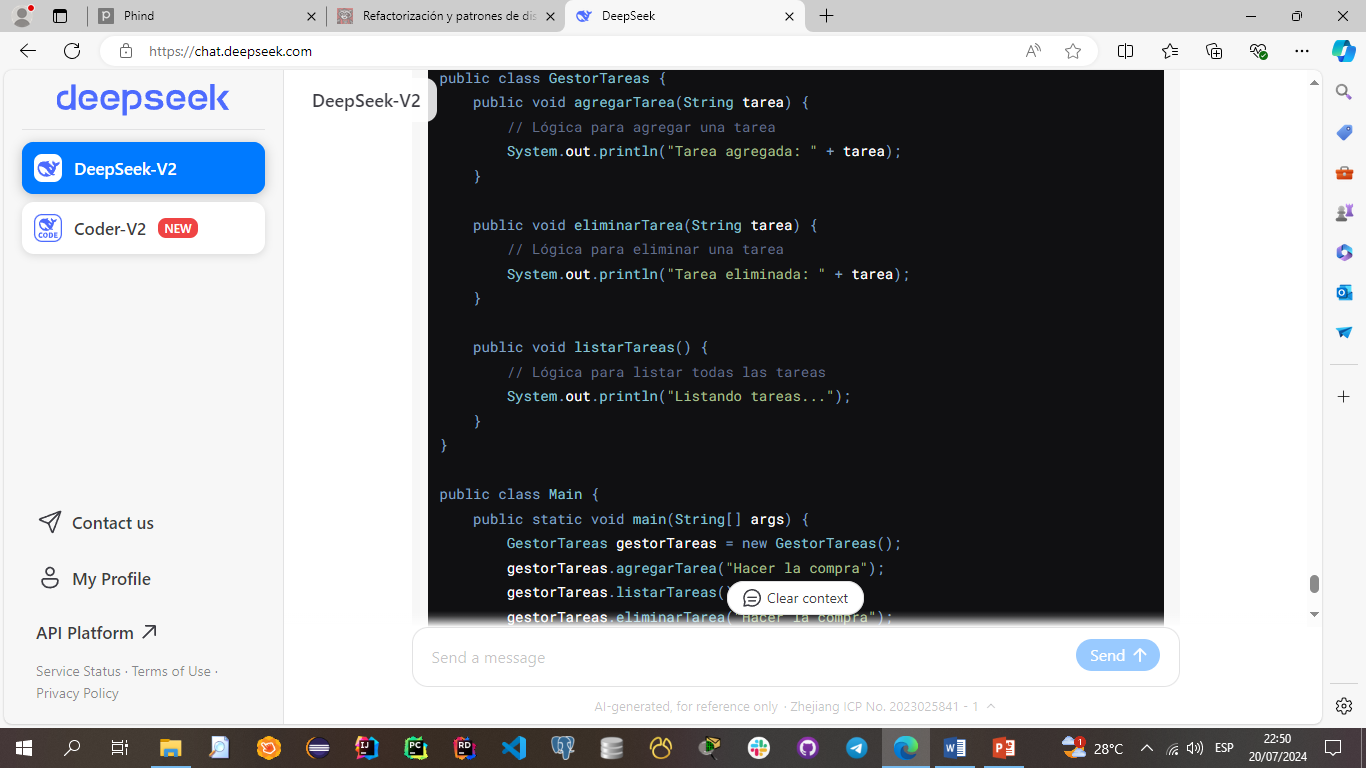
El patrón Alta Cohesión es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de servicios y componentes.
* **Spring Framework**: En la inyección de dependencias y la gestión de beans.
* **.NET Framework**: En la gestión de servicios y componentes.

**Patrones Relacionados:**

* **Patrón de Diseño Singleton**: Para asegurar que una clase tenga una única instancia.
* **Patrón de Diseño Factory**: Para la creación de objetos.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.

**Código de ejemplo:**



**POLIMORFISMO**

**Otros Nombres:**

* Polimorfismo
* Polymorphism

**Objetivo:**

El objetivo principal del patrón Polimorfismo es permitir que objetos de diferentes clases puedan ser tratados de manera uniforme a través de una interfaz común. Esto promueve la flexibilidad, la extensibilidad y la reutilización del código.

**Aplicaciones:**

El patrón Polimorfismo se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara separación de responsabilidades y una fácil extensibilidad. Ejemplos incluyen sistemas empresariales, aplicaciones de gráficos, y cualquier sistema donde sea necesario tratar objetos de diferentes clases de manera uniforme.

**Estructura:**

**Participantes:**

* **Interfaz**: La interfaz o clase abstracta que define la operación.
* **ClaseA**: Una clase que implementa la interfaz y proporciona una implementación específica de la operación.
* **ClaseB**: Otra clase que implementa la interfaz y proporciona una implementación específica de la operación.

**Consecuencias:**

* **Ventajas**:
  + **Flexibilidad**: Permite tratar objetos de diferentes clases de manera uniforme.
  + **Extensibilidad**: Facilita la adición de nuevas clases que implementen la interfaz.
  + **Reutilización**: Los objetos pueden ser reutilizados en diferentes contextos.
* **Desventajas**:
  + Puede introducir una capa adicional de complejidad si no se gestiona adecuadamente.

**Usos conocidos:**

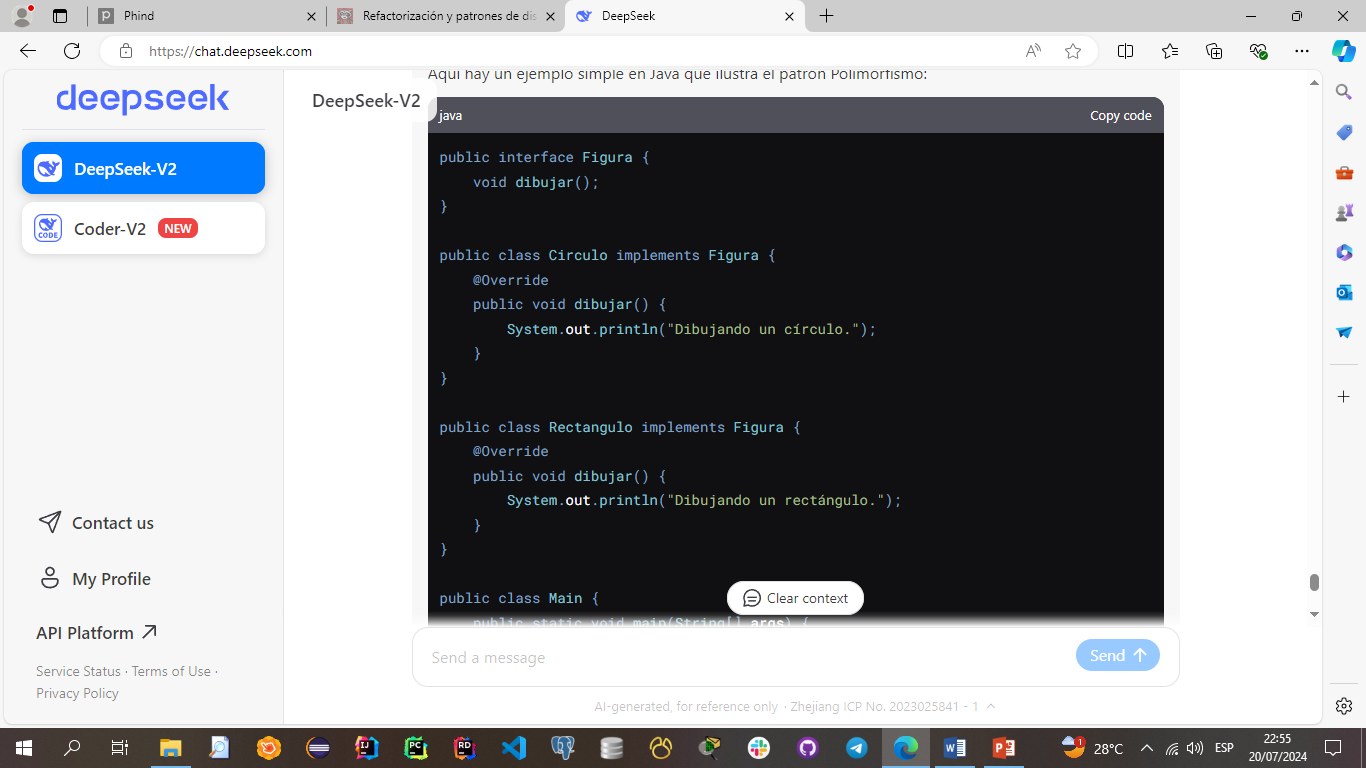
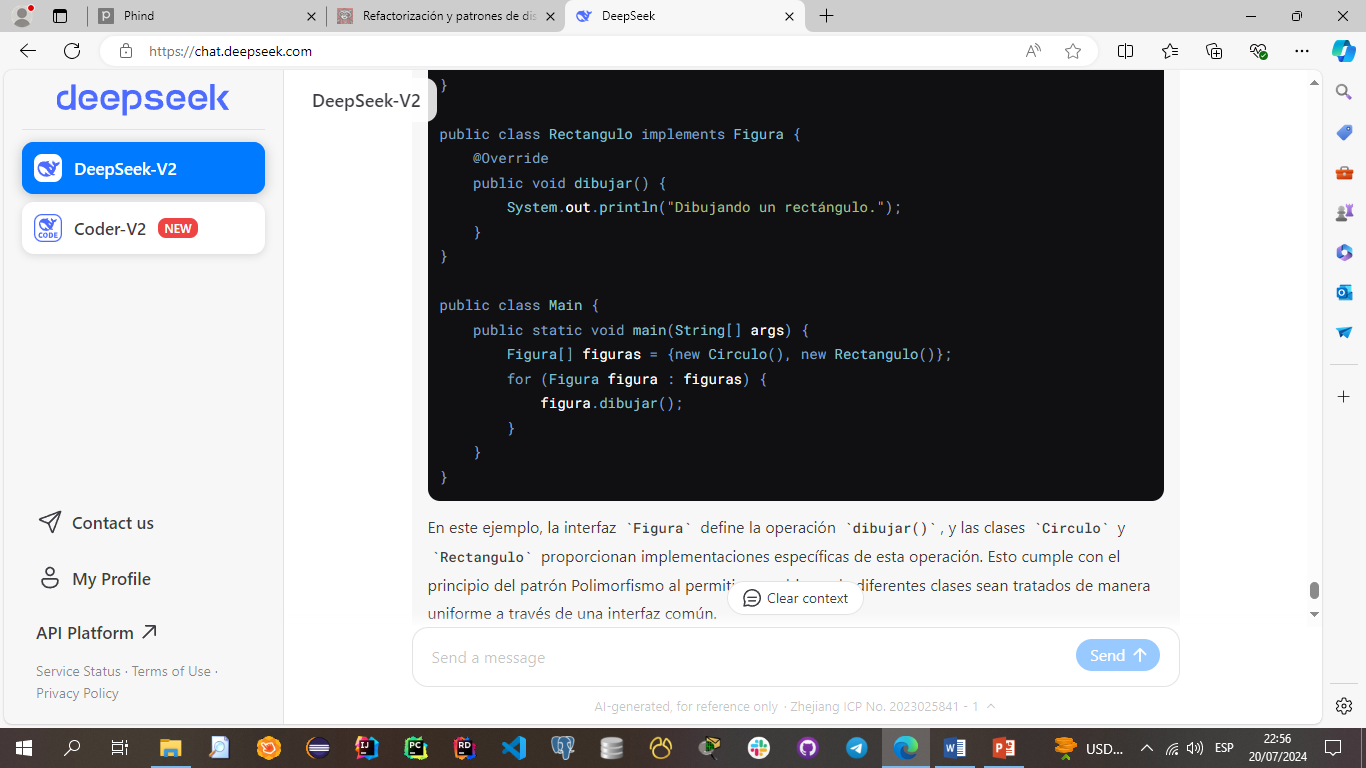
El patrón Polimorfismo es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de servicios y componentes.
* **Spring Framework**: En la inyección de dependencias y la gestión de beans.
* **.NET Framework**: En la gestión de servicios y componentes.

**Patrones Relacionados:**

* **Patrón de Diseño Factory**: Para la creación de objetos.
* **Patrón de Diseño Strategy**: Para encapsular algoritmos y permitir su intercambio.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.

**Código de ejemplo:**



**FABRICACIÓN PURA**

**Otros Nombres:**

* Fabricación Pura
* Pure Fabrication

**Objetivo:**

El objetivo principal del patrón Fabricación Pura es introducir una clase que no tiene una contraparte en el dominio del problema, pero que ayuda a mejorar la cohesión, reducir el acoplamiento y facilitar la reutilización del código. Esto promueve un diseño más limpio y mantenible.

**Aplicaciones:**

El patrón Fabricación Pura se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara separación de responsabilidades y un diseño flexible. Ejemplos incluyen sistemas empresariales, aplicaciones de gestión, y cualquier sistema donde sea necesario introducir clases adicionales para mejorar la estructura del sistema.

**Estructura:**

**Participantes:**

* **Servicio**: La clase que no representa un concepto del dominio del problema, pero que ayuda a mejorar la estructura del sistema.

**Consecuencias:**

* **Ventajas**:
  + **Mejora de la cohesión**: La clase puede agrupar responsabilidades relacionadas.
  + **Reducción del acoplamiento**: La clase puede servir como intermediario entre otras clases.
  + **Facilita la reutilización**: La clase puede ser reutilizada en diferentes contextos.
* **Desventajas**:
  + Puede introducir una capa adicional de complejidad si no se gestiona adecuadamente.

**Usos conocidos:**

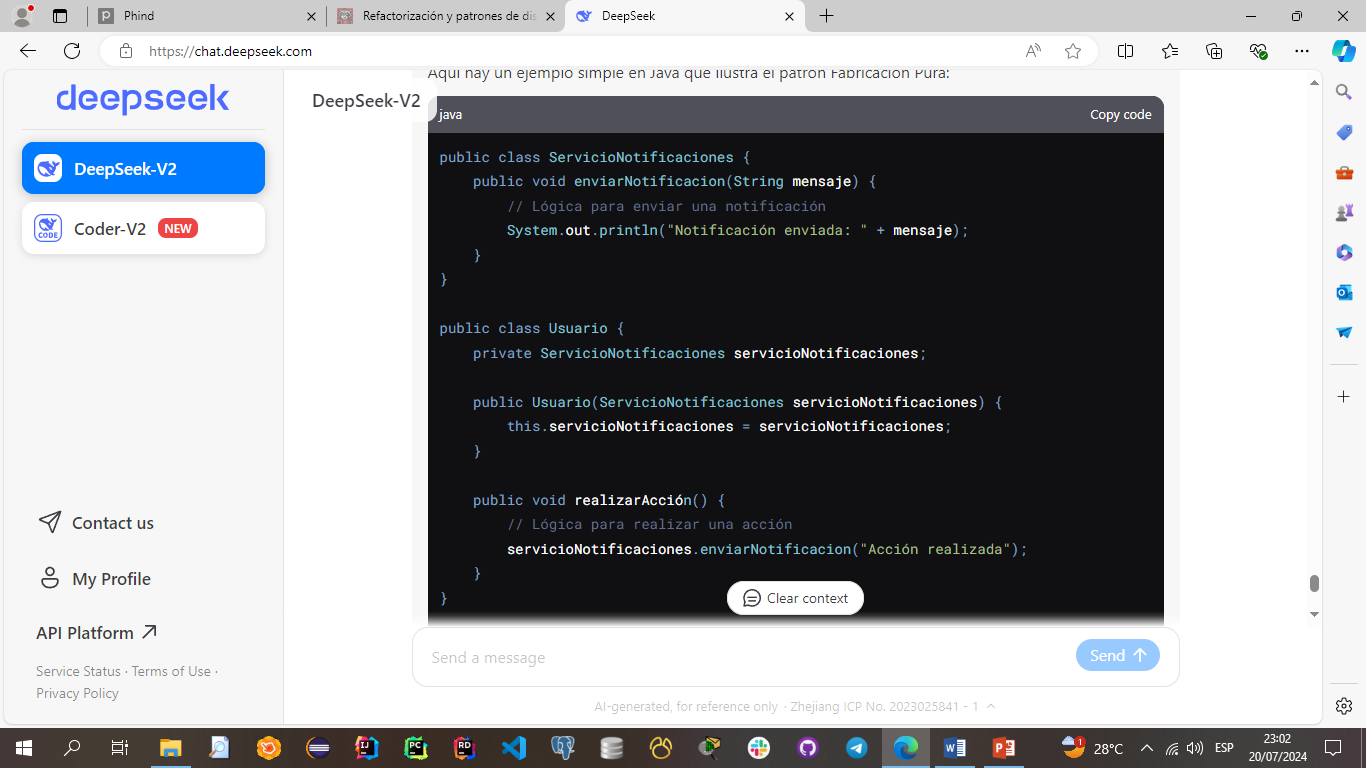
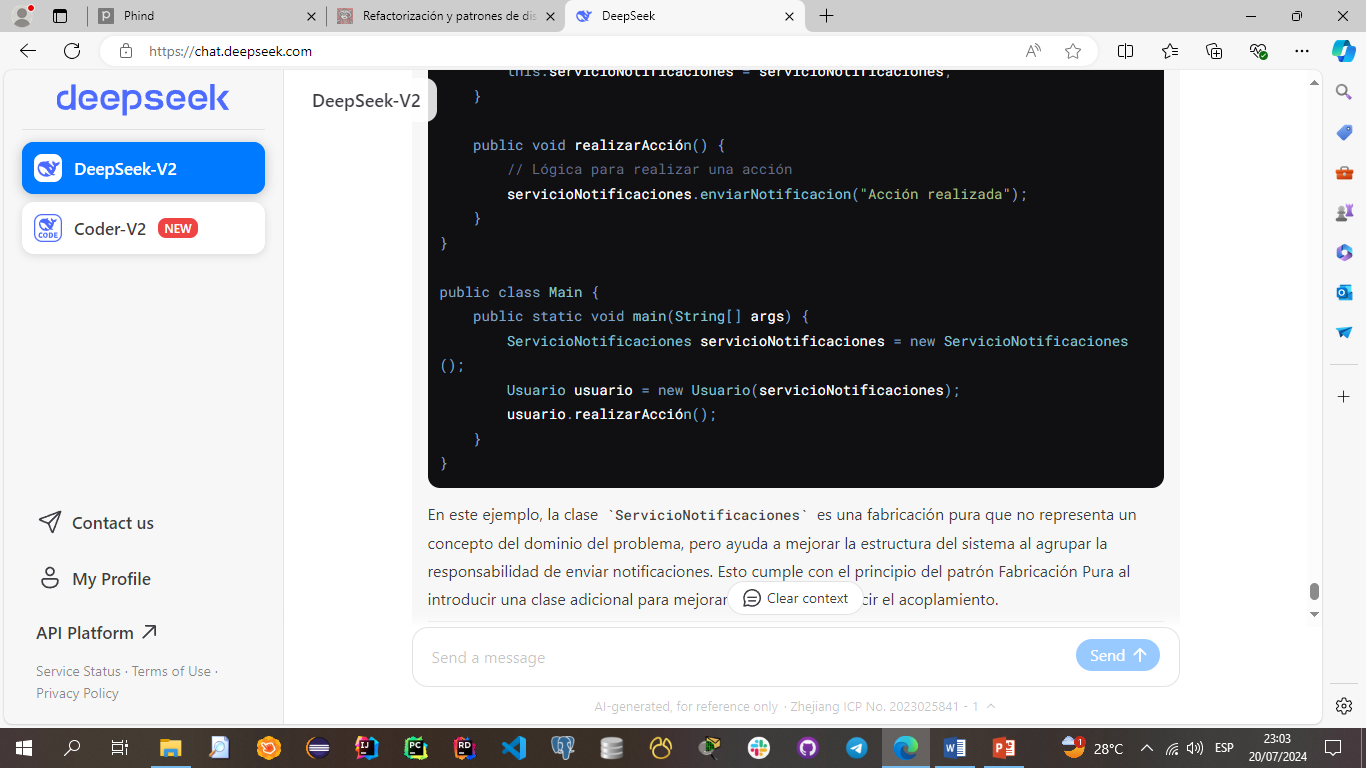
El patrón Fabricación Pura es utilizado en muchos frameworks y sistemas de software, incluyendo:

* **Java EE**: En la gestión de servicios y componentes.
* **Spring Framework**: En la inyección de dependencias y la gestión de beans.
* **.NET Framework**: En la gestión de servicios y componentes.

**Patrones Relacionados:**

* **Patrón de Diseño Singleton**: Para asegurar que una clase tenga una única instancia.
* **Patrón de Diseño Factory**: Para la creación de objetos.
* **Patrón de Diseño Observer**: Para notificar cambios a otros objetos.

**Código de ejemplo:**



**NO HABLES CON EXTRAÑOS**

**Otros Nombres:**

**Objetivo:**

El objetivo principal del patrón "No Hables con Extraños" es reducir las dependencias directas entre clases, promoviendo la comunicación a través de intermediarios. Esto ayuda a mantener un bajo acoplamiento y una alta cohesión, lo que facilita la reutilización y el mantenimiento del código.

**Aplicaciones:**

El patrón "No Hables con Extraños" se utiliza en diversos sistemas de software, especialmente en aquellos que requieren una clara separación de responsabilidades y un diseño modular. Ejemplos incluyen sistemas empresariales, aplicaciones de gestión, y cualquier sistema donde sea necesario minimizar las dependencias directas entre clases.

**Estructura:**

**Participantes:**

* **ClaseA**: Una clase que realiza una operación y se comunica a través de un intermediario.
* **Intermediario**: La clase que actúa como intermediario entre ClaseA y ClaseB.
* **ClaseB**: Otra clase que realiza una operación y se comunica a través de un intermediario.

**Consecuencias:**

* **Ventajas**:
  + **Reducción del acoplamiento**: Minimiza las dependencias directas entre clases.
  + **Mejora de la encapsulación**: Las clases pueden mantener sus detalles de implementación ocultos.
  + **Facilita la reutilización**: Las clases pueden ser reutilizadas en diferentes contextos.
* **Desventajas**:
  + Puede introducir una capa adicional de complejidad si no se gestiona adecuadamente.

**Usos conocidos:**

**Patrones Relacionados:**

**Código de ejemplo:**

