Resumen Pilares

Encapsulamiento:
Significa ocultar los detalles internos de un objeto y exponer solo una interfaz para interactuar con él. Esto se logra mediante el uso de modificadores de acceso (como público, privado y protegido) para controlar el acceso a los atributos y métodos de un objeto. El encapsulamiento ayuda a mejorar la seguridad, el mantenimiento y la reutilización del código.

Herencia:
Permite la creación de jerarquías de clases, donde una clase derivada (hija) puede heredar atributos y comportamientos de una clase base (padre). La herencia promueve la reutilización de código, evita la duplicación y facilita la extensibilidad del sistema.

Polimorfismo:
Se refiere a la capacidad de un objeto para presentar múltiples formas. El polimorfismo permite que objetos de diferentes clases respondan de manera diferente a la misma llamada de método. Esto se logra mediante el uso de la herencia y la implementación de métodos con la misma firma en diferentes clases. El polimorfismo facilita la flexibilidad y la modularidad del código.

Abstracción:
Consiste en representar entidades del mundo real como objetos con características y comportamientos relevantes para el sistema. La abstracción ayuda a simplificar y gestionar la complejidad al enfocarse en los aspectos esenciales de un objeto y ocultar los detalles irrelevantes. También permite crear clases abstractas e interfaces para definir contratos y especificaciones que las clases concretas deben cumplir.

Estos pilares son fundamentales en la programación orientada a objetos y proporcionan un enfoque estructurado y eficiente para el desarrollo de software.

Resumen principios SOLID

Los principios *SOLID* son un conjunto de principios de diseño de software que promueven la creación de código limpio, flexible y mantenible. A continuación, se presentan los principios SOLID con su nombre y significado:

Principio de Responsabilidad Única (SRP - Single Responsibility Principle):
Un objeto o clase debería tener una única responsabilidad. Esto significa que una clase debe tener solo una razón para cambiar, lo que facilita su comprensión, reutilización y mantenimiento.

Principio de Abierto/Cerrado (OCP - Open/Closed Principle):
Las entidades de software deben estar abiertas para su extensión pero cerradas para su modificación. Esto implica que el código debe ser fácilmente extensible mediante la adición de nuevas funcionalidades, sin necesidad de modificar el código existente.

Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):
Los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin afectar la funcionalidad del programa. Esto asegura que las clases derivadas sean compatibles con las clases base y evita resultados inesperados en el código.

Principio de Segregación de Interfaces (ISP - Interface Segregation Principle): Los clientes no deben depender de interfaces que no utilizan. Este principio propone dividir interfaces grandes en interfaces más pequeñas y cohesivas, específicas para cada cliente, evitando la implementación de métodos innecesarios.

Principio de Inversión de Dependencia (DIP - Dependency Inversion Principle):
Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Este principio establece que las dependencias deben ser abstracciones (interfaces o clases abstractas) en lugar de implementaciones concretas, lo que permite una mayor flexibilidad y facilidad para realizar cambios.

Estos principios SOLID ofrecen directrices importantes para desarrollar software bien estructurado y modular, lo que facilita la mantenibilidad, la extensibilidad y la reutilización del código.

Plus “Tell, Don’t Ask”

El principio de *"Tell, Don't Ask"* (en español, "Decir, No Preguntar") es un principio de diseño de software que promueve la encapsulación y la responsabilidad en la interacción entre objetos.

Este principio establece que en lugar de preguntar por el estado interno de un objeto para tomar decisiones y realizar acciones, se debe "decirle" al objeto qué hacer y permitir que él mismo se encargue de realizar las acciones necesarias. En otras palabras, en lugar de solicitar información y luego tomar decisiones basadas en esa información, se debe delegar la responsabilidad al objeto correspondiente para que realice las acciones necesarias en función de su estado interno.

Al seguir este principio, se evita exponer el estado interno de un objeto y se promueve el encapsulamiento y la cohesión. Esto mejora la modularidad, el mantenimiento y la extensibilidad del código, ya que cada objeto es responsable de realizar sus propias acciones y no se violan los principios de responsabilidad única y ocultamiento de información. Además, facilita la adición de nuevas funcionalidades y reduce el acoplamiento entre objetos.

En resumen, el principio de "Tell, Don't Ask" sugiere que en lugar de preguntar a un objeto sobre su estado para tomar decisiones, se le debe decir qué hacer y permitir que el objeto se encargue de realizar las acciones correspondientes en base a su estado interno.

Plus “Demeter law”

La Regla de Demeter, también conocida como el Principio de No Hables con Extraños (Law of Demeter o LoD), es un principio de diseño de software que establece restricciones en la interacción entre objetos para reducir el acoplamiento y promover la encapsulación.

La regla de Demeter se basa en la idea de que un objeto debe tener un conocimiento limitado sobre otros objetos y solo debe interactuar con aquellos que sean directamente necesarios para llevar a cabo su trabajo. En resumen, un objeto solo debe hablar con sus "vecinos" inmediatos y no debe tener conocimiento detallado sobre la estructura interna de otros objetos.

La regla de Demeter se puede resumir en los siguientes puntos:

Cada método debe tener conocimiento limitado sobre los objetos con los que interactúa directamente (sus propios atributos, parámetros de entrada, objetos creados internamente, etc.).
Evitar la cadena de llamadas a través de múltiples objetos. Un objeto solo debería hablar directamente con sus "amigos" más cercanos.
No acceder a los métodos y propiedades de objetos que se obtienen indirectamente a través de otros objetos.
El objetivo de la regla de Demeter es reducir la dependencia y el acoplamiento entre objetos, lo que hace que el código sea más flexible, mantenible y reutilizable. Además, favorece el principio de encapsulamiento al ocultar detalles internos y evitar el acceso directo a la estructura interna de otros objetos.

Aplicar la regla de Demeter ayuda a mejorar la modularidad y facilita los cambios en el diseño, ya que los objetos se comunican de manera más independiente y las modificaciones en un objeto no afectan a otros objetos de forma inesperada.

Plus “GRASP”

GRASP (General Responsibility Assignment Software Patterns) es un conjunto de patrones de asignación de responsabilidades que se utilizan en el diseño de software orientado a objetos. Estos patrones ayudan a determinar cómo asignar las responsabilidades entre las

clases y objetos de un sistema de manera cohesiva y de acuerdo con los principios de diseño sólidos.

Los patrones GRASP se centran en identificar las responsabilidades adecuadas para cada clase u objeto en el diseño de software. Al seguir estos patrones, se busca lograr un diseño de software flexible, mantenible y de calidad.

A continuación, se mencionan algunos de los patrones GRASP más conocidos:

Expert (Experto):
Asigna una responsabilidad a la clase que tiene la información necesaria para cumplir esa responsabilidad. Se busca asignar la responsabilidad a la clase que tenga la mayor cantidad de conocimiento y experiencia en relación con esa responsabilidad.

Creator (Creador):
Asigna la responsabilidad de crear instancias de objetos a la clase que tiene la información necesaria para configurar adecuadamente esos objetos. Se busca evitar la creación de objetos en clases que no tienen la información necesaria para inicializarlos correctamente.

Controller (Controlador):
Asigna la responsabilidad de coordinar y controlar las operaciones del sistema a una clase que actúa como punto central de control. Se busca evitar la dependencia directa entre objetos y fomentar la centralización del control en una clase específica.

Low Coupling (Bajo Acoplamiento):
Busca reducir el acoplamiento entre las clases asignando responsabilidades de manera que las interacciones entre ellas sean mínimas. Se busca lograr un diseño flexible y fácil de mantener.

High Cohesion (Alta Cohesión):
Busca que las responsabilidades asignadas a una clase estén relacionadas y sean cohesivas. Se busca lograr un diseño en el que las clases tengan una única responsabilidad bien definida.

Estos son solo algunos ejemplos de los patrones GRASP. Cada patrón proporciona una guía para asignar responsabilidades en el diseño de software, promoviendo una estructura clara y modular. Los patrones GRASP se utilizan como herramientas para ayudar en el proceso de diseño y facilitar la creación de sistemas orientados a objetos de alta calidad.
[1]

[1] No me doy mas y necesito un cafe