David Santiago Álvarez Ríos-20231020122

1. Principio de Responsabilidad Única (SRP - Single Responsibility Principle):

- Este principio sostiene que una clase debe tener una única razón para cambiar, es decir, debe tener una única responsabilidad o función en el sistema. Al dividir el código en clases más pequeñas y especializadas, se mejora la cohesión y se facilita el mantenimiento y la comprensión del código.

2. Principio de Abierto/Cerrado (OCP - Open/Closed Principle):

- El OCP establece que una clase debe estar abierta para su extensión, permitiendo agregar nuevas funcionalidades mediante la creación de nuevas clases, pero cerrada para su modificación, evitando alterar el código existente. Esto se logra mediante el uso de interfaces, herencia y polimorfismo.

3. Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):

- Este principio se refiere a la relación de herencia entre las clases. Si una clase A es una subclase de B, se debe poder usar objetos de la clase B en cualquier lugar donde se espera un objeto de la clase A, sin cambiar el comportamiento esperado del programa.

4. Principio de Segregación de Interfaces (ISP - Interface Segregation Principle):

- El ISP indica que una clase no debe ser forzada a implementar interfaces que no utiliza. En lugar de tener interfaces grandes y genéricas, se deben preferir interfaces más pequeñas y específicas. Esto evita que las clases dependan de métodos que no necesitan, mejorando la cohesión y reduciendo la complejidad.

5. Principio de Inversión de Dependencia (DIP - Dependency Inversion Principle):

- El DIP propone que las dependencias en un sistema deben invertirse, de manera que las clases de alto nivel no dependan de clases de bajo nivel, sino de abstracciones compartidas. Esto se logra mediante la introducción de interfaces o clases abstractas que ambas dependan, facilitando la flexibilidad y permitiendo cambiar implementaciones sin afectar clases de alto nivel.

Al aplicar estos principios SOLID, los desarrolladores pueden construir sistemas más modulares, flexibles y mantenibles. Cada principio aborda aspectos específicos del diseño de software y, cuando se aplican en conjunto, contribuyen a la creación de un código más robusto y fácil de entender, adaptar y extender a medida que los requisitos cambian.

Además de estos principios SOLID, existen otros principios y prácticas que los desarrolladores suelen considerar para mejorar la calidad del software y facilitar su mantenimiento. Aquí hay cinco principios adicionales:

1. Principio de KISS (Keep It Simple, Stupid):

- KISS aboga por la simplicidad en el diseño y desarrollo del software. La idea es que la solución más simple que cumple con los requisitos es generalmente la mejor. Evitar la complejidad innecesaria facilita la comprensión y el mantenimiento del código.

2. Principio DRY (Don't Repeat Yourself):

- DRY insta a evitar la duplicación de código. Si hay una funcionalidad común que se repite en diferentes partes del código, debería refactorizarse en una única ubicación. Esto reduce la redundancia, facilita las actualizaciones y mejora la coherencia.

3. Principio YAGNI (You Aren't Gonna Need It):

- YAGNI sugiere que no se deben agregar características al software hasta que sean realmente necesarias. Evitar la implementación prematura de funcionalidades no solo ahorra tiempo, sino que también ayuda a mantener el código más limpio y enfocado en los requisitos actuales.

4. Principio de Composición sobre Herencia:

- Este principio aboga por favorecer la composición de objetos sobre la herencia de clases cuando sea posible. La composición permite construir objetos complejos combinando partes más simples, lo que suele ser más flexible y fácil de entender que depender en exceso de la herencia.

5. Principio de Separación de Preocupaciones (Separation of Concerns):

- Este principio sugiere dividir un sistema en módulos o componentes, cada uno de los cuales se centra en una tarea o preocupación específica. Al hacerlo, se mejora la modularidad y la capacidad de cambio, ya que las modificaciones en una preocupación no afectan directamente a las demás.

Estos principios, junto con los principios SOLID, forman un conjunto de pautas valiosas para el diseño de software eficiente y mantenible. Al aplicar estas prácticas, los desarrolladores pueden crear sistemas más robustos, escalables y fáciles de mantener a lo largo del tiempo.