**Introducción a SOLID**

**SOLID** es un acrónimo que representa cinco principios básicos de diseño de software orientado a objetos. Estos principios fueron introducidos por Robert C. Martin y son fundamentales para crear sistemas de software robustos, flexibles y mantenibles. Aplicar estos principios ayuda a evitar el acoplamiento excesivo y mejora la cohesión del código, facilitando su evolución y adaptación a cambios futuros

**Principios SOLID**

1. **Single Responsibility Principle (SRP) - Principio de Responsabilidad Única**:
   * **Definición General**: Cada clase debe tener una única responsabilidad, es decir, solo una razón para cambiar. Esto significa que una clase debe centrarse en hacer una sola cosa y hacerlo bien.
   * **Explicación Teórica**: Este principio se enfoca en la cohesión de las clases. Una clase debe representar una sola parte del sistema y no debe intentar hacer demasiadas cosas a la vez. Al mantener las clases enfocadas en una única responsabilidad, se facilita su comprensión, prueba y mantenimiento.
   * **Explicación en Java**: En Java, esto se traduce en diseñar clases que tienen un solo propósito. Por ejemplo, una clase User debe gestionar solo los datos del usuario, mientras que una clase UserService debe manejar la lógica de negocio relacionada con los usuarios.

Texto

Descripción generada automáticamente

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

* **Definición General**: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión pero cerradas para modificación.
* **Explicación Teórica**: Este principio promueve la extensión del comportamiento de una clase sin modificar su código fuente existente. Se logra mediante el uso de la herencia y la composición. Esto permite añadir nuevas funcionalidades con el mínimo riesgo de introducir errores en el código existente.
* **Explicación en Java**: En Java, esto se puede lograr creando clases abstractas o interfaces que las clases concretas pueden implementar o extender.

Texto

Descripción generada automáticamente

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

* **Definición General**: Los objetos de una clase derivada deben poder reemplazar a los objetos de la clase base sin afectar el comportamiento del programa.
* **Explicación Teórica**: Este programa. Esto se logra garantizando que las clases derivadas mantengan las promesas contractuales de la clase base.
* **Explicación en Java**: En Java, las subclases deben sobrescribir los métodos de las clases base de manera que no rompan la lógica del programa. Es crucial que las subclases no introduzcan excepciones inesperadas o cambien el comportamiento fundamental de los métodos heredados.

Texto

Descripción generada automáticamente

En este caso, Ostrich viola el principio de sustitución de Liskov porque no puede volar, lo que no es consistente con el comportamiento esperado de la clase base Bird.

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

* **Definición General**: Los clientes no deben verse obligados a depender de interfaces que no utilizan.
* **Explicación Teórica**: Este principio sugiere que es mejor tener muchas interfaces pequeñas y específicas que una única interfaz grande y general. Las interfaces pequeñas aseguran que los clientes solo conozcan los métodos que realmente usan, evitando la implementación de métodos no deseados.
* **Explicación en Java**: En Java, se debe diseñar interfaces específicas en lugar de generalizar todas las funcionalidades en una sola interfaz. Esto mejora la modularidad y flexibilidad del código.

Texto

Descripción generada automáticamente

1. **Dependency Inversion Principle (DIP) - Principio de Inversión de Dependencias**:

* **Definición General**: Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
* **Explicación Teórica**: Este principio sugiere que en lugar de que los módulos de alto nivel dependan directamente de módulos de bajo nivel, ambos deben depender de abstracciones (interfaces o clases abstractas). Esto reduce el acoplamiento entre módulos y mejora la flexibilidad y reusabilidad del código.
* **Explicación en Java**: En Java, se implementa creando interfaces o clases abstractas que definen el comportamiento y luego haciendo que las clases concretas las implementen.

Texto

Descripción generada automáticamente