**INTRODUCTION TO THE SOLIED PRINCIPLES**

**Overview of the solid principles**

Solid tiene cinco principios y por sus siglas son:

Single Responsability Princple

Open/ closed principle

Liskov Substtitution Principle

Interface Segregation Principle

Interface Segregation Prnciple

Dependecy Inversion Principle

Algunos argumentan que estos principios no son importantes, sin embargo, seguirlos puede ayudar a garantizar un software de calidad en la perspectiva de los desarrolladores.

**Design Problem**

Generar un programa donde se debe producir un laberinto, los cuales se debe poder imprimir usando caracteres ASCII o dibujarlos en una imagen.

**Single responsability principle**

Este principio está relacionado con el principio mas general de Cohesión que dice que las responsabilidades de cualquier componente (método, clase, subsistema, etc.) deben estar estrechamente alineadas y enfocadas en un solo propósito.

También se relaciona con los principios de Localización de decisiones de diseño y encapsulamiento. Este principio puede ayudar aumentando la reutilización y mantenibilidad, reducir la complejidad, aunque el número de clases podría aumentar.

**Open / Closed Principle**

Una clase es abierto si todavía está disponible para la extensión, una clase es cerrado si está disponible para que lo use otra clase y por lo tanto no debe modificarse.

Un sistema de clases está abierto a la extensión y cerrado a la modificación si

-Los métodos públicos se declaran utilizando interfaces o clases abstractas

- Implementaciones o clases concretas heredar las declaraciones de métodos públicos de las interfaces, clases abstractas o clases virtuales puras.

**Interfaces, Abstract clases, Pure virtual classes**

-Revisión: la herencia permite que una especialización reutilice la generalización: miembros de datos, Declaración de métodos, Definiciones de métodos.

-Una interfaz es como una clase base, pero solo permite declaraciones de método; No permite que los miembros de datos sean declarados o definidos, no tiene implementación de métodos.

-En UML, los nombres de clases y métodos virtuales puros y abstractos se escriben en cursiva.

-Una clase virtual abstracta y pura (C++) puede incluir miembros de datos y algunas implementaciones de métodos

- El principio Abierto/ Cerrado alienta a los desarrolladores: usar interfaces clases abstractas y clases virtuales para declarar miembros de datos públicos

**Open/ Closed Principle**

Formas de lograr el principio abierto / cerrado: Herencia, Agregación y Parametrización.

Seguir el principio abierto/cerrado puede ayudar a los desarrolladores: Reducir la complejidad y aumentar la extensibilidad.

**Interface segregation principle**

Una interfaz es una “ventana” o “portal” a la funcionalidad de un componente, representa los métodos públicos de un componente. Java admite interfaces directamente.

Los métodos públicos de un componente se pueden agrupar por propósito o responsabilidad tal como se capturan y declaran en interfaces o clases abstractas.

Seguir este principio ayuda al desarrollador a: Reducir la complejidad, Aumentar la extensibilidad, Aumente la reutilización, Aumentar la mantenibilidad.

**Dependency inversión principle**

Se organiza el sistema en capas como: bibliotecas reutilizables o marcos estas serán más abstractas y otras estará orientadas a detalles.

Los componentes de las capas abstractas no deben depender de los componentes de las capas de detalle, las abstracciones no deben depender de los detalles.

Este principio se aplica cuando:

-Las abstracciones no deben depender de los detalles.

-Los módulos de alto nivel no deben depender de los módulos de bajo nivel.

-Tanto los módulos de bajo nivel como los de alto nivel deben depender de abstracciones.

Ayuda a los desarrolladores: aumentando la reutilización y la mantenibilidad.

**Software engineering goals**

Software engineers aim to build quality products on time and within budget, some desirable Qualities : understability, efficiency,reliability, reusability ,estensibility.

**Common Paradigms**

Object Orientation (OO) Logic Programming (LP) Aspect Orientation (AO)

Genetic Programming (GP) Functional programming (FP) Structures program (SP)

Java support OO and FP, as well as AO, LP, GP when used with appropriate algorithms or libraries.

**Problem Background**

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees. In some cases, there are conflicting definitions within the same paradigm.

Core Problem