OOP HW18 Software Engineering Principles

DATE: 01th February 2021

VILLEGAS ESTRELLA SALMA ABIGAIL

INTRODUCTION TO THE SOLID PRINCIPLES

SOLID is a mnemonic acronym for five principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle

• Some argue that these are the" first five" principles

• That claim has not been justified or widely accepted

• However, whether they are the "first five" principles is not very important

• Following these principles can help ensure quality software, primarily from a developers' perspective

SINGLE RESPONSIBILITY PRINCIPLE

Every class should be responsible for a single part of the system's functionality, a class's responsibility should be entirely encapsulated by the class, a class's properties should be narrowly aligned with that responsibility

This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose

Also related to the principles of: Localization of design decisions, Encapsulation

OPEN/CLOSED PRINCIPLE

Software entities (e.g., classes, generics) should be open for extension but closed to modification

A class is open if it is still available for extension, a class is closed

Ways to achieve the open/closed principle

• Inheritance: Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes. Java does not support multiple inheritance, so a class can have multiple base classes

• Aggregation: Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically

• Parameterization: Use a generic to capture a template solution and instantiate it with the specific data types

INTERFACE SEGREGATION PRINCIPLE

An interface is a" window" or "portal" into the functionality of a component, an interface represents public methods of a component, an interface doesn't have to declare all of the possible public methods of a component; a component can have many interfaces, Java does support interfaces directly.

DEPENDENCY INVERSION PRINCIPLE

Organize the system into layers: some layers, like reusable libraries or frameworks will be more abstract or policy-setting layer, others will be detail oriented

Components from the abstract layers should not depend on components from the detail layers; instead, they should depend on abstractions that the detailed components implement, abstractions should not depend on details

Implementation details should depend on abstractions

THREE CORE PRINCIPLES

Abstraction

The principle of abstraction can improve understandability, testability, maintainability, and reusability. It can also allow developers to track modules more effectively, as it will highlight weaknesses with localization of design decisions, unnecessary coupling, and weak linkage.
For each component, have a clear and unambiguous statement explicit about the component's accessible features or functionality. The features and functionality exhibited should be no more or less than what other components may need or depend on.

Encapsulation

Ensures that a component's private implementation details are isolated so that they cannot be viewed or modified by other components. This will lead to better testability, maintainability and reliability. It will also help to clearly separate concerns and avoid random pairing.

Modularity

Appears in a software system when it is loosely coupled (the extent to which components depend on each other) and cohesive (the extent to which properties of a component that are related to the main responsibility of the component) components that separate each important or mutable design decision within a component and ensure that ideas are closely related tight as possible.