INTRODUCTION TO THE SOLID PRINCIPLES

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 if it is available for use by other class, and therefore should not be modified

A system of classes is open for extension and closed for modification, if

Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes (in Java)

INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES

Review: Inheritance allows a specialization (a derived class) to re-use the generalization's (a base class's):

Data members

Method declarations

Method definitions (i.e., their implementations)

OPEN/CLOSED PRINCIPLE

Ways to achieve the open/closed principle

Inheritance

Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual clases

Aggregation

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

Parmeterization

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

Following the Open/Closed Principle can help developers

Reduce complexity by reducing coupling (dependencies among components)

DEPENDENCY INVERSION PRINCIPLE

How to apply the Dependency Inversion Principle

Abstractions should not depend on details

High-level modules should not depend on low-level modules

Both low-level and high-level modules should depend on abstractions

“Program to the abstraction”

Following the Dependency Inversion Principle helps Developers

Increase Reusability

Increase Maintainability

SOFTWARE ENGINEERING GOALS

Software engineers aim to build quality products on time and within budget
Some Desirable Qualities:

understandability

testability

maintainability

efficiency

eliability

security

extensibility

openness

interoperability • reusability

PROBLEM BACKGROUND

Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees However, the concepts and definitions of these principles differ across

paradigms.In some cases, there are conflicting definitions within the same paradigm • There are also many other proposed principles that overlap and break up the ideas differently.

CONTRIBUTIONS OF THIS INITIAL PAPER

The purpose of this paper is NOT to reinvent the concepts of modularity, abstraction, or encapsulation.Instead, it is to stimulate discussion about the unification of existing ideas.

BEST PRACTICES, PATTERNS, AND IDIOMS

Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level. Patterns exemplify principles, by providing proven solutions to recurring problems in specific contexts. Idioms are techniques or solution for expressing a certain algorithm or

data structure in a specific programming language, in a way that is consistent with certain principles.

OBSERVATIONS RELATIVE TO MODULARITY

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways.Two concepts that can help achieve this desirable characteristic:

Coupling: the degree to which components depend on each other

Cohesion: the degree to which the properties of a component relate to the component's primary responsibility

PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY

Practices and Criteria:

Localization of design decisions

Low Coupling

High Cohesion

Modular Reasoning