**INTRODUCTION TO THE SOLID PRINCIPLES**

Dependency Inversion Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation

Single Responsibility Principle

**OVERVIEW**

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

**DESING PROBLEM**

• You’ve been contracted to build a Maze Generator

• The program must produce mazes

• The program must be able to print the mazes using ASCII character or draw them in an image

**SINGLE RESPONSIBILITY PRINCIPLE**

• Core ideas

• This principle is vey closely related to the more general principle of Cohesion

• This principle is also related to the principles of:

• Localization of design decisions

• Encapsulation

• Following this principle can help

**OPEN/CLOSED PRINCIPLE**

• Core Ideas:

• Software entities

• Original definitions:

• A class is open if it is still available for extension

• Revised definitions:

• A system of classes is open for extension and closed for modification

**INTERFACES, ABSTRACT CLASSES, PURE VIRTUAL CLASSES**

• Review: Inheritance allows a specialization

• An interface is like a base class, but only allows for Method declarations• In UML, the names of abstract and pure virtual classes and methods are written in italics

• An abstract and a pure virtual class (C++) may include data members and some method implementations

• The modern Open/Closed Principle encourages developers to

• use interfaces, abstract classes, and pure virtual classes to declare public data members

**OPEN/CLOSED PRINCIPLE**

• Ways to achieve the open/closed principle

• Inheritance

• Aggregation

• Parameterization

• Following the Open/Closed Principle can help developers

**LISKOV SUBSTITUTION PRINCIPLE**• if S is a specialization of T, then an S object can be used wherever a T object is required,

e.g.,

• Strong behavioral subtyping – an S object must be able to do everything any T object

• Following the Liskov Substitution Principle can help developers

• Increase Reuse

• Increase Extensibility

• Increase Maintainability

**FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE**

• Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file

• When implementing a specialization, Widget, of some Product, ensure that

**INTERFACE SEGREGATION PRINCIPLE**

• An interface is a ”window” or “portal” into the functionality of a component

• An interface represents public methods of a component

• Following the Interface Segregation Principle, when used with other principles, can help Developers

• Reduce complexity by increasing Cohesion and reducing Coupling

• Increase extensibility

• Increase reuse

• Increate maintainability

**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

• How to apply the Dependency Inversion Principle

• Abstractions should not depend on details

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

• 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

• reliability

• security

• extensibility

• openness

• interoperability

• reusability

**COMMON PARADIGMS**• Object orientation (OO) • Aspect orientation (AO) • Functional programming (FP) • Logic programming (LP) • Genetic programming (GP) • Structured program (SP)

**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

**CORE PROBLEM**• There are no general, unifying definitions, especially for multiparadigm software development

**CONTRIBUTIONS OF THIS INITIAL PAPER**• Clarify the purpose of software-engineering (SE) principles, in general, and distinguish them from “best practices”, idioms, and patterns

• Propose drafts of paradigm-independent definitions for the MAE principles

**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

**PURPOSE OF PRINCIPLES**

• According to the Merriam-Webster and Oxford dictionaries**SOFTWARE ENGINEERING PRINCIPLES**

1) a truth or proposition that supports reasoning about the desirable characteristics of a software system

2) a rule for creating software with certain desirable characteristics

3) an aspect of software design that imparts certain desirable characteristics

**BEST PRACTICES, PATTERNS, AND IDIOMS**

• Best practices are procedures or techniques that help developers adhere to principles

• Patterns exemplify principles

• Idioms are techniques or solution for expressing a certain algorithm or data structure in a specific programming language

**OBSERVATIONS RELATIVE TO MODULARITY**

• Earlier and seminal work by David Parnas (1972, plus later works)

• Design decisions need to be identified and implemented in one place. We call this Localization of Design Decisions

• Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways

• Low Coupling and High Cohesion were first explored for SP, but have been heavily discussed for OO and AO

• The first principle of Robert Martin’s SOLID principles is a restatement of High Cohesion

• Martin Fowler and other experts believe that modularity will avoid common code smells, like Long Method, Large Class, Long Parameter List, Feature Envy, and Inappropriate Intimacy

**PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY**

Practices and Criteria:

• Localization of design decisions

• Low Coupling

• High Cohesion

• Modular Reasoning

Tradeoffs:

• Localization of design decisions and high cohesion can lead to many fine grain components, which is good for testability, extensibility, and reuse, but may hinder readability

• However, the lack modularity will compromise desirable characteristics

• Adherence or violation of the modularity principles typically affects multiple components

Paradigm Notes:

• For LP, the components are primarily predicates, rules, and facts.

• Developers must ensure every predicate represents a single idea or responsibility. In other words, every predicate should be highly cohesive.

**OBSERVATIONS RELATIVE TO ABSTRACTION**• From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others

• Creating good software abstractions is hard

• Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

**TWO COMMON PROBLEMS WITH ABSTRACTION**

• Leaky abstraction – other components end up relying on details not explicitly stated in the abstraction

• External characteristics are not defined as such

**PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION**

Practices and Criteria:

• Meaningful labels and identifiers

• Context-aware labels and identifiers

• Abstraction completeness

• Abstraction sufficiency

**OBSERVATIONS RELATIVE TO ENCAPSULATION**

• Three categories of existing definition for encapsulation:

• The bundling of data with operations

• The hiding decisions behind logical barriers

• The organization of components to minimize ripple effects

**PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION**

Practices and Criteria:

• Conceptual barriers

• Programmatic barriers

• Usage barriers

**SUMMARY**

• Clarified the purpose of software-engineering principles

• Proposed a template for documenting principles

• Proposed drafts of paradigm-independent definitions for the MAE principles

**FUTURE WORK** • Setup concrete empirical studies to explore those questions

• Explore metrics for systematically assessing quality in mixed-paradigm software system

• Investigations into other design principles beyond MAE