Solid Principles

The word SOLID is an acronym for the following principles:

- Single Responsibility Principle

- Open/closed principle

- Liskov substitution principle

- Interface separation principle

- Dependency Inversion Principle

According to many these previous principles are the first five, although this has not been confirmed. In any case, following these principles guarantees software quality.

Single Responsibility Principle (main ideas):

- Each class is responsible for one part of the system (in its functionality).

- The responsibility of the class must be encapsulated by the class.

- The properties of the class are related to that responsibility.

This whole principle is familiar with the principle of cohesion, in which it is specified that any component is related to the purpose. In addition, it relates to the localization of design decisions and encapsulation.

Open/closed principle (main ideas):

- Software entities are open for extension, but closed to modification.

Original definitions:

An open class is available for extension.

A class is closed if it is to be used by another class.

Revision:

Inheritance allows specialization and reuse generalization.

Method declarations:

An interface is like a base class, but only allows method declarations.

It does not allow data members to be declared.

Ways to achieve the open/closed principle:

- Inheritance: move public methods to their own abstractions, public methods of a class can be grouped into multiple abstractions.

Java does not support multiple inheritance, so a class can have multiple base classes.

- Aggregation: Encapsulate behaviors in objects and allow them to change dynamically.

This technique has been incorporated into the strategy pattern.

- Parameterization: Capture a template solution with specific data types.

Liskov substitution principle:

The product of a base class with a method called knows, which is used to save the object to the file.

Do not reload the object from a file.

Make sure that Widget.save() accomplishes everything it was programmed to do.

Liskov's substitution principle serves to: increase reusability, increase extensibility, increase maintainability.

Interface separation principle:

Fundamental concepts: an interface is a window into the functionality of a component, the interface does not have to declare the possible methods of a component, a component can have many interfaces.

Dependency inversion principle:

Core ideas: organize the system into packages, abstractions should not depend on details, implementation details depend on abstractions.

Part two:

Software Engineering Goals

- supports reasoning about the desirable characteristics of a software system
- creates software with certain desirable characteristics
- an aspect of software design that imparts desirable characteristics.

Best practices, patterns and idioms

- These are procedures or techniques that help developers adhere to principles, without having to consider the principles or details of a situation at a theoretical level.

- The patterns exemplify the principles, providing proven solutions to recurring problems in specific contexts.

- Idioms are techniques that express a particular algorithm or data structure in a programming language.

Observation relative to modulary

- Good modularity should minimize the domino effect when software changes occur in expected (and some unexpected) ways.

- There are 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 modulary

Counterparts:

- Localization of design decisions and high cohesion can lead to many components which is good for testability, extensibility and reusability, but can hinder readability.

- By itself, modularity does not guarantee desirable features.

- However, the lack of modularity will compromise desirable features

- Adherence to or violation of modularity principles often affects multiple components.

Observations relative to abstraction

- Creating good software abstractions is difficult

- Software abstraction requires developers to examine large and

of details, and then determine the most salient and distinctive concepts.

- Abbott et al. describe an abstraction as the "reification and conceptualization of a distinction."

Paradigm-independent definition for encapsulation

PARADIGM INDEPENDENT DEFINITION FOR ABSTRACTION

Practices and criteria:

- Meaningful labels and identifiers

- Context-aware labels and identifiers

- Completeness of abstraction

- Abstraction Sufficiency OBSERVATIONS RELATING TO ENCAPSULATION

NON-REDUNDANCY AND COMPLEMENTARITY

NATURE OF MAE PRINCIPLES

CRITERION #1

- Modularity is concerned with decomposition into components, whereas abstraction and encapsulation are concerned with individual components.

- Therefore, modularity cannot be subsumed by either of the other two.

- Abstraction and encapsulation can be considered dual to each other.

CRITERION #2

- We demonstrate that the second criterion is met, i.e., that developers choose to follow each principle independently.