Introduction to the SOLID Principles

SOLID is an acronym for:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE

Every class should be working on only one thing of the program, and they have to be encapsulated

OPEN/CLOSED PRINCIPLE

Software entities should be opened for extensions but closed to modification, this means that a class is opened if it is available for extensions, if it's closed if its available for use of other classes.

We have the abstract classes as an example of an Open/Closed Principle, also we can achieve the open/close principle with: Inheritance, Aggregation and parametrization.

LISKOV SUBSTITUTION PRINCIPLE

For example, if s is a specialization of T, then an S object can be used wherever a T object is required.

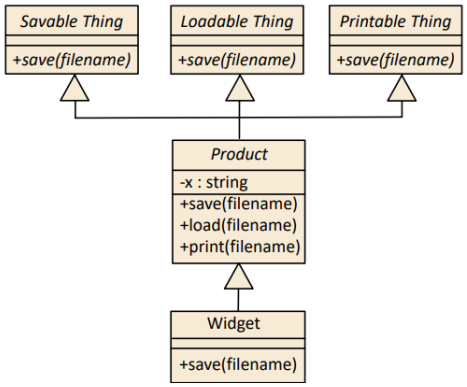
Let Product be a base class, with one virtual method, called save, whose intent is to save an object to a file, this handles exceptions appropriately

Following the Liskov Substitution Principle can help developers:

- Increase Reuse
- Increase Extensibility
- Increase Maintainability

INTERFACE SEGREGATION PRINCIPLE

One class can be implemented to more classes



This method reduce complexity by increasing Cohesion, increase extensibility, increase reuse and increase maintainability.

DEPENDENCY INVERSION PRINCIPLE

The Inversion Principle organizes the system into layers, like reusable libraries, Components from the abstract layers should not depend on components from the detail layers.

How to apply the Dependency Inversion Principle

- Abstraction should not depend on details
- Both low-level and high-level modules should depend on abstractions

The dependency Inversion Principle Increases reusability and maintainability

Software Engineering Goals

Software engineers aim to build quality products on time and within budget

For this reason, we have the Three core Principles: Modularity, Abstraction, Encapsulation.

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

There are no general, unifying definitions, especially for multiparadigm software development.

The principles are hard to teach and Programmer often don't understand the core principles, and therefore don't benefit from their guidance, especially in multi-paradigm software development.

In other words, a principle is a foundational concept (truth, proposition, rule, etc.) that leads to and supports reasoning about desirable characteristics, such as maintainability, efficiency, openness, reusability, etc.

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.

OBSERVATIONS RELATIVE TO MODULARITY

Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways. The other four SOLID principles overlap to some degree with basic idea of Low Coupling are actually specific best practices for achieving Low Coupling in certain contexts.

PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY

For FP, the components are primarily functions, but could also include other artifacts like build scripts. By definition, a pure function in FP only depends on values that are passed in as input parameters.

Every interesting or potentially changeable design decision needs to be localized. This is done by defining a predicate and set of rules for each design decision.

OBSERVATIONS RELATIVE TO ABSTRACTION

Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts. For each component, there is an explicit and clear declaration of the component's accessible features or functionality. Depending on the paradigm and programming language.

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

PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION

Ensure that the private implementation details of a component are insulated so they cannot be accessed or modified by other components. Doing so will lead to better testability, maintainability, and reliability.

Practices and Criteria:

- Conceptual barriers
- Programmatic barriers
- Usage barriers

NON-REDUNDANCY AND COMPLIMENTARY – CRITERION

- Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different.
- We show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets.

It's very important to clarify the purpose of Software Engineering principles, so we don't make mistakes applying the concepts when we are coding.