THE SOLID PRINCIPLES

OVERVIEW OF THE SOLID PRINCIPLES

SOLID is a mnemonic acronym for five principles

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

1.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
- "A class should have only one reason to change.", Robert C. Martin

2.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

3.LISKOV SUBSTITUTION PRINCIPLE

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

4.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
- No client (user of a component) should be forced to depend on methods that it does not use
- The public methods of a component can be grouped by purpose or responsibility as captured and declared in interfaces, or abstract classes

5.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

SOFTWARE ENGINEERING PRINCIPLES

- 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.
- If some concept, P, is an effective principle for achieving a set of desirable characteristics Q, then the degree to which a software engineer adheres to P should predicate the degree to which Q is present in the software artifacts.

BEST PRACTICES, PATTERNS, AND IDIOMS

- Patterns exemplify principles, by providing proven solutions to reoccurring 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 consistency with certain principles.

PRINCIPLES VS.BEST PRACTICES, PATTERNS, AND IDIOMS VS. DESIRABLE CHARACTERISTICS

- Principles are not desirable characteristics, but adherence to a principle should lead to desirable characteristics in the software, e.g.
- Abstraction à maintainability, reuse
- Principles should should give developers ways to
- Reason about design decisions
- Assess whether or how well a design either conforms to a principle
- Balance choices between conflicting objectives and design alternatives.

OBSERVATIONS RELATIVE TO MODULARITY

- Design decisions need to be “hidden” from the users of the component in which they are placed – this is actually encapsulation

OBSERVATIONS RELATIVE TO ABSTRACTION

- From a process perspective, abstraction is the act of bringing certain details to the forefront while suppressing all others.
- From a software artifact perspective, an abstraction is anything that exposures certain details that others can use and rely on
- Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts

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 effect