INTRODUCTION TO THE SOLID PRINCIPLES

SOLID is a mnemonic acronym for five principles:

• Single Responsibility Principle

• Open/Closed Principle

• Liskov Substitution Principle

• Interface Segregation Principle

• Dependency Inversion Principle

Following these principles can help ensure quality software, primarily from a developers' perspective.

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

This principle is very closely related to the more general principle of Cohesion, which says that the responsibilities of any component (method, class, sub-system, etc.) should be tightly aligned and focused on a single purpose,

OPEN/CLOSED PRINCIPLE

• Software entities (e.g., classes, generics) should be open for extension but closed to modification.

Original definitions:

• 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.

Ways to achieve the open/closed principle:

Inheritance

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

• The public methods of one class can be grouped into multiple abstractions

• Each abstraction should focus on a single purpose, as per the Single Responsibility Principle

• Have concrete classes inherit from these abstraction

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

Aggregation

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

• This technique has been embodied in something called the strategy pattern – more on this later

Parameterization

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

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:

• The implementation of save in Widget adheres to the purpose of save in

Product

• don't have it do some unrelated thing, like re-load the object from a file instead

• Widget.save() doesn't rely on stronger assumptions than Product.save()

• Programmatically implement any special conditions that Widget.save() required and handle exceptions appropriately

• Ensure that Widget.save() accomplishes, as minimum, all that Product.save() is supposed to accomplish.

• If Product.save() is supposed to save the x attribute to a file, then Widget.save() must do at least this much.

INTERFACE SEGREGATION PRINCIPLE

Foundational Concepts:

• 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

Core Idea:

• 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.

DEPENDENCY INVERSION PRINCIPLE

Core Ideas:

• 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

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”