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

SINGLE RESPONSIBILITY PRINCIPLE

• Core ideas:
• “A class should have only one reason to change.”, Robert C. Martin
• This principle is vey 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
• Localization of design decisions
• Encapsulation
• Following this principle can help
• Increase Reuse and Maintainability
• Reduce Complexity, even though the number of classes might increase

OPEN/CLOSED PRINCIPLE

• Ways to achieve the open/closed principle
• Inheritance
• Move public methods into their own abstractions, namely interfaces, abstract classes, or pure virtual classes
• Aggregation
• Encapsulate behaviors in sub-part objects and allow those sub-part object to change dynamically
• Parameterization
• Reduce complexity by reducing coupling (dependencies among components)
• Increase extensibility

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.
• Following the Liskov Substitution Principle can help developers
• Increase Reuse
• Increase Extensibility
• Increase Maintainability

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.

DEPENDENCY INVERSION PRINCIPLE

• 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”
• Following the Dependency Inversion Principle helps Developers
• Increase Reusability
• Increase Maintainability