UNIVERSIDAD DE LAS FUERZAS ARMADAS “ESPE”

HW-18 Software Engineering Principles

Name: Javier Paucar

NRC: 7490

The solid principles 1.14.40 AM

SOLID is a mnemonic acronym:

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 developer's perspective.

SINGLE RESPONSIBILITY PRINCIPLE:

- Every class should be responsible for a single part of the system’s functionality.
- “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 should be tightly aligned and focused on a single purpose.

This principle is also related to the principles of: localization of design decision and encapsulation.

Following this principle can help: reduce complexity, even though the number of classes might increase.

OPEN/CLOSED PRINCIPLE:

Software entities 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.

A system of classes is open for extension and closed for modification, if:

- Public methos are declared using interfaces, or abstract classes
- Users
- Concrete classes inherit the public method declarations from the interfaces, abstract classes, or pure virtual classes.

INTERFACES. ABSTRACT CLASSES, PURE VIRTUAL CLASSES:

Inheritance allows a specialization to re-use the generalization's:

- Data members
- Method declarations
- Methods definitions

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

- It does not allow data members to be declared
- It has no methods implementation
- Java supports interfaces

In UML, the names are written in italics

An abstract and pure virtual class may include data members and some method implementations:

- There is at least one method declaration without an implementation

The modern Open/Closed Principle encourages developers to:

- Use interfaces, abstract classes, and pure virtual classes to declare public data members

Ways to achieve the open/closed principle:

- Inheritance:
  - Move public methods into their own abstractions.
  - The public methos of one class can be grouped into multiple abstractions
  - Each abstraction should focus on a single purpose
  - Java does not support multiple inheritance, so 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

- Parameterization

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

Following the Open/Closed Principle can help developers

- Reduce complexity
- Increase extensibility.

LISKOV SUBSTITUTION PRINCIPLE:

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

FOLLOWING THE LISKOV SUBSTITUTION PRINCIPLE:

When implementing a specialization, Widget, of some Product, ensure that:

- The widget implementation adheres to the purpose of product specialization

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.

A component can have many interfaces.

Java does support interfaces directly.

An interface represents public methods of a component.

The public methods of a component can be grouped by purpose or responsibility

No client should be forced to depend on methods that it does not use.

INTERFACE SEGREGATION PRINCIPLE:

Following the Interface Segregation Principle, when used with other principles, can help Developers:

- Reduce complexity by increasing cohesion and reducing coupling
- Increase extensibility
- Increase reuse
- Increase maintainability

DEPENDENCY INVERSION PRINCIPLE:

Organize the system into layers

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

Following the Dependency Inversion Principle helps Developers:

- Increase Reusability
- Increase Maintainability