NAME: Camila Teca

INTRODUCTION TO THE SOLID PRINCIPLES

Following these principles can help ensure quality software

• SINGLE RESPONSIBILITY PRINCIPLE:

The class's responsibility must be encapsulated and every class must be responsible for only one part of the functioning of the system. This principle is related to the principle of cohesion (must focus on a single purpose) and is related to encapsulation.

This principle helps to reduce complexity, apply reusability and maintainability.

• OPEN/CLOSED PRINCIPLE:

A system of classes is open if it is available for extension and closed if it is available for use by another class, that is, it is closed for modification.

This principle is achieved:

-Inheritance, Aggregation, Parameterization, Encapsulation

-Public methods in your interfaces or abstract classes

-Group public methods into multiple abstractions

-Each abstraction must focus on a single purpose.

-Concrete classes inherit from these abstractions

-Technique "strategy pattern"

This principle helps reduce complexity and increase extensibility.

• LISKOV SUBSTITUTION PRINCIPLE:

Use of specialization, Strong behavior subtyping: both objects can do the same thing, Using conditions for methods, Conservation of supertype invariants, History restriction

Following this principle is achieved:

-A Product be a base class using the "save an object to a file" method.

-Implementation of specializations: Widget of some product fulfills a single purpose or a single function within the system.

-Implementation of any special condition that requires Widget and handling exceptions appropriately

This principle helps increase reusability, extensibility, and maintainability.

• INTERFACE SEGREGATION PRINCIPLE:

An interface is a functional "window" that represents the public methods of a component, without having to declare them. A component can have many interfaces

This principle achieves:

-The user may or may not depend on methods that are not used

-Group the public methods of a component by purpose or responsibility and declared in interfaces, or abstract classes

This principle helps reduce complexity, increases cohesion, coupling, extensibility, reusability, and maintainability.

- **DEPENDENCY INVERSION PRINCIPLE:**

Organize the system in layers, Abstractions should not depend on details, Implementation details must depend on abstractions

This principle achieves:

Abstractions should not depend on details

High-level modules must not depend on low-level modules, both must depend on abstractions

"Program to abstraction"

This principle helps increase reusability and maintainability.

UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION

Modularity, abstraction and encapsulation have value in all common software development paradigms, these paradigms are:

Object orientation (OO), Aspect orientation (AO), Functional programming (FP), Logic programming (LP), Genetic programming (GP), Structured program (SP)

Purpose of principles:

The principles will be a rule that must be followed to build quality products on time and within budget

Use of the best practices, patterns, and idioms:

They are a support to the developers helping them to follow the principles.

-Best practices: they are techniques that help to fulfill to the principles, without considering the theoretical details.

-Patterns: exemplify the principles

-The idioms: they are techniques or solutions to express a certain data structure in a specific programming language, in a coherent way.

There are three core principles:

- **MODULARITY:**

Observations relative to modularity:

-Location of design decisions.

-Design decisions must be “hidden” from users: encapsulation

-Minimize the effects of software: use of coupling and cohesion,

Some concepts of modularization:

- Grady Booch said that modularization is “To build modules that are cohesive

-Martin Fowler and other experts believe that modularity will prevent common code smells (long method, large class, long parameter list)

-G. Kiczales and others say that a developer should seek to understand it well enough to make changes.

Paradigm-independent definition for modularity:

Modularity exists in a software system when it is made up of loosely coupled, cohesive components that isolate every major or modifiable design decision in one component.

Modularity can improve understandability, testability, maintainability, reliability, security, extensibility, and reusability. It can also help during the software development process by delineating loosely coupled units of work.

Practices and Criteria:

Localization of design decisions, Low Coupling, High Cohesion, Modular Reasoning

•ABSTRACTION:

Observations relative to abstraction:

Abstraction is the act of saving important information while others are suppressed. Software abstraction requires a lot of work where developers examine large and various collections in detail, to determine the most outstanding

Common problems with abstraction:

-Leaky abstraction and excessive abstraction

-Insufficient control given to users of the abstraction

-Inadequate access to the information contained in the abstraction

Paradigm-independent definition for abstraction:

The explicit declaration of each component may or may not be part of the code or documentation, where its functions or features are accessible. Adherence to the principle of abstraction can improve understandability, testability, maintainability, and reusability. It can also allow developers to follow modularity more effectively,

Practices and Criteria:

Meaningful labels and identifiers, Context-aware labels and identifiers, Abstraction completeness, Abstraction sufficiency

•ENCAPSULATION:

Observations relative to encapsulation:

The encapsulation is not unique to OO, exist three categories for encapsulation:

-The bundling of data with operations

-The hiding decisions behind logical barriers

-The organization of components to minimize ripple effect

Paradigm-independent definition for encapsulation:

The private implementation of a component must be isolated so that no component can access it.

Encapsulation will lead to better testability, maintainability, and reliability.

Practices and Criteria:

Conceptual barriers, Programmatic barriers, Usage barriers