ANDRADE ALAN
NRC: 7490

INTRODUCTION TO SOLID PRINCIPLES

SOLID refers to a MNEMONIC ACRONYM that contains 5 principles, which are: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.

Design problem

You have been hired to build a maze generator.
The program should produce mazes containing N x M quarter squares, where N and M are the user-specified width and height of the maze.
The program must be able to print the mazes using ASCII characters or draw them on an image.

Open – closed principle

- Core Ideas:
- 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
- Revised definitions:
- A system of classes is open for extension and closed for modification, if
- Public methods (e.g., the abstractions) are declared using interfaces, or abstract classes in Java
- Ways to achieve the open/closed principle

Inheritance

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

Aggregation

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

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 by reducing coupling (dependencies among components)
- Increase extensibility

Interfaces, abstract classes, pure virtual classes

- Review: Inheritance allows a specialization (a derived class) to re-use the generalization's:
- Data members
- Method declarations
- Method definitions (i.e., their implementations)

Following the 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.

Dependency inversión 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

UNIFYING DEFINITIONS FOR MODULARITY, ABSTRACTION, AND ENCAPSULATION AS A STEP TOWARD FOUNDATIONAL MULTI PARADIGM SOFTWARE ENGINEERING PRINCIPLES

Software engeniering goals

- Software engineers aim to build quality products on time and within budget
- Some Desirable Qualities:
- understandability
- testability
- maintainability

- efficiency
- reliability
- security
- extensibility
- openness
- interoperability
- reusability

Common paradigms

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

Problema blackground

- Modularity, abstraction, and encapsulation have value in all these common software development paradigms, albeit to different degrees
- However, the concepts and definitions of these principles differ across
- paradigms
- In some cases, there are conflicting definitions within the same paradigm
- There are also many other proposed principles that overlap and break up
- the ideas differently.

Contributions of the initial paper

- The purpose of this paper is not to reinvent the concepts of modularity, abstraction, or encapsulation
- Instead, it is to stimulate discussion about the unification of existing ideas.

BEST PRACTICES, PATTERNS, AND IDIOMS

- Best practices are procedures or techniques that help developers adhere to principles, without having to consider the details of a situation at a theoretical level.
- Patterns exemplify principles, by providing proven solutions to recurring 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 consistent with certain principles.

OBSERVATIONS RELATIVE TO MODULARITY

- Good modularity should minimize ripple effects when the software changes occur in expected (and some non-expected) ways
- Two concepts that can help achieve this desirable characteristic:
- Coupling: the degree to which components depend on each other
- Cohesion: the degree to which the properties of a component relate to the component's primary responsibility

Paradigm independet definition for Modulation

Practices and Criteria:

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

OBSERVATIONS RELATIVE TO ABSTRACTION

- Creating good software abstractions is hard

- Software abstraction requires developers to sift through large and diverse collections of details, and then determine the most salient and distinguishing concepts
- Abbott et al. described an abstraction as the “reification and conceptualization of a distinction”

Paradigm independet definition for Abstraction

- •Practices and Criteria:
- Meaningful labels and identifiers
- Context-aware labels and identifiers
- Abstraction completeness
- Abstraction sufficiency

Paradigm independet definition for Encapsulation

- Practices and Criteria:
- Conceptual barriers
- Programmatic barriers
- Usage barriers

Non-redundancy and complimentary criterion N1

- Modularity deals with the decomposition of system into components, whereas abstraction and encapsulation deal with individual components
- Therefore, modularity cannot be subsumed by either the other two
- And, conversely
- Abstraction and encapsulation might be considered duals of each other, but one cannot subsume the other because the mechanisms for doing each are different

Non-redundancy and complimentary criterion N2

We show satisfaction of the second criteria, namely that developers and choose to follow each principle independent, with an example consisting of four functional-identical code snippets

- A simple program snippet with good Modularity, Abstraction, Encapsulation
- Same as #1, but with just good Modularity
- Same as #1, but with just good Abstraction
- Same as #1, but with just good Encapsulation