Object Oriented Design

SOLID Principles

unity coach

SOLID

- In computer programming, SOLID (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) is a mnemonic acronym introduced by Michael Feathers for the "first five principles" named by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design.
- The intention is that these principles, when applied together, will make it more likely that a programmer will create a system that is easy to maintain and extend over time.
- The principles of SOLID are guidelines that can be applied while working on software to remove code smells by providing a framework through which the programmer may refactor the software's source code until it is both legible and extensible.
- It is part of an overall strategy of agile and Adaptive Software Development.
- en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Single Responsibility Principle

- The single responsibility principle is a computer programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- Robert C. Martin expresses the principle as, "A class should have only one reason to change."
- en.wikipedia.org/wiki/Single_responsibility_principle

Open Closed Principle

- In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.
- The name open/closed principle has been used in two ways. Both ways use inheritance to resolve the apparent dilemma, but the goals, techniques, and results are different.
- en.wikipedia.org/wiki/Open/closed_principle

Liskov Substitution Principle

- Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of T (correctness, task performed, etc.).
- More formally, the Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled Data abstraction and hierarchy.
- It is a semantic rather than merely syntactic relation because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular.
- en.wikipedia.org/wiki/Liskov_substitution_principle

Interface Segregation Principle

- The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.
- ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- Such shrunken interfaces are also called role interfaces.
- ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.
- en.wikipedia.org/wiki/Interface_segregation_principle

Dependency Inversion Principle

- In object-oriented design, the dependency inversion principle refers to a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details.
- The principle states:
  - A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - B. Abstractions should not depend on details. Details should depend on abstractions.
- This design principle inverts the way some people may think about object-oriented programming, dictating that both high- and low-level objects must depend on the same abstraction.
- The idea behind points A and B of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought as an abstract interaction between them. This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.
- In many cases, thinking about the interaction in itself as an abstract concept allows the coupling of the components to be reduced without introducing additional coding patterns, allowing only a lighter and less implementation dependent interaction schema.
- en.wikipedia.org/wiki/Dependency_inversion_principle