OOP Principles

Object Oriented Programming

Piotr Fulmański

SOLID

SOLID

In software engineering, SOLID is a mnemonic acronym for *five* design principles intended to make object-oriented designs more *understandable*, *flexible*, and *maintainable*. The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin first introduced in his 2000 paper Design Principles and Design Patterns.

The SOLID ideas are:

- The **Single-responsibility** principle: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.

- The **Open–closed** principle: "Software entities ... should be open for extension, but closed for modification."

- The **Liskov substitution** principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." See also design by contract.

- The **Interface segregation** principle: "Clients should not be forced to depend upon interfaces that they do not use."

- The **Dependency inversion** principle: "Depend upon abstractions, [not] concretions."

The SOLID acronym was introduced later, around 2004, by Michael Feathers.

Although the SOLID principles apply to any object-oriented design, they can also form a core philosophy for methodologies such as agile development or adaptive software development.

SOLID

Single-responsibility

The principle of single responsibility says that each class should be responsible for one specific thing. In particular, there should be one specific reason to modify a given class. Applying this principle **significantly increases the number of classes in the program**, and at the same time **reduces the number of multi-purposes classes** such as the Swiss Army Knife.

SOLID

Open–closed

Each class should be written in such a way that it is possible to add new functionalities without the need to modify it. Modification is strictly prohibited as changing the declaration of any method may crash the system elsewhere.

SOLID

Liskov substitution

Any derived class can always be used in place of the base class. If we have the base class *Animal*, from which two classes inherit: *Dog* and *Cat*, then any function that accepts *Animal* type in the parameter should also handle the *Dog* and *Cat* instance. If additional conditions are needed, or worse, an exception is raised depending on the type of class, it is breaking the Liskov rule.

SOLID

Interface segregation

The rule is not to create interfaces with methods that the class does not use. Interfaces should be specific and as small as possible.

SOLID

Dependency inversion

The principle is to use a polymorphic interface wherever possible, especially in function parameters. If we have a function parameter that takes a mathematical figure, it would be much better to adopt an interface or an abstract class of mathematical figures than a specific figure. Thanks to this, we do not make a single method dependent on a specific type, but on the interface that large groups of subtypes can implement.

SOLID

- https://www.p-programowanie.pl/paradygmaty-programowania/zasady-solid

GRASP

GRASP

General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP, is a set of "nine fundamental principles in object design and responsibility assignment" first published by Craig Larman in his 1997 book Applying UML and Patterns.

Grasp

- http://www.kamilgrzybek.com/design/grasp-explained/
- https://www.fluentcpp.com/2021/06/23/grasp-9-must-know-design-principles-for-code/