Waiting for tea... # Good programming practices - SOLID #1

SOLID is an acronym introduced by Robert C. Martin (aka "Uncle Bob") in the early 2000s which describes 5 simple principles whereby written code can be more readable and easier to maintain.

Single responsibility principle

In **object-oriented programming**, the **single responsibility principle** states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

Robert C. Martin defines a responsibility as a *reason to change*, and concludes that a class or module should have one, and only one, reason to change. As an example, consider a module that compiles and prints a report. Such a module can be changed for two reasons. First, the content of the report can change. Second, the format of the report can change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

Text: Wikipedia - http://en.wikipedia.org/wiki/Single_responsibility_principle
Poster: Folks from LosTechies.com and Joey Devilla - http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/
Compilation: Marcin Zajączkowski - http://solidsoft.wordpress.com/ - Solid Soft - Working code is not enough

TBC

Waiting for tea... Good programming practices - SOLID #2

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 behavior to be modified without altering its **source code**. This is especially valuable in a production environment, where changes to source code may necessitate **code reviews**, **unit tests**, and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

The open closed principle is one of the oldest principles of Object Oriented Design. Historically it has been used in two ways. Both ways use inheritance to resolve the apparent dilemma, but the goals, techniques, and results are different. OCP was rediscovered in 90s. In Robert C. Martin's article we can read that modules that adhere to Open-Closed Principle have 2 primary attributes:

1. "Open For Extension" - It is possible to extend the behavior of the module as the requirements of the application change (i.e. change the behavior of the module).

2. "Closed For Modification" - Extending the behavior of the module does not result in the changing of the source code or binary code of the module itself.

Text: Wikipedia - http://en.wikipedia.org/wiki/Open/closed_principle & LosTechies.com - "Pablo's SOLID Software Development"
Poster: Folks from LosTechies.com and Joey Devilla - http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/
Compilation: Marcin Zajączkowski - http://solidsoft.wordpress.com/ - Solid Soft - Working code is not enough

TBC

Waiting for tea... Good programming practices - SOLID #3

Liskov substitution principle

Liskov substitution principle (a.k.a. design by contract) was initially introduced by Barbara Liskov in a 1987 and states that methods that use references to base classes must be able to use objects of derived classes without knowing it.

Liskov's principle imposes some standard requirements on signatures that have been adopted in newer object-oriented programming languages (usually at the level of classes rather than types - see nominal vs. structural subtyping for the distinction):

- contravariance of method arguments in the subtype,
- covariance of return types in the subtype,
- no new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

In addition to these, there is a number of behavioral conditions that subtype must meet. These are detailed in a terminology resembling that of design by contract methodology, leading to some restrictions on how contracts can interact with inheritance:

- preconditions cannot be strengthened in a subtype,
- postconditions cannot be weakened in a subtype.
- invariants of the supertype must be preserved in a subtype,
- history constraint (the "history rule").

A typical sign of LSP's violation is an instanceOf/static_cast usage with "if" statement to determine type of passed object and a method which should be call to process it.

Waiting for tea... Good programming practices - SOLID #4

Interface segregation principle

Clients should not be forced to depend upon interfaces that they do not use .

Interface segregation principle is used for clean development and is intended to help a developer avoid making his software impossible to change. If followed, the ISP will help a system stay decoupled and thus easier to refactor, change, and redeploy. The ISP says that once an interface has gotten too "fat" it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them. In a nutshell, no client should be forced to depend on methods it does not use.

The problem with "fat" interfaces is that anytime you wish to change your obese interfaces, or classes, and the methods that enable its gluttony, you will likely have to change or recompile all the clients of that interface, even if they are unrelated.

The visible symptom of overgrown interface is a number of methods it holds and also throwing "Unsupported" exception in its implementations.

Text: Wikipedia & LosTechies.com - "Pablo's SOLID Software Development"
Poster: Folks from LosTechies.com and Joey Devilla -
http://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

Waiting for tea... Good programming practices - SOLID #5

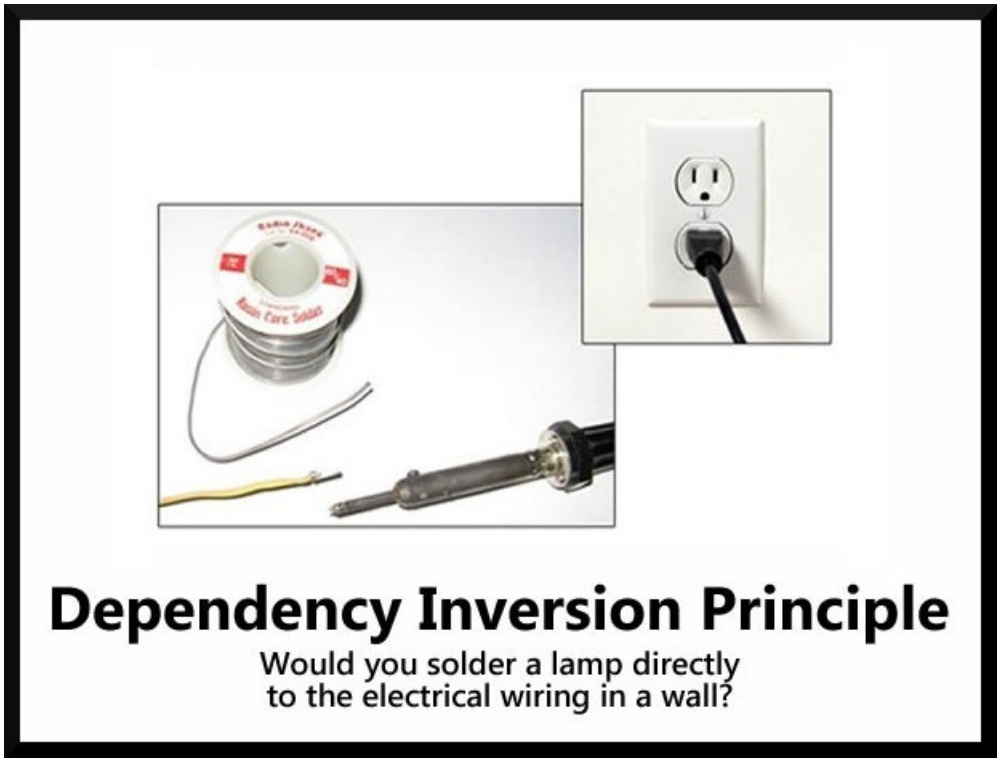
Dependency inversion principle

> High-level modules should not depend on low-level modules. Both should depend on abstractions.
> Abstractions should not depend upon details. Details should depend upon abstractions.

Dependency inversion principle refers to a specific form of decoupling where conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (e.g. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details.

In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. In this composition, higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.

The goal of the dependency inversion principle is to decouple high-level components from low-level components such that reuse with different low-level component implementations become possible. This is facilitated by the separation of high-level components and low-level components into separate packages/libraries, where interfaces defining the behavior/services required by the high-level component are owned by, and exist within the high-level component's package. The implementation of the high-level component's interface by the low level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship. Various patterns such as Plugin, Service Locator, or Dependency Injection are then employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.



Waiting for tea... **Good programming practices - SOLID #6**

SOLID

5 simple principles whereby written code can be more readable and easier to maintain

Single responsibility principle

There should never be more than one reason for a class to change.

Open/closed principle

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Liskov substitution principle

Functions that use ... references to base classes must be able to use objects of derived classes without knowing it.

Interface segregation principle

Clients should not be forced to depend upon interfaces that they do not use .

Dependency inversion principle

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.