Software Design Principles (S.O.L.I.D) GitHub

- ✓ Are you a software developer/researcher/teacher?
- ✓ Do you write poor-quality software, and you don't know what are your deficiencies exactly?
- ✓ Have you read the SOLID principles, but you forget them or cannot explain them clearly?
- ✓ Have you wondered about the direct relation between SOLID principles and OOP concepts or design patterns?

Then, this **capsule** is for you! You can now find all the answers in **one table**.

Caveat:

A reader should be familiar with SOLID and OOP principles before reading this capsule summarization. It is not meant to explain these principles from scratch. This file is free to print or distribute as is.

Principle	**S**ingle Responsibility (SRP)	**O**pen/Closed (OCP)	**L**iskov Substitution (LSP)	**I**nterface Segregation (ISP)	**D**ependency Inversion (DIP)
What is the meaning?	- A class or method should have one purpose only. Example	- A class or method should be open for extension but closed for modification. (It is an extension of SRP) Example	- Any instance of a parent class can be replaced with its child instance without breaking the code/negative side effects. –(It is an extension of OCP) Example	"Clients should not be forced to depend upon interfaces that they do not use" Example	"High level modules should not depend upon low-level modules. Both should depend upon abstractions." Example
When is it violated?	- When a class/method has 2 or more different responsibilities.	- When a class/method must be modified to add a new feature or functionality. (A Class/method here is open for modification and closed for extension)	- When a child alters the behavior of its parent. - When it is not possible to substitute a parent class reference with a child class object freely. - When a method/its signature/attribute exists in a parent class, but it is not applicable to **all** childs. - When a child uses a different datatype for the parent's attributes.	- when the client code is forced to implement methods it does not use.	- When a generic (high-level) class depend on a specific (low-level) class. Objects are created inside the class. - When a generic class is coupled to a specific class.
How to spot violation? When you see …	- Long lines of code for a class/method. - Lots of methods inside one class. - Constant interference in tasks within a development team.	- Lots of if statements to do something. - A class with hard-coded references to another class. - Objects are created inside a class.	- An empty implementation for a method in a child class. - A 'method not implemented' exception is raised inside a method of the child class, though method is not needed. - if statements and **isinstance()** method to identify the actual subclass to decide logic flow.	- An interface with many methods that maybe not related. - An interface with related methods but not used/needed by all clients that implement this interface.	- A high-level class can not be used without using a low-level class. - Object of a specific type is initialized inside a more generic class.
How to achieve it? How to avoid violation? How to solve violation?	- By making a class/method does one thing and does it well. - By splitting one class/method into 2 or more logical classes/methods.	- By splitting a responsibility of class/method in a way that the class/method can be extended or replaced but not modified. - By using Inheritance to make use of a class/method. - By adding one or more abstraction layers. - By introducing multilevel inheritance (either by higher level parents or interfaces)	- By adding one or more abstraction layers: either by higher level parents or interfaces) - By thinking of high-level abstractions/interfaces instead of low-level concrete implementation. - By not assuming all child classes will use the methods/their signatures of the parent class.	- Add more interfaces. - Put one method in an interface or multiple cohesive methods. - Split up one interface into multiple. - Make sure ALL clients will need to implement ALL methods in an interface.	- Invert the dependency. - Make high-level classes not dependent on low-level classes. - Make Both depend on abstractions. - Avoid initializing a specific object inside a generic class. - Introduce an abstraction layer between high-level and low-level classes.
-Why do we need it? -What are benefits? -How are benefits achieved?	- Promotes **Robustness:** by making changes without breaking the code. - Promotes **Maintainability**: by having only one reason to change/replace a class/method. - Promotes **Usability**: by making it easy to decide to use a class/method.	- Promotes **Extensibility**: by adding a new feature/use case to a class without modifying the existing code. (Done by inheritance) - Promotes **Maintainability**: by not needing to modify a single class/method (crucial when it is compiled/distributed and unmodifiable). - Promotes **re-utilization:** by using existing functionalities and not re-implement them.	- Promotes **Extensibility**: by providing a way to test and add newly created sub-classes to existing working tested code or a compiled project/library. - Promotes **Understandability:** by making the structure logical and easy to understand. - Promotes **Maintainability**: by reducing the if statements written to check the type of the subclasses.	- Promotes **Readability**: by making the code more understandable and not implementing unneeded methods. - Promotes **Maintainability**: by reducing the amount of unnecessary code and by splitting the logic across many single-purpose interfaces.	- Promotes **Maintainability**: by reducing dependencies and encouraging loose coupling between classes. - Promotes **Robustness:** by making high-level classes depend on abstractions, not concrete implementation.
What OOP concept is leveraged?	- **Abstraction** How? - **Encapsulation** How?	- **Inheritance** How? - **Abstraction** How? - **Encapsulation** How? - **Polymorphism** How?	- **Abstraction** How? - **Inheritance** How?	- **Abstraction** How? - **Inheritance** How?	- **Abstraction** **How?** - **Polymorphism** How?
What Design patterns is leveraged?	- KISS – Keep It Simple Stupid - DRY – Don't Repeat Yourself	- DRY – Don't Repeat Yourself - Template Pattern - Factory Pattern	- DRY – Don't Repeat Yourself	- KISS – Keep It Simple Stupid - DRY – Don't Repeat Yourself	- DRY – Don't Repeat Yourself

Table 1 – SOLID Principles and its relation to OOP and design patterns.

Author: **Mohamed H. Oreaba** | Head of Software Engineering | M.Sc. in Computer Science | The American University in Cairo | **mhamdy@aucegypt.edu**