SOLID

The SOLID principles were introduced by Robert C. Martin in his paper “Design Principles and Design Patterns”, in 2000. They have revolutionized the world of object-oriented programming. Why? Simply they help to create more maintainable, understandable, and flexible programs. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of work further down the road!
In particular, Solid principles deal with the "Release/Reuse Equivalence Principle". Reusability is one of the most oft claimed goals of OOP: in general, I reuse the code if, and only if, I never need to look at the source code; I need only to deal th libraries.

SOLID acronym:

- Single - responsibility principle: "a class should have one and only responsability, one single purpose; in particular, a class should have one and only one reason to change."
- Open - closed principle: "software entities (classes, modules, functions, etc.) should be open for extension but closed for modification."
- Liskov substitution principle: "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that programs."
- Interface segregation principle: "a client should not be forced to implement an interface that it doesn’t use, or clients should not be forced to depend on methods 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 on details. Details should depend on abstractions."

Let’s see them more in details.

- Single - responsability principle:
  - testing: a class with one responsibility will have far fewer test cases;
  - lower coupling: less functionality in a single class will have fewer dependencies and thus less possible problem;
  - organization: smaller, well-organized classes are easier to debug.

  In practice: identifying the responsibilities of an object is not simple.
- Open - closed principle:
  - we prevent the modification of existing code which could cause potential new bugs in an otherwise application;
  - we should write our code so that in the future we will be able to add new functionality without changing the existing code.

It is difficult to obtain a structure adhering to the open - close principle 100%, there will always be aspects that will escape our control; the important thing is that these aspects are those with the least probability of changing over time.

- Liskov substitution principle:
  - we should be able to replace a class with its subclasses without generating errors;
  - it requires the objects of our subclasses to behave in the same way as the objects of our superclass.

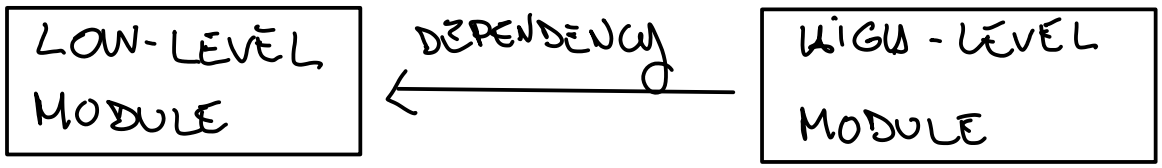
  In practice: overridden method of a subclass needs to accept the same input parameter values as the method of the superclass, as well as for the return value.
- Interface segregation principle:
  - it means that larger interfaces should be splitter into smaller ones;
  - we can ensure that implementing classes only need to be concerned about the methods that are of interest to them (avoid interface pollution by just adding new methods);
  - goal: reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts (similar to single - responsibility principle).

  Attention: The Liskov substitution principle and the Interface segregation principle may look similar in practice; the best distinction is what they are both looking at. The LSP is looking at the abstract design, while the ISP is dealing with the concrete code. More specifically:
  - the LSP deals with inheritance that does not make sense in terms of an object-oriented world;
  - the ISP deals with methods that are not used.

  Generally the LSP has a solution that involves dividing up into two different classes that are inherited from, while the ISP usually has a solution that involves making smaller interfaces with classes that use multiple, small interfaces.

- Dependency Inversion Principle:
  - based on the open - closed principle and the Liskov substitution principle;
  - instead of high-level modules depending on low-level modules, both should depend on abstractions (an abstract class or interface);
  - this does not just change the direction of the dependency; it splits the dependency between the high-level and low-level modules by introducing an abstraction between them; so that you get two dependencies:
    - the high-level module depends on the abstraction, and
    - the low-level depends on the same abstraction.

In practice: high-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, we need to introduce an abstraction that decouples the high-level and low-level modules from each other.



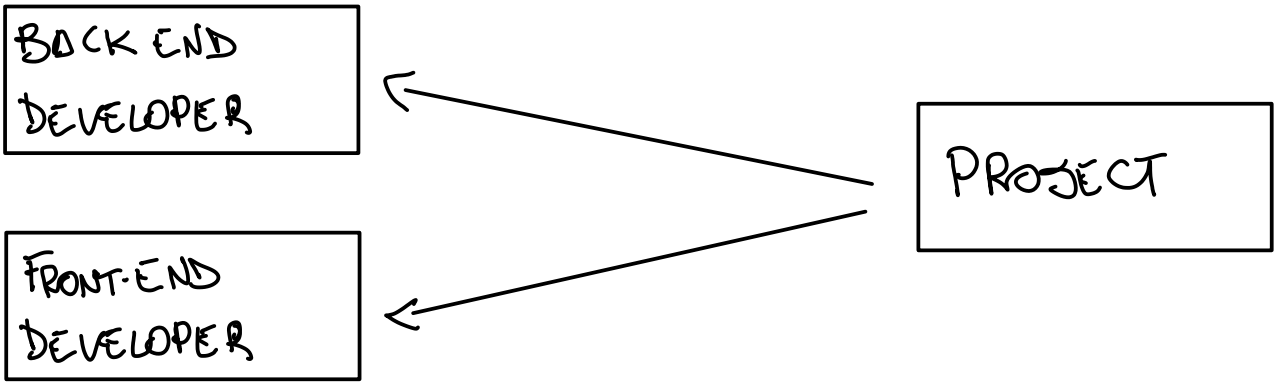
VIOLATION OF DIP

DIP:

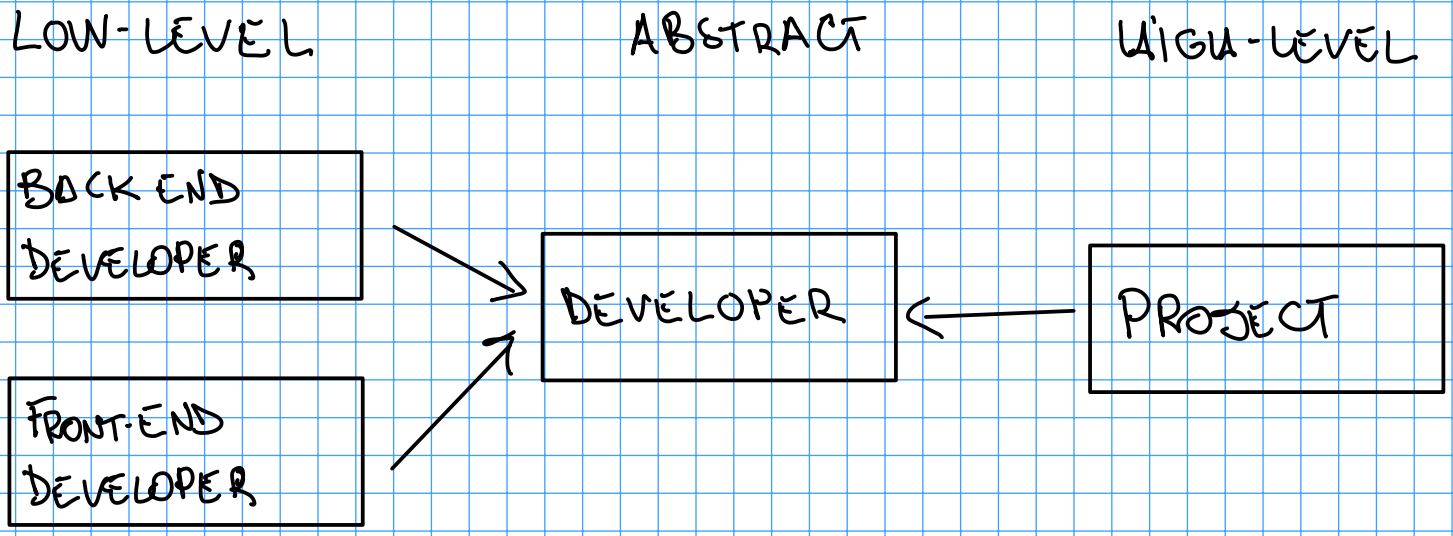
FIRST:

LOW-LEVEL

HIGH-LEVEL



AFTER:


LOW-LEVEL
ABSTRACT
HIGH-LEVEL
BACK END DEVELOPER
FRONT-END DEVELOPER
DEVELOPER
PROJECT