SOLID

S - Single Responsibility Principle
O - Open/Closed Principle
L - Liskov Substitution Principle
I - Interface Segregation Principle
D - Dependency Inversion

Single Responsibility Principle

- class or module should do one thing only
- counter-example
    -> class that opens a connection to the database,
      pulls out some table data, and writes the data to a file.
      This class has multiple reasons to change:
        -adoption of a new database,
        -modified file output format,
        -deciding to use an ORM, etc.
    In terms of the SRP, we'd say that this class is doing too much.

Open/Closed Principle

- code entities should be open for extension, but closed for modification.
- class should be closed for modification,
    but it can be extended by, for instance, inheriting from it and overriding
    or extending certain behaviors.
- Examples
    switch statement somewhere that you needed to go in and add
    to every time you wanted to add a menu option to your application.

    Apple, Google, and Microsoft does not provide OS Source code,
    they make the core phone functionality closed for modification and they open it to an extension.

Liskov Substitution Principle
--
- Any child type of a parent type should be able to stand in for that parent without things blowing up.
- Child class should have business logic inheriting from Parent class.
- Example
    Animal class -> MakeNoise() method,
    Subclass of Animal should reasonably implement MakeNoise().
    Cats should meow, dogs should bark, etc.
    What you wouldn't do is define a MuteMouse class that throws IDontActuallyMakeNoiseException.
    This violates the LSP, that this class has no business inheriting from Animal.

Interface Segregation Principle

- states that interfaces (like god classes) should be split into several interfaces.
- Large interfaces makes it harder to extend smaller parts of the system.

- You can create several smaller interfaces instead (depends on the class though).
- A client should never be forced to implement an interface that
    it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.

- Counter Example

    shape interface -> volume method(),
    but we know that squares are flat shapes and that they do not have volumes,
    so this interface force Square class to implement a method that it has no use of.

    -instead create another interface called -> SolidShapeInterface -> volume() Method,
    eg. solid shapes like cubes etc. can implement this interface.

Dependency Inversion

- describes that depends upon abstractions(generic class/interface) rather than upon concrete details.
- A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.
- B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

- Counter Example

    MySQLConnection - low level module & PasswordReminder -> high level module,
    this violates the principle as the PasswordReminder module -> forced to depend on MySQLConnection module.

    change in the database engine, also have to edit the PasswordReminder class

    PasswordReminder module -> should not care what database your application uses, to fix this “code to an interface”,
    create an interface having a connect() method and
    MySQLConnection class implements this interface, Hence no matter the type of database your application uses.