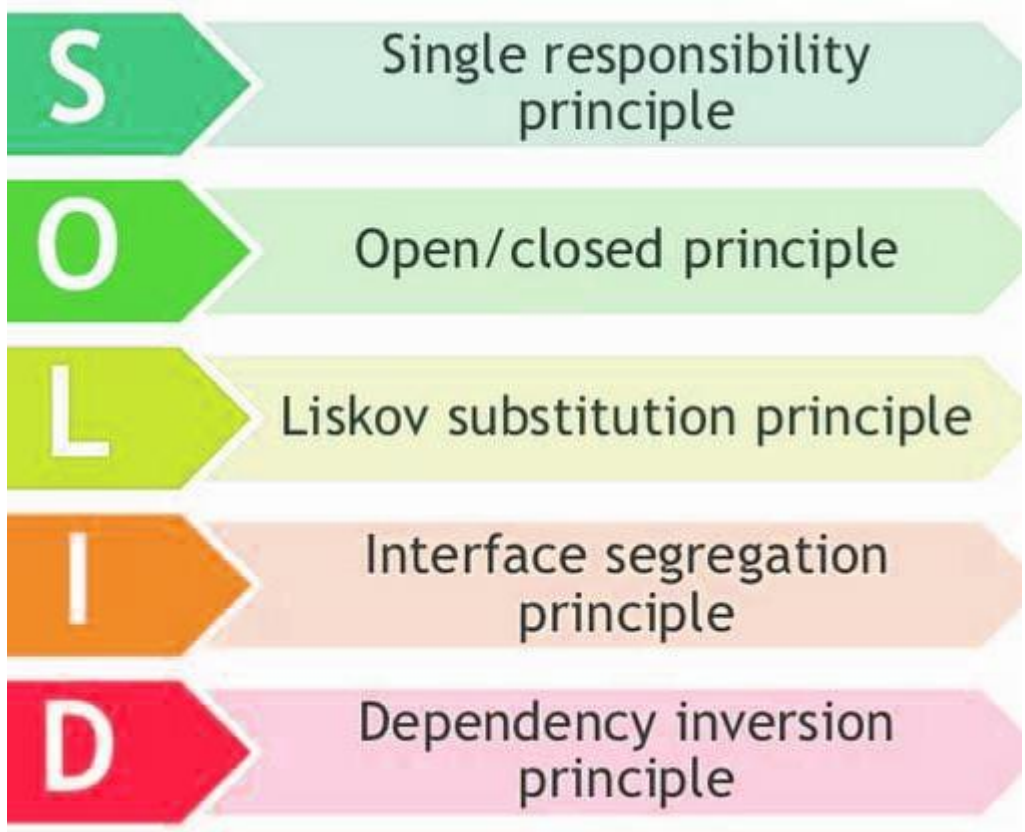
SOLID PRINCIPLES

BY **DAVID CASTLE**

WHAT IS **SOLID?**



Single Responsibility Principle

- **SOLID** is a mnemonic acronym for five design principles intended to make software designs more **understandable**, **flexible** and **maintainable**.
- The principles are a subset of many principles promoted by American software engineer and instructor **Robert C. Martin aka Uncle Bob.**
- Though they apply to any object-oriented design, the **SOLID** principles can also form a core philosophy for methodologies such as **agile development** or **adaptive software development**.

fdb First Databank

Single Responsibility Principle

‘A **class** or **module** should have one and only one reason to change’.

fdb First Databank

Single Responsibility Principle

Why is it useful?

- **Readability** – As your applications grow in size and complexity, readability becomes one of the top priorities. Code that is not readable, will lead to several failure points. The Single Responsibility Principle, ensures that your code is clean and readable at all times.
- **Testability** – Breaking down your code into small modules, that do only one thing, makes them easy to test.
- **Reusability** – Your code is now tested, and clean which means that they can be reused in several parts of your code.
- **Maintainability** – Code written with this principle in mind is easy to maintain on a long run.

Single Responsibility Principle

fdb First Databank

Single Responsibility Principle

Single responsibility means that your class (or any entity for that matter, including a method in a class, or a function in structured programming) should only do one thing. If your class is responsible for getting users' data from the database, it shouldn't care about displaying the data as well. Those are **different responsibilities** and should be handled **separately.**

fdb First Databank

Single Responsibility Principle

How do we solve this?

fdb First Databank

Single Responsibility Principle

The Single Responsibility Principle is one of the simplest of the principles, but one of the hardest to get right. Con-joining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about. Indeed, the rest of the principles we will discuss come back to this issue in one way or another.

fdb First Databank

Open/Closed Principle

‘Software entities should be **open** for **extension** but **closed** for **modification**.’

fdb First Databank

Open/Closed Principle

The **Open/Closed Principle** states that a module should be **open** for **extension** but **closed** for **modification**. That means you should be able to **extend** a module with new features not by changing its source code, but by adding new code instead. The goal is to keep working, tested code intact, so over time, it becomes bug resistant.

Open/Closed Principle

"Once done, don't change it, extend it"

Open/Closed Principle

How do we solve this?

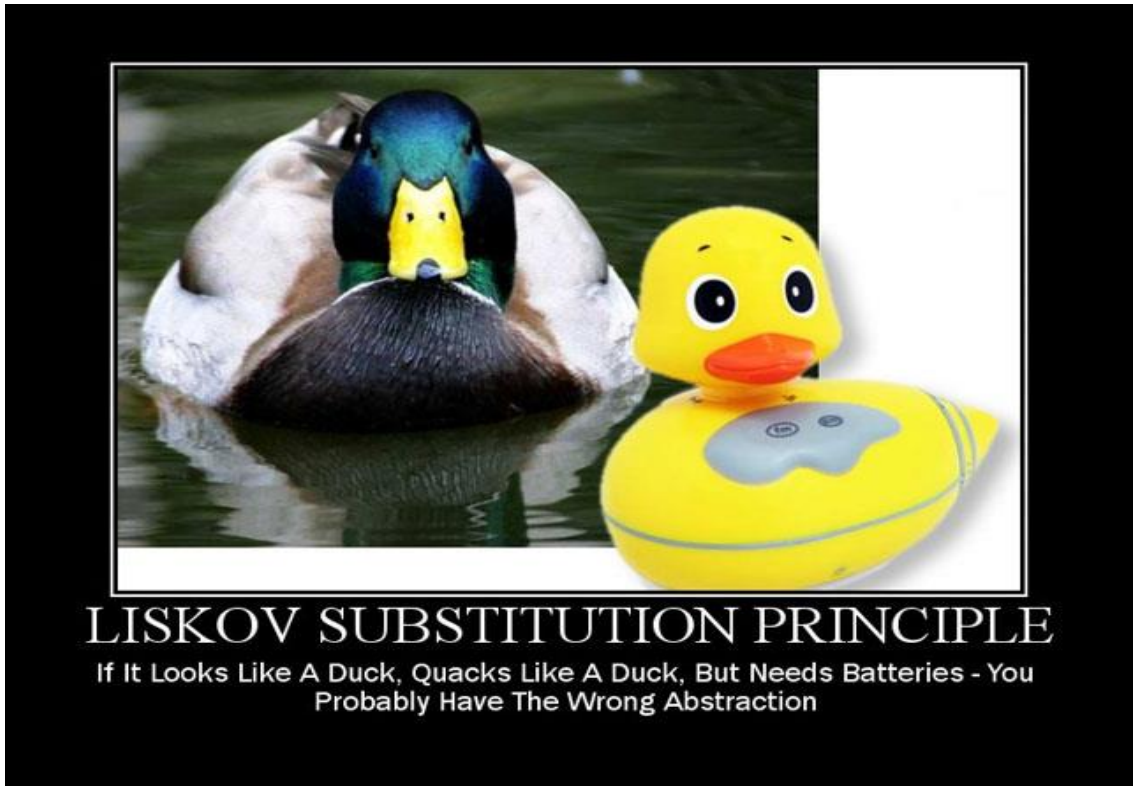
fdb First Databank

Open/Closed Principle

There is much more that could be said about the **open-closed** principle. Conformance to this principle is what yields the greatest benefits claimed for object oriented technology; i.e. **reusability** and **maintainability.** It requires a dedication on the part of the designer to apply **abstraction** to those parts of the program that the designer feels are going to be subject to change.

fdb First Databank

Liskov Substitution Principle



Liskov Substitution Principle

Or, in software developing terms, you should be able to substitute a class with any of its subclasses, without breaking the system. Putting it more simply, implementations of the same interface should never give a different result.

fdb First Databank

Liskov Substitution Principle

How do we solve this?

fdb First Databank

Liskov Substitution Principle

The **Open-Closed** principle is at the heart of many of the claims made for OOD. It is when this principle is in effect that applications are more **maintainable**, **reusable** and **robust**. The **Liskov Substitution** is an important feature of all programs that conform to the **Open-Closed** principle. It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

fdb First Databank

Interface Segregation Principle

‘Clients should not be forced to depend upon interfaces that they don't use’.

fdb First Databank

Interface Segregation Principle

Interface Segregation Principle

When more means less

fdb First Databank

Interface Segregation Principle

The **Interface Segregation Principle** states that clients should not be forced to depend on methods that they do not use. Interfaces should belong to clients, not to libraries or hierarchies. Application developers should favour thin, focused interfaces to "fat" interfaces that offer more functionality than a particular class or method needs.

Interface Segregation Principle

How do we solve this?

fdb First Databank

Interface Segregation Principle

Interfaces that are not specific to a single client such as fat interfaces, lead to inadvertent couplings between clients that ought otherwise to be isolated. By making use of the ADAPTER pattern, either through delegation (object form) or multiple inheritance (class form), fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

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**.'

fdb First Databank

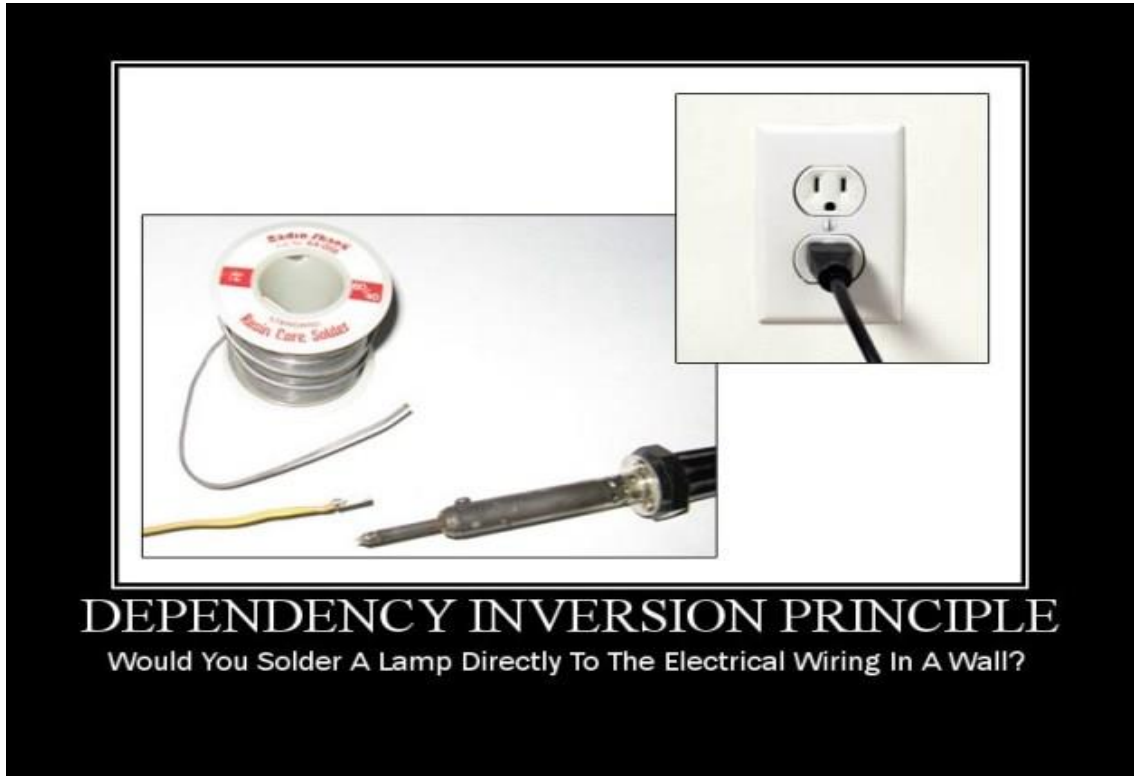
Dependency Inversion Principle

Let's use an example to explain this. Looking into your house, you're using electricity to plug in your laptop, your phone, lamp, etc. They all have a unified interface to get electricity: the socket. The beauty of it is that you don't need to care about the way the electricity is provided. You simply rely on the fact that you can use it when needed.

Now, imagine if instead of depending on the socket as an interface, you had to wire things up every time you needed to charge your phone. In software terms, that's what we do whenever we depend on concrete implementations: we know too much about how things are implemented which makes it easy to break the system.

fdb First Databank

Dependency Inversion Principle



Dependency Inversion Principle

Conclusions

- You **don't** want your modules to be tightly coupled together or it defeats the purpose of having them.
- You **do** want your modules to be highly cohesive, so they are all working efficiently towards the same goal.
- You **do** want to keep your modules as encapsulated as possible, so no one else knows (or needs to know) about their implementation details.
- The **SOLID** design principles essentially represent tests as to whether you are properly implementing those three characteristics.