



SOLID

Kris.Hermans@pxl.be

**DE HOGESCHOOL
MET HET NETWERK**

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook





SOLID

Software development is not a Jenga game.



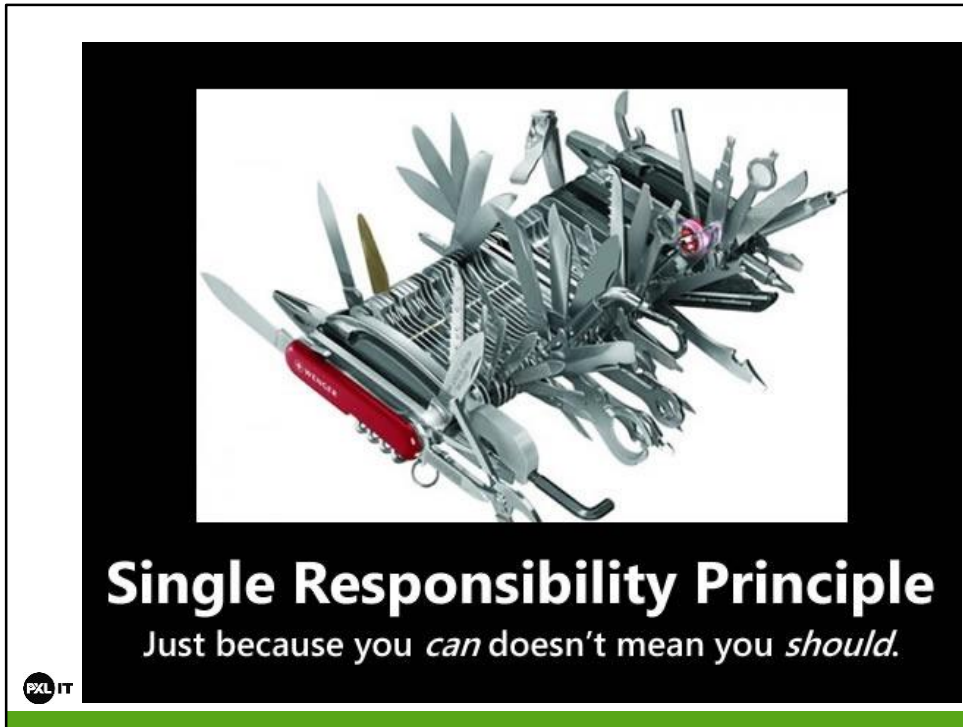
S-O-L-I-D

- Single Responsibility
- Open Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion



These principles, when applied, makes you a better developer and makes your software more maintainable, readable, testable etc.

Robert C. "Uncle Bob" Martin first gathered the object-oriented design principles that would eventually go under the acronym SOLID almost fifteen years ago, but the SOLID principles have been making the rounds lately and play an ever important role in software development.



The “S” in SOLID is for Single Responsibility Principle, which states that every object should have a single responsibility and that all of its services should be aligned with that responsibility. “Responsibility” is defined as “a 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.

Demo: OxygenMeter

- Before: what is wrong with this class?
- After: refactoring



Source: <http://www.blackwasp.co.uk/SRP.aspx>



Explain the image: you can add functionality (put on a coat), but don't need to change existing functionality (surgery).

Open-Closed Principle

- Open for extension
- Closed for modification



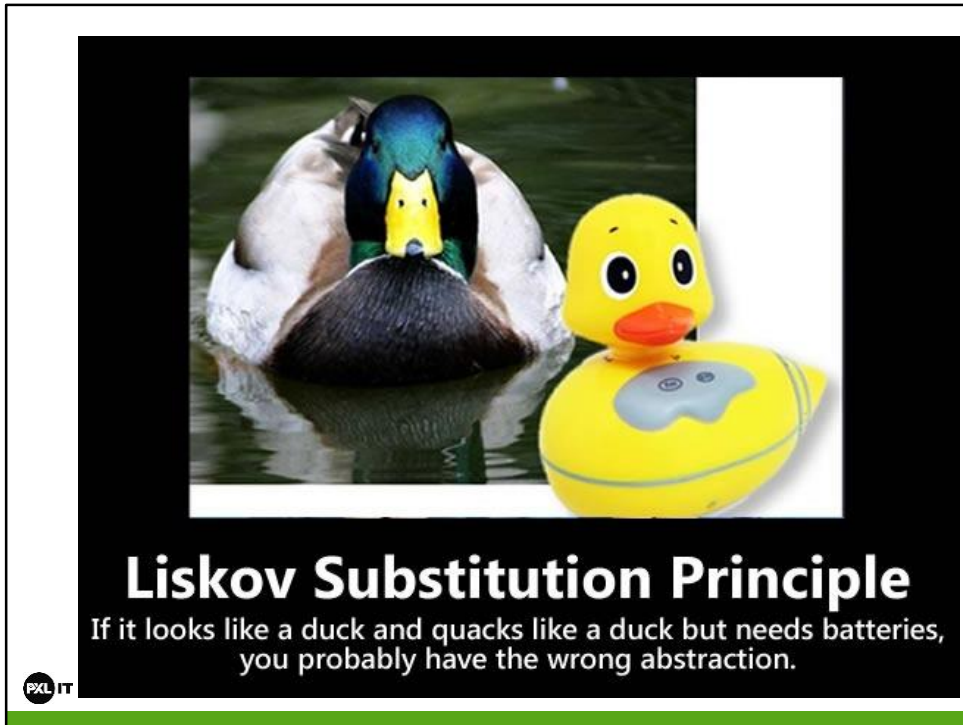
Software entities – such as classes, modules, functions and so on – should be open for extension but closed for modification. The idea is that it's often better to make changes to things like classes by adding to or building on top of them (using mechanisms like subclassing or polymorphism) rather than modifying their code.

Demo: Logger

- Before: what's wrong?
- After: Refactoring



Source: <http://www.blackwasp.co.uk/OCP.aspx>



The "L" in SOLID is for Liskov Substitution Principle, which states that subclasses should be substitutable for the classes from which they were derived. For example, if `MySubclass` is a subclass of `MyClass`, you should be able to replace `MyClass` with `MySubclass` without bugging up the program.

Liskov Substitution

- It specifies that you should design your classes so that client dependencies can be substituted with subclasses without the client knowing about the change. All subclasses must, therefore, operate the same manner as their base classes.
- Several rules for adhering to this principle, eg. Covariance, Contravariance, Invariants, etc.



<http://www.blackwasp.co.uk/LSP.aspx>

Demo: ProjectFile

- Before: what's wrong?
- After: Refactoring



The `ReadOnlyFile` class violates the LSP in several ways. Although all of the members of the base class are implemented, clients cannot substitute `ReadOnlyFile` objects for `ProjectFile` objects. This is clear in the `SaveFileData` method, which introduces an exception that cannot be thrown by the base class. Next, a postcondition of the `SaveFileData` method in the base class is that the file has been updated on disk. This is not the case with the subclass. The final problem can be seen in the `SaveAllFiles` method of the `Project` class. Here the programmer has added an if statement to ensure that the file is not read-only before attempting to save it. This violates the LSP and the OCP as the `Project` class must be modified to allow new `ProjectFile` subclasses to be detected.

There are various ways in which the code can be refactored to comply with the LSP. One is shown below. Here the `Project` class has been modified to include two collections instead of one. One collection contains all of the files in the project and one holds references to writeable files only. The `LoadAllFiles` method loads data into all of the files in the `AllFiles` collection. As the files in the `WriteableFiles` collection will be a subset of the same references, the data will be visible via these also. The `SaveAllFiles` method has been replaced with a method that saves only the writeable files.

The `ProjectFile` class now contains only one method, which loads the file data. This method is required for both writeable and read-only files. The new `WriteableFile` class extends `ProjectFile`, adding a method that saves the file data. This reversal of the

hierarchy means that the code now complies with the LSP.




The “I” in SOLID is for Interface Segregation Principle, which states that clients should not be forced to depend on methods they don’t use. If a class exposes so many members that those members can be broken down into groups that serve different clients that don’t use members from the other groups, you should think about exposing those member groups as separate interfaces.

Demo: Contact

- Before: what's wrong?
- After: Refactoring



<http://www.blackwasp.co.uk/ISP.aspx>



Dependency Inversion Principle

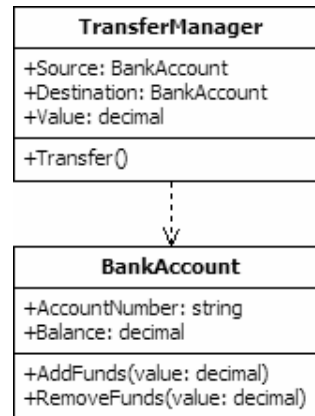
Would you solder a lamp directly
to the electrical wiring in a wall?

PXL IT

The “D” in SOLID is for Dependency Inversion Principle, which states that high-level modules shouldn’t depend on low-level modules, but both should depend on shared abstractions. In addition, abstractions should not depend on details – instead, details should depend on abstractions.

Demo: banking solution

- Before: what's wrong?



<http://www.blackwasp.co.uk/DIP.aspx>

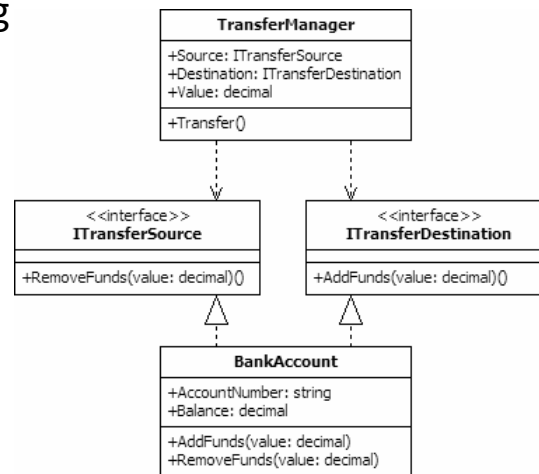
The DIP can be described more easily with an example. Consider a banking solution. As a part of the software it is necessary to transfer money between accounts. This may involve a class for a bank account with an account number and a balance value. It may include methods that add or remove funds from the account. To control transfers between accounts you may create a higher level *TransferManager* class. This may have properties for the two accounts involved in the transaction and for the value of the transfer. A possible design is shown in the slide.

The problem with such a design is that the high level *TransferManager* class is directly dependent upon the lower level *BankAccount* class. The *Source* and *Destination* properties reference the *BankAccount* type. This makes it impossible to substitute other account types unless they are subclasses of *BankAccount*. If we later want to add the ability to transfer money from a bank account to pay bills, the *BillAccount* class would have to inherit from *BankAccount*. As bills would not support the removal of funds, this is likely to break the rules of the Liskov Substitution Principle (LSP) or require changes to the *TransferManager* class that do not comply with the Open / Closed Principle (OCP). Further problems arise should changes be required to low level modules. A change in the *BankAccount* class may break the *TransferManager*. In more complex scenarios, changes to low level classes can cause problems that cascade upwards through the

hierarchy of modules. As the software grows, this structural problem can be compounded and the software can become fragile or rigid.

Demo: banking solution

- After: Refactoring



PXL IT

Applying the DIP resolves these problems by removing direct dependencies between classes. Instead, higher level classes refer to their dependencies using abstractions, such as interfaces or abstract classes. The lower level classes implement the interfaces, or inherit from the abstract classes. This allows new dependencies to be substituted without impact. Furthermore, changes to lower levels should not cascade upwards as long as they do not involve changing the abstraction.

The effect of the DIP is that classes are loosely coupled. This increases the robustness of the software and improves flexibility. The separation of high level classes from their dependencies raises the possibility of reuse of these larger areas of functionality. Without the DIP, only the lowest level classes may be easily reusable.

Exercise / Assignment

- Read the example code
- Choose **2** principles and apply to the DevOps Case
 - How were they violated (Before)?
 - How did you solve it (After)?

References

- <https://blogs.msdn.microsoft.com/cdndevs/2009/07/15/the-solid-principles-explained-with-motivational-posters/>
- <http://www.blackwasp.co.uk/SOLIDPrinciples.aspx>