



OOP via Python: Session 08

The **SOLID** Edition

Stephen Leach, Jan 2022



SOLID Design Principles

Wikipedia has a punchy introduction:

- **SOLID** is an [acronym](#) for a group of five [good](#) principles (rules) in [computer programming](#).
- SOLID allows programmers to write code [that is](#) easier to understand and change later on.
- SOLID [is often used](#) with systems that use an [object-oriented design](#).
- SOLID was [promoted by Robert C. Martin](#) but the name itself was created by Michael Feathers.



S.O.L.I.D.

- Single responsibility principle - "Class has one job to do"
- Open/closed principle - "Happy to be used by others; Unhappy to be changed by others."
- Liskov substitution principle - "Class can be replaced by any of its children"
- Interface segregation principle - "Code should not depend on methods it does not use"
- Dependency inversion principle - "When classes talk to each other in a very specific way, they should use promises (interfaces, parents), so classes can change as long as they keep the promise."



Thinking Critically about Design Principles

- Is it clearly stated?
- Are the principles sufficiently well-defined to be testable?
- Is there evidence supporting them?
- Are they constructive or simply evaluative? (Principle vs Metric)

**S is for
Single-responsibility principle**



S is for "Single responsibility principle"

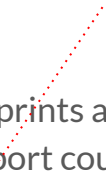
- Every *module, class* or *function* in a program should have responsibility over a single part of that program's functionality
- Martin expressed it as "A class should have only one reason to change," although, because of confusion around the word "reason" he also stated "This principle is about people." And also he claimed it is based on the principle of cohesion, as described by Tom DeMarco.
- Cohesion is described as "the degree to which the elements inside a module belong together"



SRP - Example 1

As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for **different** causes.

Q: Could a content change also require a format change, so just one reason?



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.

Q: Why not separate functions?





SRP - Example 2

... we are going to create an Invoice class with **four** functionalities as Adding and Deleting Invoices, Error Logging as well as Sending Emails. As we are putting all the above four functionalities into a single class or module, we are violating the Single Responsibility Principle in C#. This is because **Sending Emails** and **Error Logging** are not a part of the Invoice module.

Q: Four functionalities but only two separated out. Can that be right?



Make sense of SRP

```
public class Invoice {
    public void AddInvoice() {
        try {
            // Here we need to write the Code for adding invoice
            // Once the Invoice has been added, then send the mail
            MailMessage mailMessage = new MailMessage("EMailFrom", "EMailTo", "EMailSubject", "EMailBody");
            this.SendInvoiceEmail(mailMessage);
        } catch (Exception ex) {
            System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
        }
    }
    public void DeleteInvoice() {
        try {
            //Here we need to write the Code for Deleting the already generated invoice
        } catch (Exception ex) {
            System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
        }
    }
    public void SendInvoiceEmail(MailMessage mailMessage) {
        try {
            // Here we need to write the Code for Email setting and sending the invoice mail
        } catch (Exception ex) {
            System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
        }
    }
}
```

**O is for
Open-closed principle**



The Open/Closed Principle

From Bob Martin's 1996 paper "The Open-Closed Principle" - Modules that conform to the open-closed principle have two primary attributes.

1. They are “Open For Extension” This means that the **behavior** of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
2. They are “Closed for Modification”. The source code of such a module is inviolate. No one is allowed to make source code changes to it.



WTH?

- This addressed a common problem in languages such as C++ where class libraries ("modules") are compiled ("closed").
- People wanted to be able to extend modules (i.e, add stuff") even though they couldn't change the classes.
- In Python, this turns into "use methods rather than functions". Methods allow you to "extend", functions don't.

**L is for
Liskov substitution principle**

—



The Liskov substitution principle

Given type S and a sub-type T , then all the properties which are true for all objects of type S must also be true for all objects of sub-type T .

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .



A Genuine Principle (!)

- It is pretty clear and reasonably unequivocal
- Evaluative rather than constructive but incremental
- AFAIK evidence that it is a good practice is non-existent
- Equality (specifically equivalence relationships) are problematic - was that obvious?

I is for
Interface segregation principle



Interface segregation principle

- "No code should be forced to depend on methods it does not use"
- This is an implementation trick solves a problem in languages such as C++ where every parameter *must* be assigned a type by the programmer. And humans tend to supply full-fat types.
 - Splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
 - Such shrunken interfaces are also called role interfaces.
- Python doesn't suffer from this issue because of 'duck' typing.

**D is for
Dependency inversion
principle**



Not a Great Name

- It's really about minimising the dependencies due to using types that include unnecessary details or (equivalently) maximising the use of alternative types
- 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.



Explicit types create dependencies

- Use duck typing



X-Ray Vision



A Certain Same-ness

- S says: use types with a few, related methods
- O says: use methods not functions (i.e. use types)
- I says: use types that have a few methods
- D says: use types not classes routinely



Minimal Types / Thinnest Protocols

- Minimal types have fewest behaviours and simplest contracts
- Code units should be organised so as to reduce the set of methods that each variable/parameter must supply
- These sets of methods implicitly form Protocols and these should relate to domain concepts in a obvious/natural way.