# From STUPID to SOLID Code!

30 July 2013 — Clermont-Fd Area, France

0:00 / 11:00

Jump to: TL;DR

Last week I gave a talk about Object-Oriented Programming at Michelin, the company I am working for. I talked about writing better code, from STUPID to SOLID code! **STUPID** as well as **SOLID** are two acronyms, and have been covered quite a lot for a long time. However, these mnemonics are not always well-known, so it is worth spreading the word.

In the following, I will introduce both **STUPID** and **SOLID** principles. Keep in mind that these are **principles**, **not laws**. However, considering them as laws would be good for those who want to improve themselves.

## STUPID code, seriously? ¶

This may hurt your feelings, but you have probably written STUPID code already. I have too. But, what does that mean?

- **S**ingleton
- **T**ight Coupling
- **U**ntestability
- **P**remature Optimization
- **I**ndescriptive Naming
- **D**uplication

In the following, I will explain the individual points with more details. This is more or less the transcript of my talk.

### Singleton ¶

The Singleton pattern is probably the most well-known design pattern, but also the most misunderstood one. Are

you aware of the *Singleton syndrome*? It is when you think the Singleton pattern is the most appropriate pattern for the current use case you have. In other words, you use it everywhere. That is definitely **not** cool.

Singletons are controversial, and they are often considered anti-patterns. You should avoid them. Actually, the use of a singleton is not the problem, but the symptom of a problem. Here are two reasons why:

- Programs using global state are very difficult to test;
- Programs that rely on global state hide their dependencies.

But should you really avoid them all the time? I would say *yes* because you can often replace the use of a singleton by something better. Avoiding static things is important to avoid something called **tight coupling**.

## Tight Coupling ¶

**Tight coupling**, also known as **strong coupling**, is a generalization of the Singleton issue. Basically, you should reduce coupling between your modules. **Coupling** is the degree to which each program module relies on each one of the other modules.

If making a change in one module in your application requires you to change another module, then coupling exists. For instance, you instantiate objects in your constructor's class instead of passing instances as arguments. That is bad because it **doesn't allow further changes** such as replacing the instance by an instance of a sub-class, a *mock* or whatever.

Tightly coupled modules are **difficult to reuse**, and also **hard to test**.

## Untestability ¶

In my opinion, **testing should not be hard!** No, really. Whenever you don't write unit tests because you don't

have time, the real issue is that your code is bad, but that is another story.

Most of the time, **untestability** is caused by **tight coupling**.

### Premature Optimization ¶

Donald Knuth said: "*premature optimization is the root of all evil*. There is **only cost**, and **no benefit**". Actually, optimized systems are much more complex than just rewriting a loop or using pre-increment instead of post-increment. You will just end up with unreadable code. That is why Premature Optimization is often considered an anti-pattern.

A friend of mine often says that there are two rules to optimize an application:

- don't do it;
- (for experts only!) don't do it yet.

### Indescriptive Naming ¶

This should be obvious, but still needs to be said: **name** your classes, methods, attributes, and variables **properly**. Oh, and don't abbreviate! You write code for people, not for computers. They don't understand what you write anyway. Computers just understand `0` and `1`. **Programming languages are for humans**.

### Duplication ¶

Duplicated code is bad, so please Don't Repeat Yourself (DRY), and also Keep It Simple, Stupid. Be lazy the right way - write code only once!

Now that I have explained what STUPID code is, you may think that your code is STUPID. It does not matter (yet). Don't feel bad, keep calm and be awesome by writing SOLID code instead!

## SOLID to the rescue! ¶

SOLID is a term describing a **collection of design principles** for good code that was invented by Robert C. Martin, also known as *Uncle Bob*.

SOLID means:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

## Single Responsibility Principle ¶

Single Responsibility Principle or SRP states that **every class should have a single responsibility**. There should **never be more than one reason for a class to change**.

Just because you can add everything you want into your class doesn't mean that you should. Thinking in terms of responsibilities will help you design your application better. Ask yourself whether the logic you are introducing should live in this class or not. Using **layers** in your application helps a lot. Split big classes in smaller ones, and avoid god classes. Last but not least, **write straightforward comments**. If you start writing comments such as `in this case`, `but if`, `except when`, `or`, then you are doing it wrong.

## Open/Closed Principle ¶

Open/Closed Principle or OCP states that software entities should be **open for extension**, but **closed for modification**.

You should make all member variables **private** by default. Write getters and setters only when you need them. I've already covered this point in a previous article, as the ninth rule of the Object Calisthenics is related to this principle.

## Liskov Substitution Principle ¶

Liskov Substitution Principle or LSP states that objects in a program should be **replaceable with instances of their subtypes without altering the correctness** of the program.

Let's take an example. A rectangle is a plane figure with four right angles. It has a `width`, and a `height`. Now, take a look at the following pseudo-code:

```
rect = new Rectangle();

rect.width  = 10;
rect.height = 20;

assert 10 == rect.width
assert 20 == rect.height
```

We simply set a `width` and a `height` on a `Rectangle` instance, and then we assert that both properties are correct. So far, so good.

Now we can improve our definition by saying that a rectangle with four sides of equal length is called a square. A square **is a** rectangle so we can create a `Square` class that extends the `Rectangle` one, and replace the first line above by the one below:

```
rect = new Square();
```

According to the definition of a square, its width is equal to its height. Can you spot the problem? The first assertion will fail because we had to change the behavior of the setters in the `Square` class to fit the definition. This is a violation of the Liskov Substitution Principle.

## Interface Segregation Principle ¶

Interface Segregation Principle or ISP states that **many** client-specific **interfaces are better than one** general-purpose interface. In other words, you should not have to

implement methods that you don't use. Enforcing ISP gives you **low coupling**, and **high cohesion**.

When talking about **coupling**, cohesion is often mentioned as well. **High cohesion** means to keep similar and related things together. The **union** of cohesion and coupling is orthogonal design. The idea is to **keep your components focused** and try to **minimize the dependencies between them**.

Note that this is similar to the Single Responsibility Principle. An interface is a contract that meets a need. It is ok to have a class that implements different interfaces, but be careful, don't violate SRP.

## Dependency Inversion Principle ¶

Dependency Inversion Principle or DIP has two key points:

- **Abstractions should not depend upon details**;
- **Details should depend upon abstractions**.

This principle could be rephrased as **use the same level of abstraction at a given level**. Interfaces should depend on other interfaces. Don't add concrete classes to method signatures of an interface. However, use interfaces in your class methods.

Note that Dependency Inversion Principle is not the same as Dependency Injection. **Dependency Injection** is about how one object knows about another dependent object. In other words, it is about how one object acquires a dependency. On the other hand, DIP is about the level of abstraction. Also, a Dependency Injection Container is a way to auto-wire classes together. That does not mean you do Dependency Injection though. Look at the Service Locator for example.

Also, rather than working with classes that are tight coupled, use interfaces. This is called programming to the interface. This **reduces dependency** on implementation specifics and makes code **more reusable**. It also ensures

that you can replace the implementation without violating the expectations of that interface, according to the Liskov Substitution Principle seen before.

## Conclusion ¶

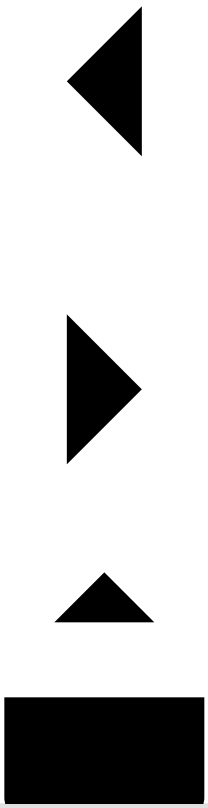
As you probably noticed, **avoiding tight coupling is the key**. It is present in a lot of code, and if you start by focusing on *fixing* this alone, you will immediately start writing better code.

If I may leave you with only one piece of advice, that would be to **use your brain**. There are a lot of principles in software engineering. Even if you don't understand all these principles, always think before writing code. Take the time to understand those that you don't understand.

Honestly, writing SOLID code is not that hard.

## Slides ¶

From STUPID to SOLID code!
by William Durand



## TL;DR ¶

**STUPID** is an acronym that describes bad practices in Oriented Object Programming:

- **S**ingleton
- **T**ight Coupling
- **U**ntestability
- **P**remature Optimization
- **I**ndescriptive Naming
- **D**uplication

**SOLID** is an acronym that stands for **five basic principles** of Object-Oriented Programming and design to *fix* STUPID code:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle

- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Rule of thumb: **use your brain**!

*By the way, if you found a typo, please fork and edit this post. Thank you so much! This post is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.*

> If you like this post or if you use one of the Open Source projects I maintain, say hello by email. There is also my Amazon Wish List. Thank you ♥

## Related Posts

Reviewing the FlexiSpot Desktop Workstation 27 inches 13 Mar 2017

PhD: ✓ 16 May 2016

Patching Linux Kernel (Raspbian & CVE-2016-0728) 21 Jan 2016

My Life On The Internets: A Year Later 16 Jan 2016

Level Up 08 Sep 2015

[Video] Nobody Understands REST, but That's Ok ;-) 02 Jun 2015

On capifony and its Future 11 Apr 2015

Playing With a ESP8266 WiFi Module 17 Mar 2015

## Comments