



# SOLID

Software development is not a Jenga game.

Son un conjunto de principios que, aplicados correctamente, ayudan a **escribir software de calidad** en cualquier lenguaje de programación orientada a objetos. Gracias a ellos, crearás código que será más fácil de leer, testear y mantener.

**S**

Single responsibility  
principle

**O**

Open/closed principle

**L**

Liskov substitution principle

**I**

Interface segregation  
principle

**D**

Dependency inversion  
principle

Fueron publicados por primera vez por Robert C. Martin, también conocido como Uncle Bob, en su libro:  
**Agile Software Development: Principles, Patterns, and Practices.**



# **Principio de Responsabilidad Única**

**NO ME IMPORTA LO QUE DIGA  
FOUCAULT**

**YO TENGO EL PODER**

El Principio de responsabilidad única (Single Responsibility Principle - SRP) fue acuñado por Robert C. Martin . SRP tiene que ver con el nivel de acoplamiento entre módulos dentro de la ingeniería del software.



HE SAYS...

"Una clase debe tener una, y solo una razón para cambiar."



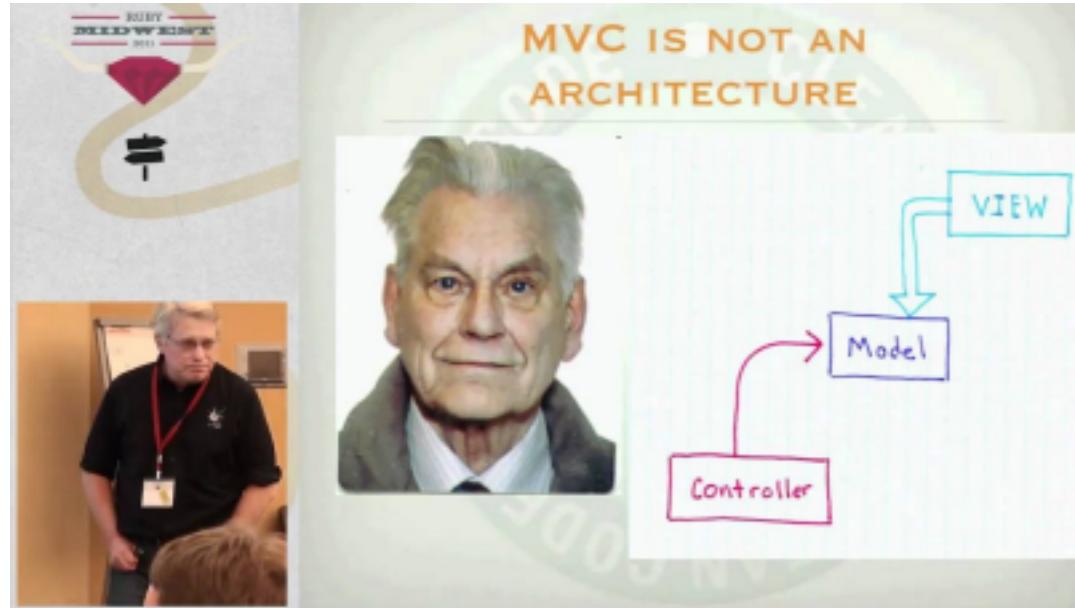
"Una clase debe tener una y solo una única causa por la cual puede ser modificada."

"Cada clase debe ser responsable de realizar una actividad del sistema."

# ¿Cómo detectar si estamos violando el Principio de Responsabilidad Única?



1. En una misma clase están involucradas dos capas de la arquitectura.



En toda arquitectura, por simple que sea, debería haber una capa de presentación, una de lógica de negocio y otra de persistencia. Si mezclamos responsabilidades de dos capas en una misma clase, es indicio de que no aplicamos un patrón arquitectónico.

## 2. El número de métodos públicos.

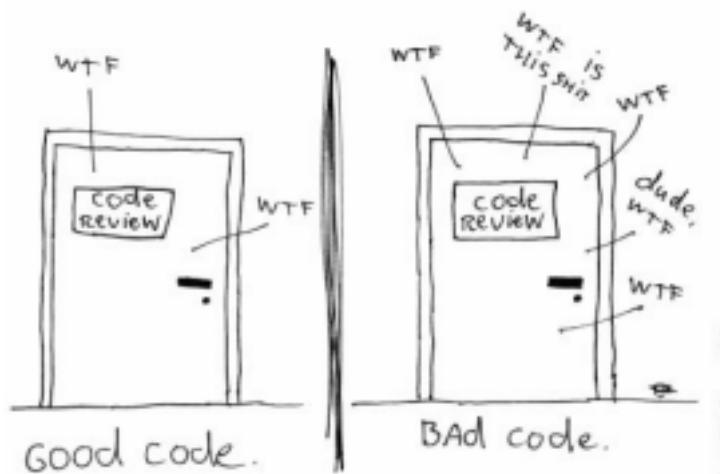


SINGLE RESPONSIBILITY PRINCIPLE  
*Just Because You Can, Doesn't Mean You Should*

Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos. Se debe detectar cómo agruparlos para separarlos en distintas clases.

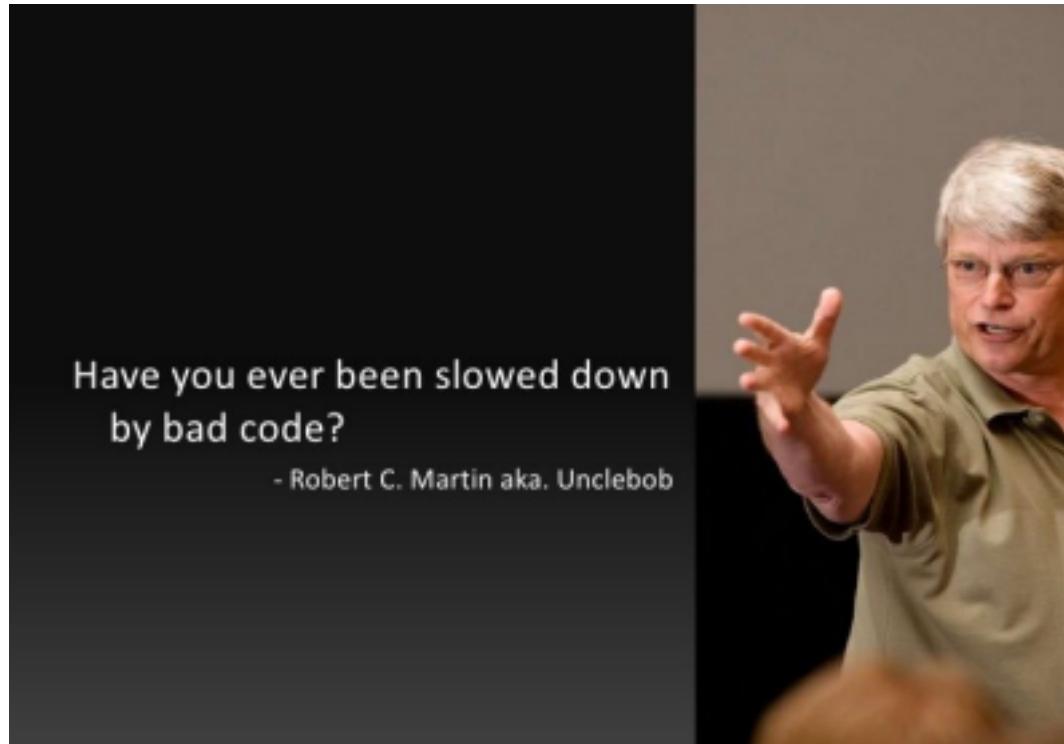
### 3. La cantidad métodos que usan cada uno de los atributos de una clase.

The ONLY VALID MEASUREMENT  
OF CODE QUALITY: WTFs/MINUTE



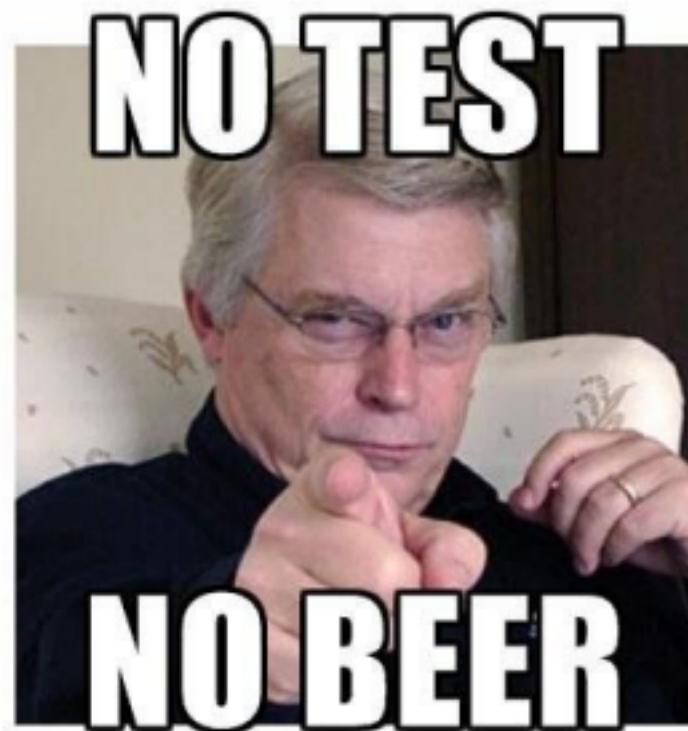
Si tenemos dos atributos, y uno de ellos se usa en unos cuantos métodos y otro en otros cuantos, esto puede estar indicando que cada campo con sus correspondientes métodos podrían formar una clase independiente. Normalmente esto estará más difuso y habrá métodos en común, porque seguramente esas dos nuevas clases tendrán que interactuar entre ellas.

#### 4. Por el número de imports o includes.



Si necesitamos importar demasiadas clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más. También ayuda fijarse a qué paquetes pertenecen esos imports. Si vemos que se agrupan con facilidad, puede que nos esté avisando de que estamos haciendo cosas muy diferentes.

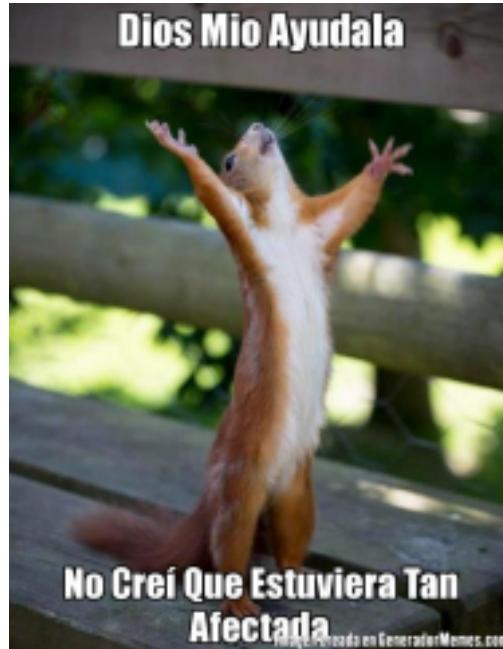
5. Nos cuesta “testear” la clase.



---

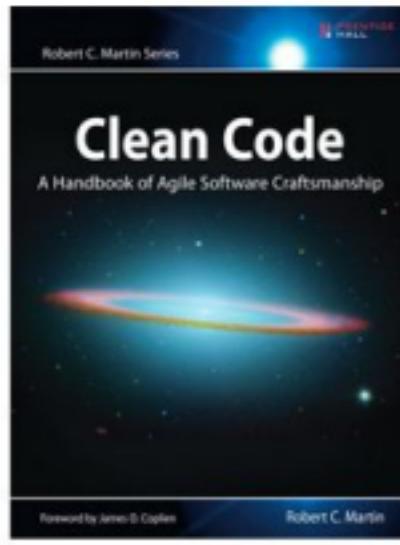
Si no somos capaces de escribir tests unitarios sobre una clase, o no conseguimos el grado de granularidad que nos gustaría, es momento de plantearse dividir la clase en dos o más.

6. Cada vez que escribes una nueva funcionalidad,  
esa clase se ve afectada.



Si una clase se modifica a menudo, es porque está involucrada en demasiadas cosas.

## 7. Por el número de líneas.



A veces es tan sencillo como eso. Si una clase es demasiado grande, hay que intenta dividirla en clases más manejables.

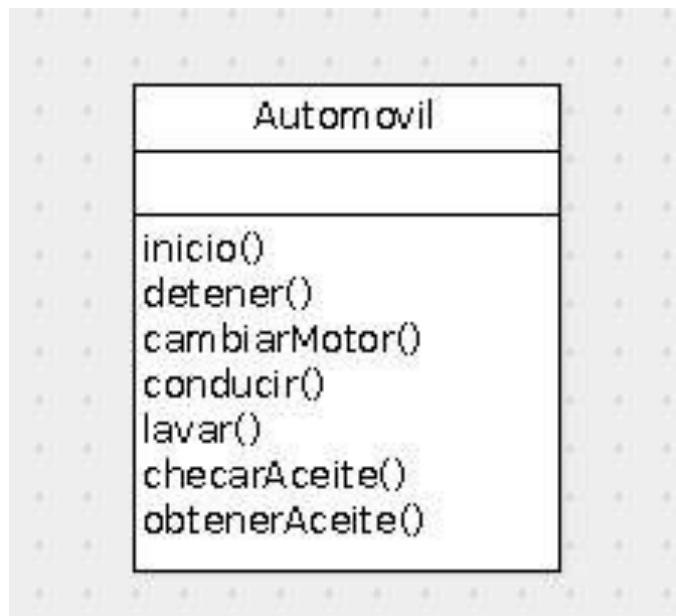
Una forma de probar este principio con lápiz y  
papel.

Una forma para probar este principio es escribir en un papel un cuadro, en la primera línea se escribe el nombre de la clase, en las siguientes líneas escribiremos el nombre de la clase y los métodos de la clase, ocuparemos tantas líneas como métodos tengamos en la clase, luego leeremos cada línea y analizaremos si la frase tiene sentido y si la clase tiene la responsabilidad del método descrito.

# Un ejemplo de análisis.



## La bendita clase Automóvil:



Veamos el resultado de aplicar el análisis en base al documento que se definió.

SRP Análisis para Automovil				Cumple SRP	Viola SRP
EI	Automovil	► Se <u>Inicia</u>	el mismo	X	
EI	Automovil	► Se <u>detiene</u>	el mismo	X	
EI	Automovil	► Se <u>cambia el motor</u>	el mismo		X
EI	Automovil	► Se <u>conduce</u>	el mismo		X
EI	Automovil	► Se <u>lava</u>	el mismo		X
EI	Automovil	► Se <u>checha el aceite</u>	el mismo		X
EI	Automovil	► Obtiene el aceite	el mismo	X	

Aclarando que el método `obtenerAceite()` se refiere solo a una lectura de aceite ó con una posibilidad de asignar esta responsabilidad a otra clase.

**Analicemos un poco nuestro  
código.**

¡Muchas Gracias por la paciencia!



# Enlaces

<http://blog.cleancoder.com/>

<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Fuente inspiradora al robo en esta presentación:

<https://devexperto.com/principio-open-closed/>