

Cuida tu código

Clean Code

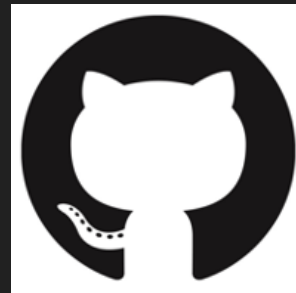
Acerca de mí



Jesús María
Escudero Justel

plain
concepts

<https://www.plainconcepts.com/>



<https://github.com/jmescuderojustel>



[@jm_escudero](https://twitter.com/jm_escudero)

Disclaimer

- Nada de esto es una verdad absoluta
- Todo es discutible **debe discutirse**
- La experiencia de cada uno es la clave

¿Qué vamos a ver?

- ¿Cómo es el **Clean Code**?
- ¿Por qué es necesario escribir **Clean Code**?
- ¿Cómo desarrollamos **Clean Code**?
- Charlaremos 😊

¿Qué prefieres?



<http://www.mrcleancarwash.com/blog/demystifying-car-maintenance-terms/>



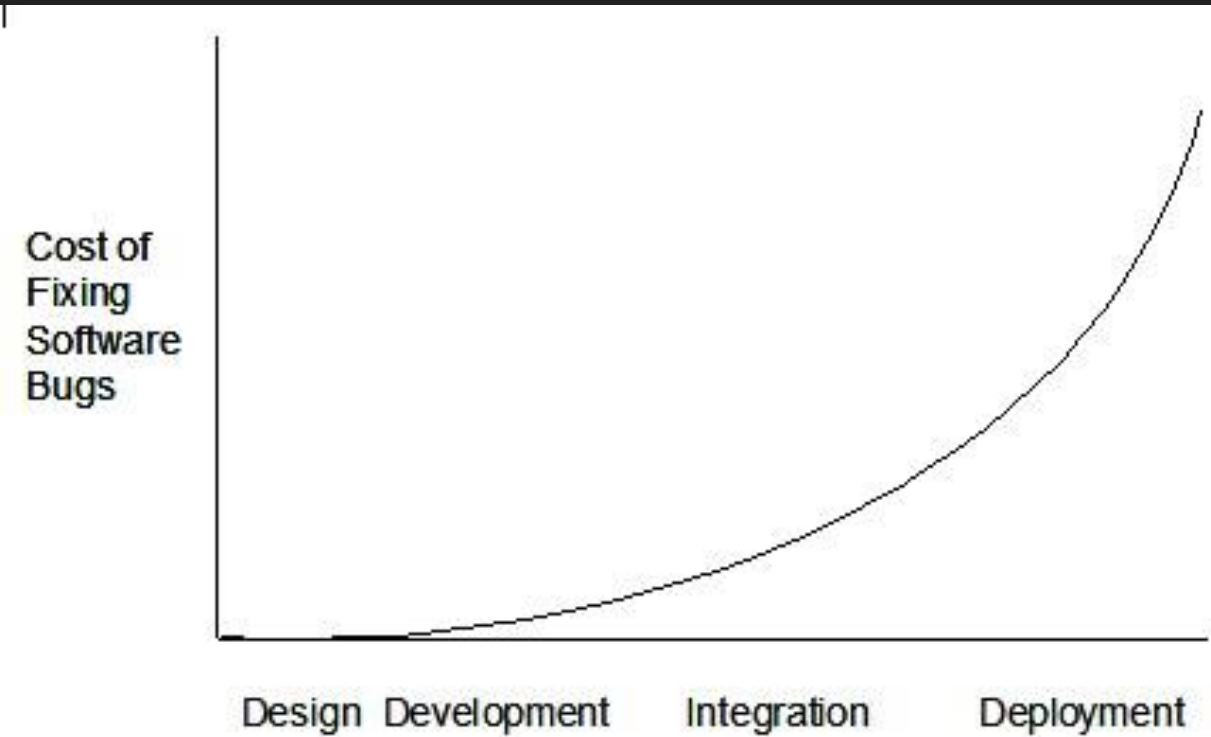
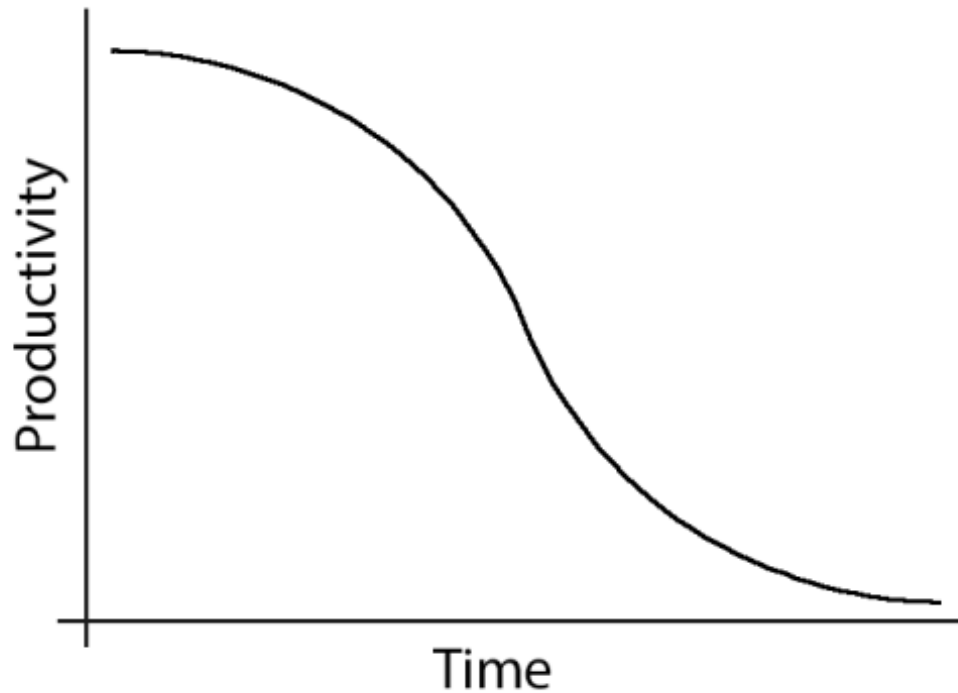
https://www.123rf.com/photo_40362683_old-car-engine-close-up.html

¿Cómo es el Clean Code?

- Elegante
- Legible (por el “yo” del futuro y por el resto)
- Sin duplicados
- Con tests (obligan a que haya Clean Code)
- Sencillo

Al leerlo, te das cuenta que el que lo ha escrito, ha cuidado el código y lo ha tratado con cariño

La realidad es cruel



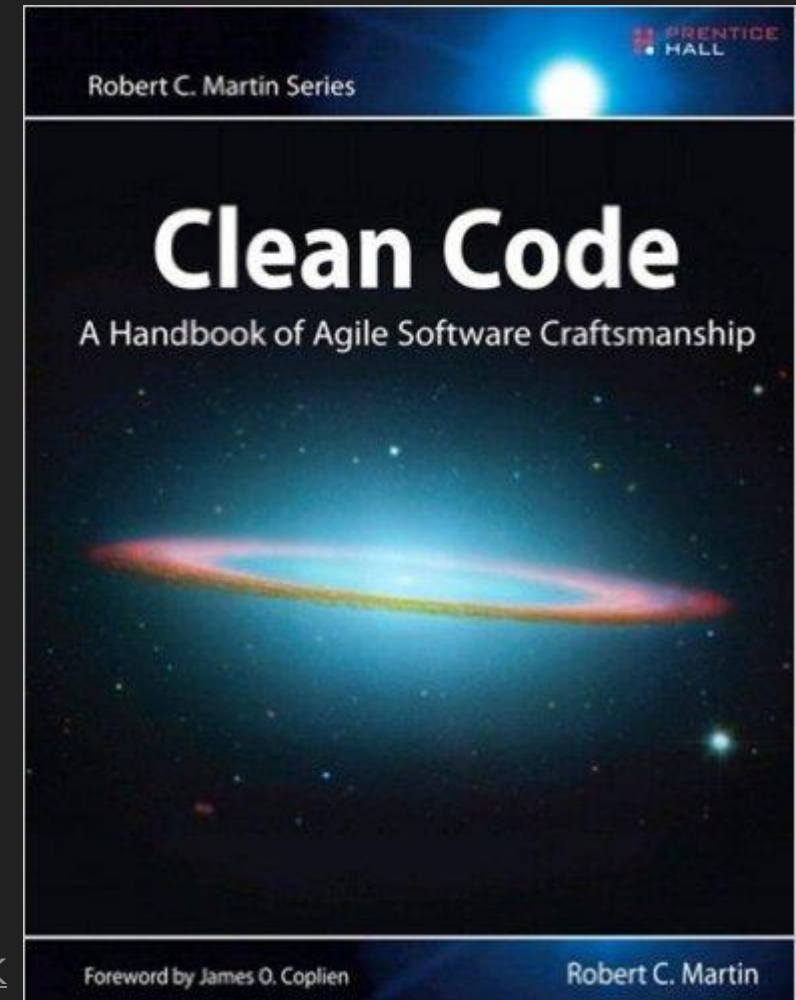
¿Por qué debería preocuparme de escribir Clean Code?

- Lees más código del que escribes (10 contra 1)
- Los proyectos deben ser mantenibles
- Minimiza bugs (y simplifica su corrección)
- Aunque parezca que avanzas más lento, es todo lo contrario
- El Clean Code origina más Clean Code

Somos profesionales

Bibliografía

- **Clean Code: A Handbook of Agile Software Craftsmanship**
 - By Robert C. Martin
- SOLID Principles in C# ([link](#))
- The Clean Code Blog ([link](#))
- Github ([link](#))



[link](#)

Naming

¿Hay algo raro en estos códigos?

`strName`

`getAddress`
`retrieveAddress`

```
for (var i = 0; i < list.length; i++) {  
    for (var j = 0; j < list[i].items) {  
        Console.WriteLine(list[i].items[j].name);  
    }  
}
```

`var manager = new Manager();`
`manager.Do();`

Naming

- Nombres de variables, métodos, clases, namespaces...
- Debemos pensarlos antes de ponerlos
- No debemos tener miedo a cambiarlos a lo largo del tiempo
 - De hecho, cambiarán
 - Con los nuevos IDEs es muy sencillo aplicar este refactor

Naming

- Los nombres deben:
 - Revelar la intención de la variable, método, ...
 - ¿Qué hace? ¿Qué pasará si uso ese método?
 - No representar cosas que no son
 - No llames **strAddress** a una variable "dirección". ¿Y si el día de mañana es de tipo **Address**?
 - No llames `userList` a una variable a menos que sea efectivamente una lista
 - Distinguir las variables
 - No uses **a1**, **a2**... usa el nombre de lo que son: **source**, **destination**...
 - Ser buscables, pronunciables (para poder hablar con tus compañeros...)

Naming

CLASES

- Sustantivos
- Evita **manager**, **info**, **data**...

MÉTODOS

- Verbos o verbos con sustantivo
- Coherencia: si dos cosas hacen lo mismo, ponles el mismo nombre (**get** VS **retrieve**)

Todos debemos hablar el mismo idioma: desarrolladores, managers, clientes...

Usa vocabulario del dominio (DDD)

Funciones

- Deben ser pequeñas (10-20 líneas como mucho)
- No deberían tener más de 2 indentaciones (if, switch...)
- Deben:
 - Hacer una cosa, únicamente
 - Hacer esa cosa bien

Funciones: naming



```
Sum(int first, int second);  
AddPrefix(string prefix);  
SaveFile(string name);
```



```
AddPrefixAndCapitalize(string prefix);  
GetSuggestion(bool includeDetails);
```

- Su nombre debe decir lo que hace
- Si no puedes ponerle un nombre, no sabes lo que hace
- Si el nombre es muy complicado, es posible que esa función haga más de una cosa
- Tendrá un verbo

Elegir un nombre no es fácil

Funciones: argumentos

☐ Ninguno

Funciones: argumentos

- Ninguno → Mola mucho



Funciones: argumentos

- Ninguno → Mola mucho
- Uno

Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal

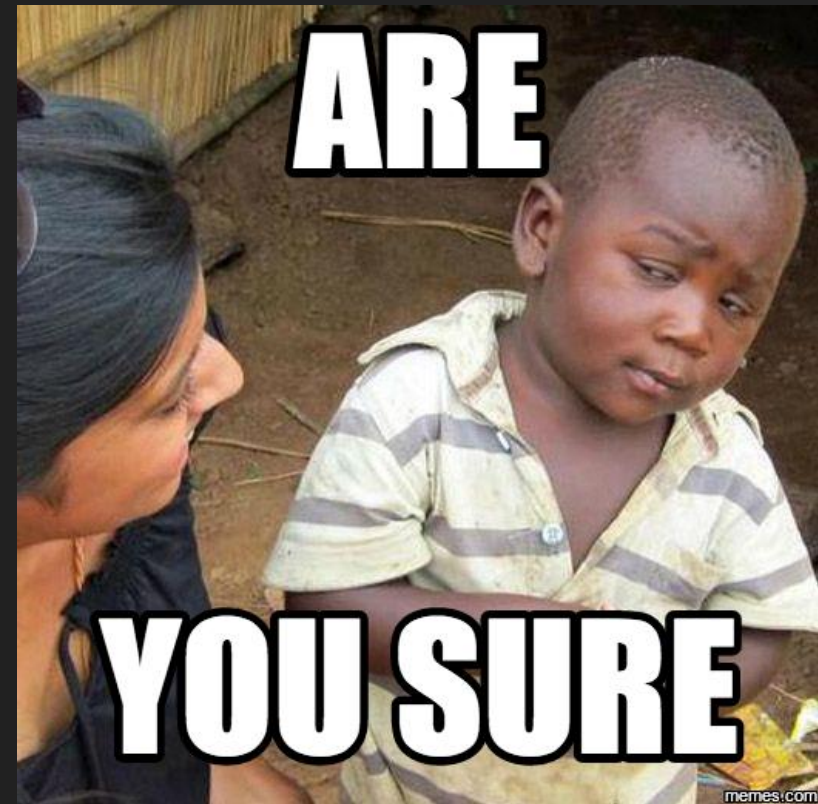


Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal
- Dos

Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal
- Dos → ¿Seguro?

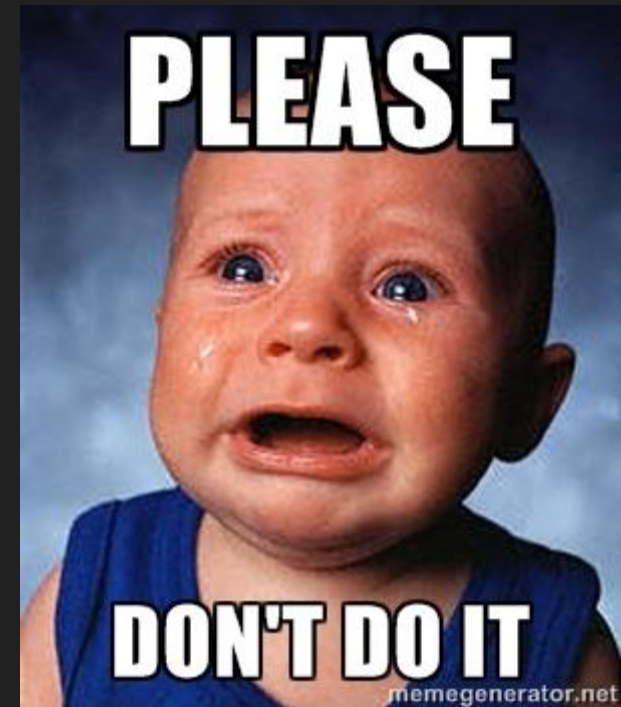


Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal
- Dos → ¿Seguro?
- Tres

Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal
- Dos → ¿Seguro?
- Tres → Necesitas una buena razón



```
CalculateFinalPrice(int pricePerUnit, int units, float  
taxInEU, float taxesInGB, int countryId, float  
volumeDiscount)
```

Funciones: argumentos

- Ninguno → Mola mucho
- Uno → No está mal
- Dos → ¿Seguro?
- Tres → Necesitas una buena razón

Cuantos más argumentos:

- Más casos contemplados en la función
- Más difícil de entender
- Muchos más casos que probar

Funciones: no olvides

- No repitas código
- Una función no debería cambiar nada que no sea suyo (o de su clase)
- Si una función es difícil de entender: no está bien

Funciones: The Law of Demeter

Un método de una clase solamente debería llamar a métodos de:

- La clase
- Un objeto creado en el método
- Un objeto pasado al método
- Un objeto que está en la clase


Comentarios


Por defecto...

NO

...aunque hay muchísimas opiniones

Comentarios: cuándo usarlos

- 
- Legales, licencias,...
 - Explicar por qué hacemos algo
 - Avisos de consecuencias
 - Documentación de API pública
 - Estructuras complicadas a simple vista (regular expressions...)
 - Notas **TODO**

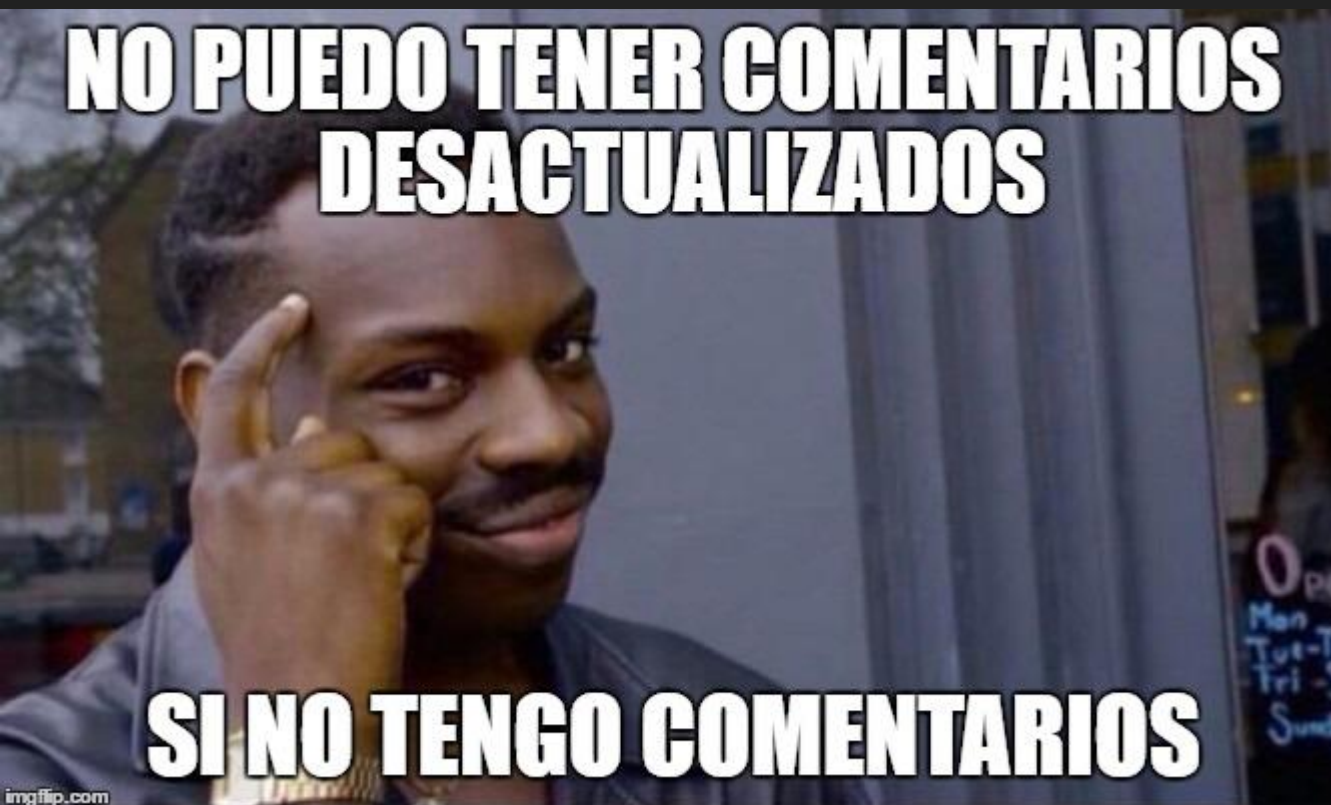
- 
- Explicar por qué el código hace lo que hace
 - Líneas vacías (dentro de **catch, else...**)
 - Argumentos de funciones
 - Funciones
 - Para marcar cambios de regiones
 - Para explicar el final de }
 - Código comentado

Comentarios

Un comentario es un síntoma de que algo necesita una explicación...

Igual no necesitas un comentario, sino cambiar tu código...

Comentarios: resumen



Es más fácil cambiar una línea de código, que una línea de código y sus comentarios

Formateo

- Cuidar la indentación
- Siempre mismos espacios entre métodos
- Las llaves { } siempre en el mismo sitio
- Formateo uniforme en todo los miembros del equipo

Coherencia

Clases: principios S.O.L.I.D.

Resumen:

- **S**: Single Responsibility Principle (SRP)
- **O**: Open closed Principle (OSP)
- **L**: Liskov substitution Principle (LSP)
- **I**: Interface Segregation Principle (ISP)
- **D**: Dependency Inversion Principle (DIP)

Classes

```
public interface IVehicle
{
    int SpeedInKmPerHour { get; }

    void SpeedUp();
    void SpeedDown();
}
```

```
public class Car: IVehicle {

    public SpeedInKmPerHour { get; private set; }

    void SpeedUp() {
        SpeedInKmPerHour += 1;
    }

    void SpeedDown() {
        SpeedInKmPerHour -= 1;
    }
}
```

```
public class Train: IVehicle {

    public SpeedInKmPerHour { get; private set; }

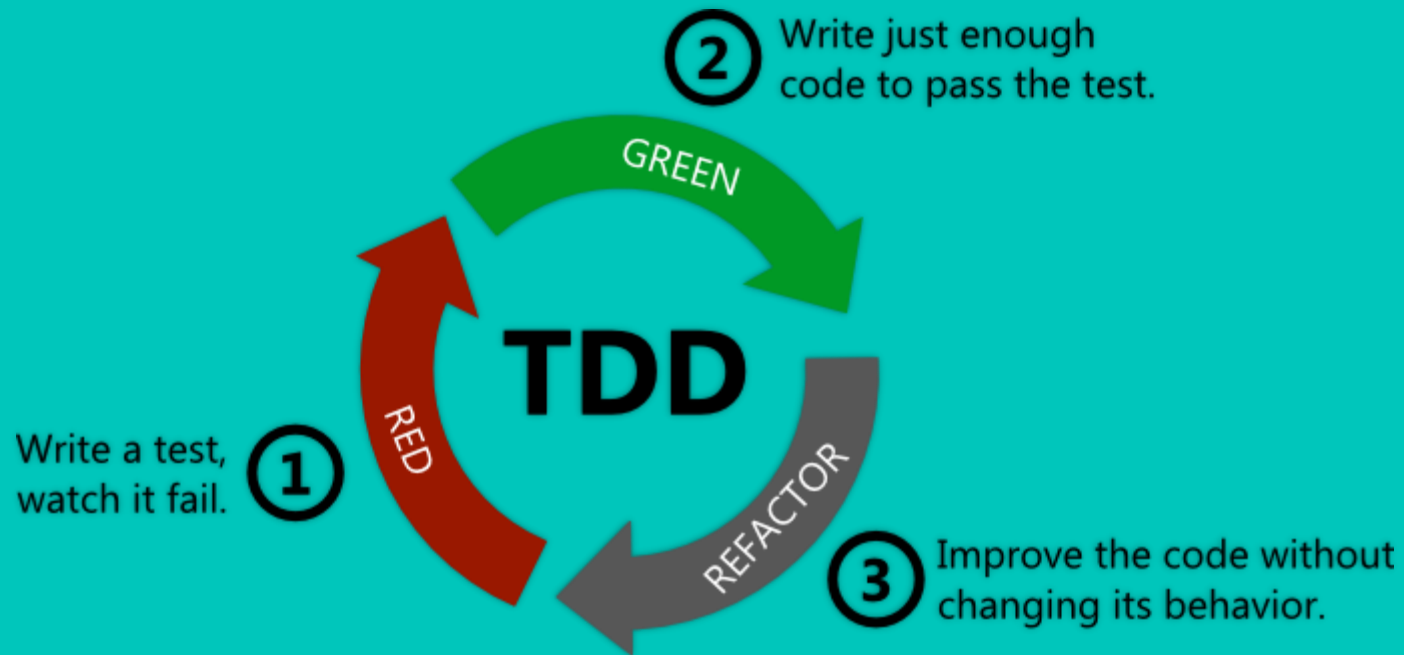
    void SpeedUp() {
        SpeedInKmPerHour += 10;
    }

    void SpeedDown() {
        SpeedInKmPerHour -= 10;
    }
}
```

Refactor

- Dedicar un tiempo a refactorizar
- Refactorizar no implica que al final haya menos código, sino que es mejor
- No tengas miedo a eliminar código
- Debes apoyarte en tests para poder tener la confianza de que el refactor no rompe nada
- El refactor puede ser eterno: marca un punto a partir del cual el beneficio no compensa el tiempo invertido

Refactor



<https://www.allaboutcircuits.com/technical-articles/how-test-driven-development-can-help-you-write-better-unit-tests>

Refactor

Practica

- Con tu propio código
- Intercambia refactors con tus compañeros (de buen rollo, ¿eh?)
- Kata:
 - <https://github.com/NotMyself/GildedRose/blob/master/src/GildedRose.Console/Program.cs>
 - <https://github.com/emilybache/Tennis-Refactoring-Kata>
- Github

¡Muchísimas gracias!

A:

- Juan (y Gorka)
- Alberto
- Borja
- Todos esos desarrolladores de los que voy aprendiendo...

¡Muchísimas gracias!

Y ahora...

