

# Clean Code

---

Funciones.



Para TrainingIT

**Por Alberto Basalo**

## 2 - Funciones

---

Claridad con el menor esfuerzo.

*"El buen código es su mejor documentación."*

--  **Steve McConnell**

Permitidme que siga tratando **la programación como un tipo concreto de escritura**. Hemos llegado hasta aquí teniendo un estilo y estética homogéneos y un vocabulario con sustantivos y verbos para expresar un dominio de conocimiento concreto. Es hora de empezar a escribir.

Y la escritura, ya se trate de blogs, de libros o de programas se basa en organizar palabras en frases. Y estas frases en párrafos. Y después vendrán ya vendrán los capítulos, o los tomos, quizás las series. Pero la unidad es la frase. Que para nosotros serán instrucciones. Aquí empieza **la claridad y la expresividad**.

Y el conjunto de frases forma los párrafos a los que llamaremos bloques. Muchos de esos bloques se comportarán como los párrafos narrativos, aparecerán una sola vez. En cambio otros aparecerán de manera repetitivas. Y algunos no aparecerán a menos que se cumplan ciertas condiciones. Esto es la base de las estructuras repetitivas y condicionales. El lugar dónde los programadores incrustamos **la lógica de negocio**.

¿Qué es un capítulo sin un título? Pues un conjunto de texto que solo cobra sentido al leerlo. En cambio, un buen título te dice mucho sobre lo que pasará después. Te hace ganar interés o te permite dejarlo para otro momento. Así les ocurre a los bloques con nombre: las funciones, subrutinas o métodos según el lenguaje (idioma) en el que programes (escribas). **Explican lo que hacen las instrucciones**.

Ah se me olvidaba que algunas de estas funciones, además, son invocadas desde diversos sitios facilitando uno de nuestros principios de limpieza: **evitar la redundancia**.

En las próximas entradas de este tutorial prestaremos especial atención a las sentencias y a su organización en bloques, estructuras y por último funciones.

*"A veces, la implementación más elegante es solo una función. No es un método. No es una clase. No es un framework. Solo una función."*

--  **John Carmack**

## 2.1 - Declaración, asignación e invocación

Claridad desde el interior.

"No soy un gran programador; Solo soy un buen programador con buenos hábitos"

--  **Kent Beck**

Los buenos hábitos para programar, **la disciplina**, es lo que hace realmente bueno a un programador. Tras el habito de estilizar el código y nombrar correctamente variables y funciones, es hora de escribir instrucciones.

Si lo piensas, todas **las instrucciones** que le das a una máquina caen en alguna de estas tres categorías: declarar el nacimiento de un variable o función; asignar valores a dichos variables o invocar las funciones.

No hay más, ni tampoco menos. Así que dediquémosle unos minutos.

### Declaración

Independientemente de las diabluras que te permita tu lenguaje (idioma), yo te propongo unas restricciones. Son unos buenos hábitos que no te costará adquirir.

- Una variable o constante por línea.
- Primero las constantes.
- Procura inicializar siempre con un valor tus declaraciones de variables.

Ya está, no es para tanto. Quizá alguno se plantee separar las declaraciones del resto de instrucciones con *una línea en blanco*. Vale; aunque en funciones pequeñas esto no será tan necesario.

Si la función no es pequeña... debería serlo.

### Asignación

Si has declarado variables es porque tienes pensado asignarles valores dinámicamente. Por ejemplo como resultado del cálculo de una expresión. Pues bien, una sola norma:

- **Haz que la expresión sea sencilla.**

¿Qué significa sencilla?

- Máximo 2 operadores aritméticos o booleanos.
- Usa paréntesis para evidenciar el orden de ejecución.
- Respeta el largo máximo de línea.
- Deja espacio alrededor de los operadores para que la expresión *respire*...

Y ¿si el expresión es mucho más compleja?

- Lleva a **funciones** todo aquello que incumpla lo anterior.

Acabarás con muchas funciones pequeñas... lo sé y me gusta.

## Invocación

Así que en cuanto la cosa se complique... **habrá que delegar en funciones**, métodos, rutinas o como le llaméis en vuestro idioma.

Y entonces tu instrucción de asignación incluirá una llamada o invocación a ese nuevo método o función.

## Atajos a vigilar

Algunos lenguajes facilitan el uso de **operadores condicionales** en medio de expresiones. Pero deben de ser sometidos a las reglas anteriores y estar muy vigilados. Considéralos como *azúcar sintáctico*: es goloso pero dañino si abusas.

### Operadores ternarios

En este caso, el uso del operador ternario se considera como si fuesen dos operadores. Por tanto invalida el anidamiento con otros ternarios o el uso de expresiones complejas en sus ramas de flujo.

- `condition ? value if true : value if false`

### Operadores lógicos

De nuevo, hay diferencias entre lenguajes. Los operadores *and*, *or*, *not* y familia no se representan siempre igual. El caso es que si abusas de notaciones muy concisas puedes estar entorpeciendo la incorporación de miembros junior; o dificultando la interpretación de una expresión demasiado *clever*

Intenta evitar los chequeos en busca nulos. Por ejemplo asignando valores por defecto en los argumentos de las funciones.

- `value = value || defaultValue;`
- `anObject && anObject.doSomething();`

```
// really? wtf!  
result = year % 400 === 0 ? true : year % 100 === 0 ? false : year % 4 === 0 ?  
true : false;
```

En resumen:

*No encadenes o agrupes estos atajos. Úsalos sólo como una abreviación de casos muy simples. Usa características del lenguaje para evitar tratar nulos y valores por defecto.*

Considera la posibilidad de **automatizar la detección de su incumplimiento** usando algún tipo de *linter* o chequeador estático.

## 2.2 - Estructuras repetitivas y condicionales

Bloques: Aquí vive la lógica.

*"Cada vez que escribas un comentario, debes sentirlo como un fallo de tu capacidad de expresión"*

--  **Robert C. Martin**

Cuando tengo cierta confianza con mis alumnos les suelo realizar una pregunta grosera:

¿Programas por dinero?

Tras el impacto viene un incómodo silencio para acabar reconociendo lo obvio. Por más que nos guste nuestra profesión,\*\* la inmensa mayoría de nuestro código la hemos escrito a cambio de dinero\*\*; o al menos de su expectativa.

Roto el hielo ya nos sinceramos y reflexionamos acerca de por qué otros nos dan su dinero. Y la respuesta suele ser que tienen un problema y nos necesitan para solucionarlo. Suele ser un problema complejo, pues de otra manera buscarían una solución menos costosa.





El caso es que **tienen un problema complejo y nosotros debemos resolverlo** programando. Es decir escribiendo en código las instrucciones que ejecutará un ordenador para satisfacer tu cliente.

Eres un traductor, un intermediario. No lo digo para menospreciar tu trabajo. Es para tomar consciencia de que eres un escritor. Y el lugar en el que mejor se ve esa labor de transmisión es en las estructuras condicionales que escribes, y su caso particular de las repeticiones. Es **en estas estructuras dónde realmente reflejas la solución al problema** de tu pagador.

### Condicionales

Son los famosos *if else switch* En tu lenguaje (idioma) puede que se digan de otra forma pero apuesto a que tienen su equivalente.

La recomendación para expresar la lógica van de lo simple a lo complejo.

-  Si es trivial puedes usar operadores ternarios
- En otro caso utiliza siempre estructuras y envuelve los bloques  entre llaves.  aunque el lenguaje no te obligue.
- LA condición de  sólo operador lógico.
- Si la condición es compleja debe convertirse en una una función cuyo nombre comenzará por un verbo del estilo *is, has, can should...*
- Favorecer el retorno **anticipado** cuando las condiciones chequean datos erróneos o incompletos.
- Favorecer el retorno **unificado**. para la lógica de negocio.
- Evitar los **switches**. Ya veremos cómo.

## Repetitivas

Son un caso particular de condiciones que repiten la ejecución de un bloque de instrucciones mientras o hasta que se cumpla una condición.

De nuevo aplican los mismo criterios:

- LA condición de ruptura `break` sólo operador lógico.
- Las variables **locales** deben ser legibles.
- Se permiten los índices clásicos `i`, `j`.

## Límites

Los algoritmos que resuelven problemas de negocio, aquellos por los que te pagan, suelen ser complejos. Es muy común encontrar grupos de las anteriores estructuras juntas para resolver un problema.

La cuestión es que no pasa nada por encontrar un

```
bucle for  
  
    que dentro lleva otro bucle for
```

Pero ¿y si dentro necesita un if?

```
bucle for  
  
    que dentro lleva otro bucle for  
  
        el if que se necesita dentro
```

¿Y qué ocurre si dentro del if hay switch?

```
bucle for  
  
    que dentro lleva otro bucle for  
  
        el if que se necesita dentro  
  
            aquí empieza el switch...
```



Pues ya vemos. El anidamiento de estructuras nos lleva un código que cada vez se hace más difícil de leer.

Así que ese va a ser nuestro primer límite. Máximo dos niveles de anidamiento.

-  1\*  \_niveles de anidamiento\*

Tampoco es agradable encontrarse una estructura, da igual un if que un for, rellena de docenas de líneas. Cuando termina la estructura, te preguntas ¿a qué venía yo aquí?

Así que ahí te va el segundo límite: no metas más de 8 líneas dentro de un bloque for o un ramo de un if. Idealmente no pases de 4. A partir de ahí, crea una función e invócala.

-  4\*  \_instrucciones por bloque\*

Por último algo que ya dije al principio. Mantén simples las condiciones y esconde la complejidad en funciones.



-  1\*  \_operadores lógicos por condición\*

Para cumplir estos límites

Te obligas a extraer código a funciones.

Te obligas a nombrar las nuevas funciones.

## Consecuencias

-  Más **reglas de negocio** descritas en las funciones
-  **Cero** necesidad de comentarios

## 🌀 2.3 - Funciones puras y métodos de clase

Pequeñas piezas para organizar programas.

*"Una función debería hacer una sola cosa, hacerla bien, y hacerla sólo ella."*

-- 📌 **Ley de Curly.**

Esta frase para enmarcar encierra la verdad esencial de este curso tutorial. **Las funciones son las piezas fundamentales de construcción de las aplicaciones** limpias. ¿Por qué? Porque son el conjunto mínimo de instrucciones que se le puede asignar un nombre y que se pueden reutilizar.

Es decir, **claridad y reutilización en un mismo artefacto**. Normal que nos inciten a prestarle toda la atención, hasta conseguir los tres mandatos:

- **Hacer una sola cosa:** Un único propósito especificado en su nombre
- **Hacerla bien:** Provista de test o al menos con facilidad para la prueba
- **Hacerla sólo ella:** Nombrarla y situarla de forma que no se duplique su cometido accidentalmente

Para conseguirlo podemos desgranar una serie de consejos y límites aplicables todas nuestras funciones.

### ✂️ Pequeñas y Claras

- 🗑️ Cuanto más pequeñas más reutilizables.
- 🗣️ Con **verbos** en su nombre que indiquen propósito
- 🙅 DRY: Don't Repeat yourself.
- 😊 con valores por defecto en sus argumentos si el lenguaje los soporta.
- 😊 sin condiciones complejas.
- 🚩 ...sin flags: crea dos variantes con nombre específico.
- 💬 ...sin comentarios. El nombre es el mejor comentario.

### ⚠️ Límites

- ✅ 0\*\*\_\*\*0 ❌ *flags*
- ✅ 1\*\*\_\*\*2 ❌ *argumentos*
- ✅ 8\_\_12 ❌ *\_complejidad ciclomática*
- ✅ 16\*\_24 ❌ *\_instrucciones\**

### 💧 Favorece el estilo funcional puro:

*En una **función pura** el valor de retorno solo está determinado por sus valores de entrada, sin efectos secundarios observables.*

-- 📌 **Alguien a quien le gustan las matemáticas.**

*Disclaimer:* Puede que el repentino auge de la programación funcional te haga dudar de si esto es una cuestión de modas. No, no lo es. Los paradigmas de programación son clásicos y se deben aplicar consciente y coherentemente. Por supuesto que los lenguajes te predisponen en mayor o menor medida hacia la programación funcional, imperativa o la orientada a objetos.



Pero este principio de pureza, obligatorio en programación funcional, es la antítesis de la globalización; y por tanto es una guía incluso en la programación con objetos.

- 🌙 **Predecibles.**
  - Ante la misma entrada,
  - deben producir la misma salida.
- 🏠 **Sin dependencias del entorno.**
  - Sus argumentos son su materia prima
  - y su maquinaria.
- ⓘ **Sin efectos secundarios en el entorno.**
  - No deben manipular variables externas
  - ni utilizar sistemas externos

Obviamente no todas tus funciones puede ser puras. La idea es que separes unas de otras y favorezcas **que la lógica resida en funciones puras.**

## 📁 Métodos en P.O.O.:


En un **método de clase** deberíamos trabajar mucho con el resto de propiedades de la clase y depender poco del exterior.

-- 📝 **Alguien a quien le gustan la encapsulación.**

En el paradigma de **Programación Orientada a Objetos**, a la función se le llama método. Y su entorno de trabajo se circunscribe a la clase en la que se define.





Los consejos y límites recomendados pueden no valer para otros paradigmas y son los siguientes:

- [0] **cuantos menos argumentos mejor.**
  - 🚩 evita argumentos *flag* usando múltiples funciones específicas.
  - favorece objetos en lugar de primitivos.
  - los argumentos en métodos públicos son señal de dependencia exterior.
- [1] **un mismo nivel de abstracción: delega en funciones privadas**
  - las instrucciones en funciones públicas deberían llamar a funciones privadas.
  - si un método tiene muchas instrucciones, es que tienen muchas responsabilidades
  - debe delegarlas en otros métodos de ayuda

-  **retornando datos; nunca errores.**
  - los errores tienen su propio flujo mediante `try-catch throw`.
  - si el lenguaje no lo permitiese, usar convenio en los argumentos
    - como los viejos *callbacks* (`err, data`).

## Objetivo: Muchas Pequeñas Funciones Organizadas

Nuestro reto es conseguir grandes aplicaciones a partir de muchas, muchísimas, funciones pequeñas. Y para ello es crucial mantener un orden y una organización que permitan **encontrar y no duplicar el conocimiento** que encierran.

-  Una función,
  -  un sólo propósito.
  - ... o al menos un mismo nivel de abstracción.
  - claramente definido en su nombre
-  Sin comentarios.
  - ¿Me repito?. MAL!!! 

*"Una función debería hacer una sola cosa, hacerla bien, y hacerla sólo ella".*

--  **Ley de Curly**



Como colofón a esta primera parte del curso tutorial te dejo esta máxima de *Uncle Bob*. Trata de cumplirla manteniendo las reglas de claridad y modularidad con **muchas funciones pequeñas bien nombradas y organizadas**.

*"La duplicidad es el principal enemigo de un sistema bien diseñado"*

--  **Robert C. Martin**