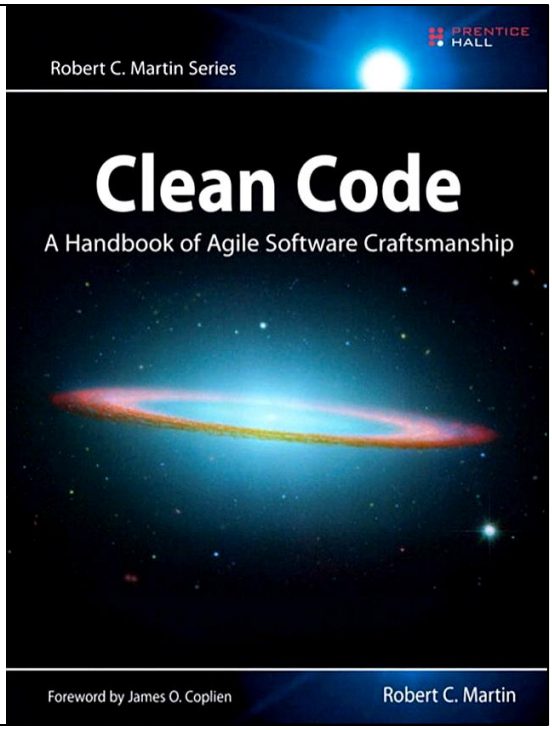
Código limpio

José Galaviz



¿Qué es código limpio?

Ante la pregunta de ¿Qué es código limpio? Formulada por Robert C. Martin, las respuestas fueron muchas y muy variadas entre la comunidad de reconocidos desarrolladores.

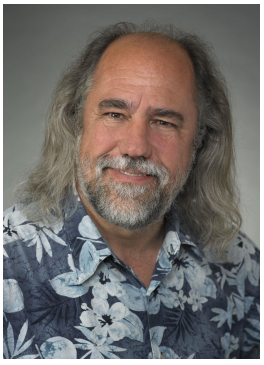
- Simple, directo, legible (Grady Booch).
- Elegante, eficiente (Bjarne Stroustrup).
- Legible (Dave Thomas).
- Predecible (Ward Cunningham).
- Hecho con cuidado (Michael Feathers).
- Sin duplicidades, breve, expresivo, simple (Ron Jeffries).

Elegante, eficiente

(Bjarne Stroustrup)

Simple, directo,

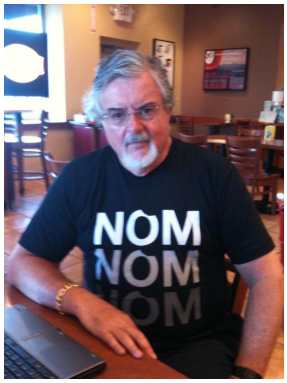
legible (Grady Booch)



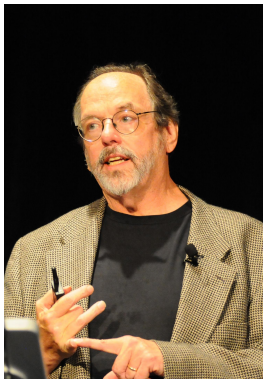
Legible (Dave Thomas)

Hecho con cuidado

(Michael Feathers)



Sin duplicidades, breve, expresivo, simple (Ron Jeffries)



Predecible (Ward Cunningham)

¿Cuáles deberían ser las normas de conducta o las actividades y actitudes correctas para lograr código limpio?
Se ha desarrollado toda una metodología basada en el concepto llamado de *las 5 s*. Originalmente formulado para hacer efectivos los procesos de manufactura en la industria, pero generalizables a otros rubros. En particular, al que nos interesa.

Seiri

Seleccionar

Deshacerse de lo innecesario

Seiri. Seleccionar, clasificar y discriminar lo necesario de lo innecesario, deshacerse de esto último. Es indispensable para tener código limpio no tener código inútil alrededor. Las rutinas que ya no se usaron, los fragmentos que terminaron siendo reemplazados por algo mejor o que ya no son necesarios deben eliminarse. No deben estar allí. Si te preocupa borrarlos porque hay trabajo invertido en ellos y podrían servir en algún momento, usa software para control de versiones y borralos.

Seiton

Acomodar para que todo fluya naturalmente

Seiton. Organizar, ordenar las cosas de tal forma que nada estorbe a nada, permitir que las cosas fluyan naturalmente, de manera óptima. Ya que tienes puras cosas útiles en el código, organiza las que quedaron para sea claro el código, reacomoda las cosas relacionadas cerca. Que la lectura de un segmento no se vea interrumpida por algo que no tiene nada que ver.

Seisou. Limpieza, comprometerse con no “ensuciar” (romper el diseño) el código, no permitir que haya partes desorganizadas o mal hechas. No hay que dejar que lo que hicimos ayer tan bien, hoy se eche a perder por las prisas, no ensucies el código limpio, no hagas rutinas poco cuidadas como un parche para que funcionen las cosas aunque no se vean tan bien. Luego de cada sesión limpia el código, reorganiza.

Seiketsu

Corregir

Encontrar anomalías, reparar

Seiketsu. Corregir (literalmente limpiar). Encontrar las anomalías, los errores, apegarse a estándares que prevean errores. Corrige lo que veas mal en cuanto te topes con ello. Depura cada parte lo más rápido posible antes de que olvides qué debe hacer. No dejes errores latentes.

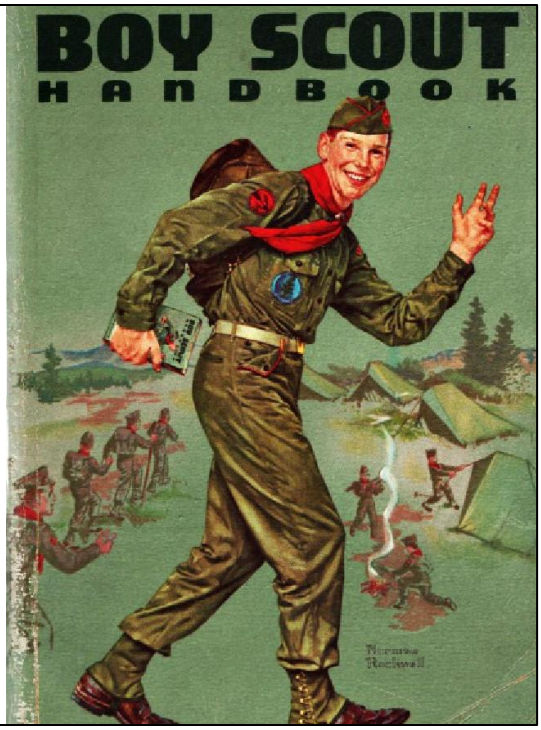
Shitsuke. Disciplina, autocontrol, no cejar en el empeño, no relajarse por cansancio o flojera en el apego a los principios previos. Se disciplinado, mantén el esfuerzo permanentemente a lo largo del desarrollo del proyecto. No aflojes las normas conforme se acerca el plazo y aumenta el cansancio.

En *The Pragmatic Programmer: From Journeyman to Master*, Andrew Hunt y David Thomas hacen una analogía entre el software y un edificio. Un factor que acelera el deterioro de un edificio es que este posea una ventana rota, luego de un tiempo hay más ventanas rotas, aparece el grafitti, las personas ensucian el interior, hay roturas en la fachada y finalmente el deterioro es tan grande que el edificio debe ser demolido. Una ventana rota lleva a pensar que el edificio no importa. Es un síntoma de abandono que conlleva a un abandono psicológico.

Cuando vemos un fragmento de código mal cuidado, que, aunque funciona, está mal hecho, está sucio o desordenado, tendemos involuntariamente a pensar que *no importa,* que “pues ya qué” y entonces las cosas van de mal en peor.

Principio Scout

Cuando acampes, deja el sitio mejor que como lo encontraste



En ese sentido es mejor aplicar el principio usado por los *Boy Scouts*: Cuando acampes en un sitio, déjalo mejor que como lo encontraste. Si puedes mejorar en algo el código que recibes (de tí mismo en el pasado o de otros programadores) **¡Hazlo!**

Elementos del
código limpio

Nombres

El nombre de una variable, una función (método) o clase, idealmente debe responder preguntas.

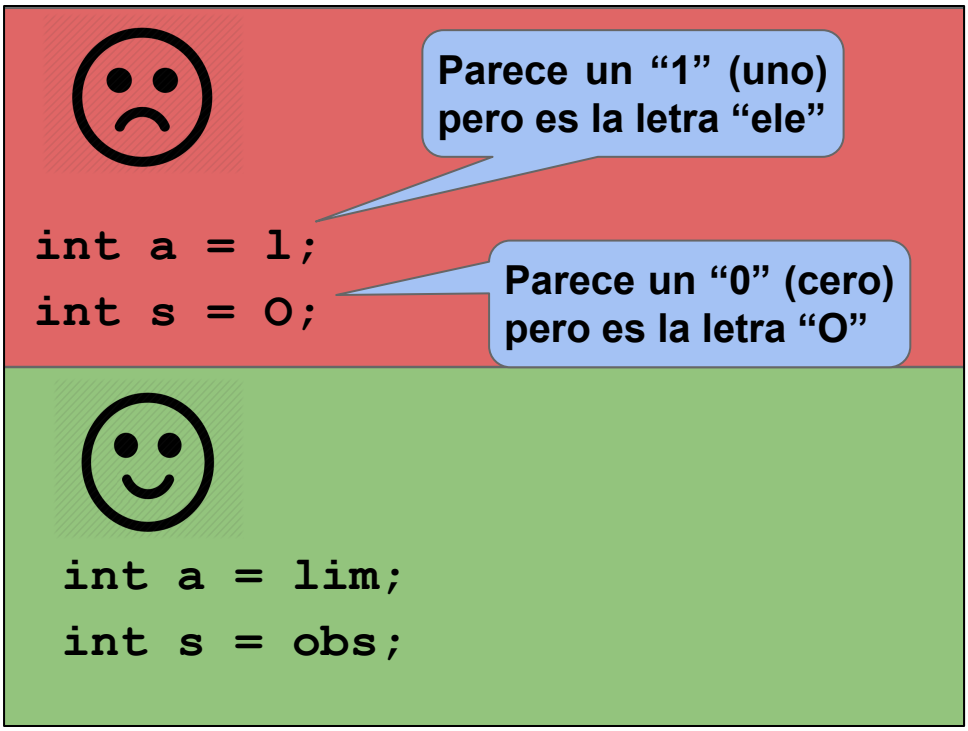
¿Para qué sirve?

¿Cómo se usa?

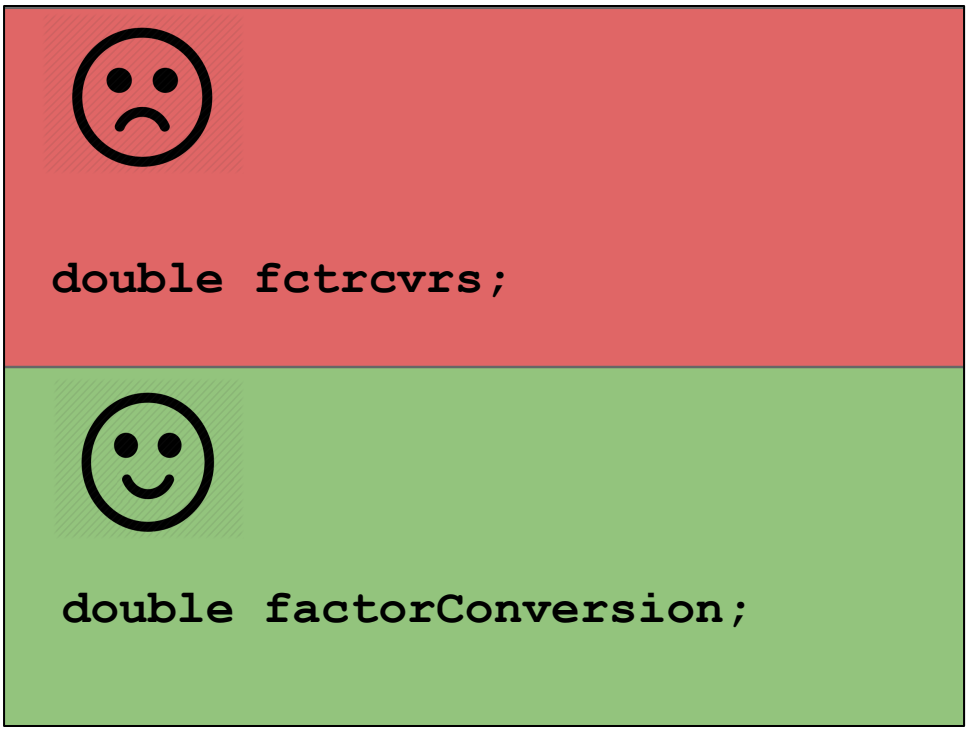
Para favorecer la legibilidad del código que escribimos le ponemos a las variables, los métodos y las clases nombres que digan algo acerca de ellas. En particular, el nombre debe ser un indicador de “para qué” sirve la entidad en cuestión y/o “cómo se usa”.

```
int d; // tiempo transcurrido
```

```
int diasTranscurridos;
```

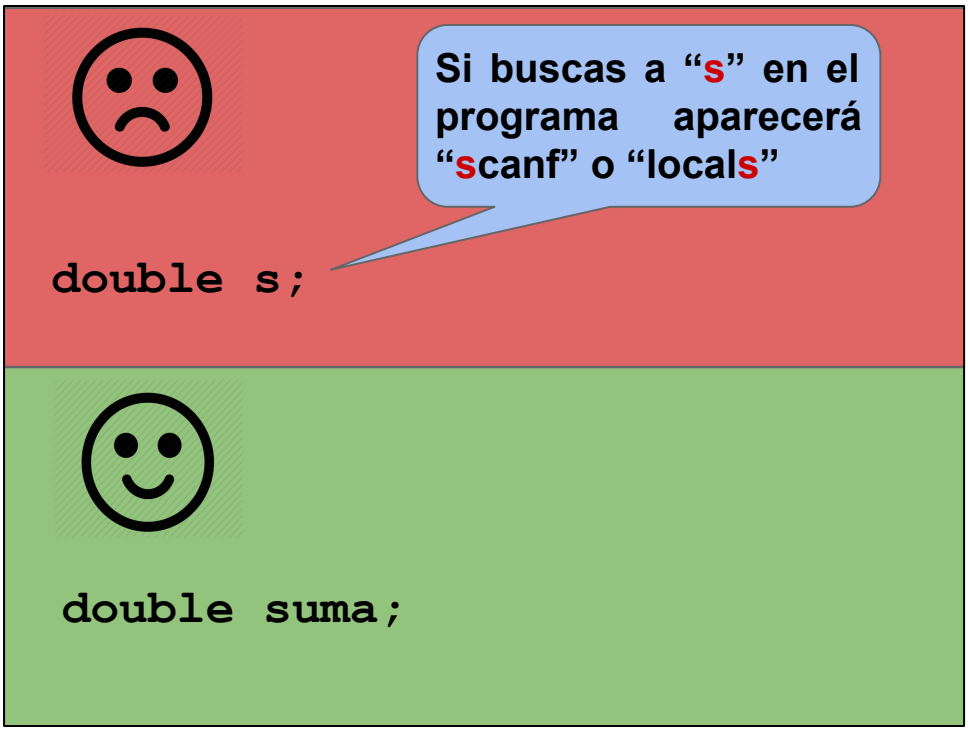


No desinformar con los nombres de las variables. En el ejemplo, la variable “l” (*ele*, la letra del alfabeto) se parece mucho al “1” (*uno,* el número) así que uno no sabe, a primera vista, cuál es el valor que se asigna a la variable “a”. En general evitar usar nombres cuyo significado intrínseco o uso común hará que se entienda algo diferente de lo que en realidad significan en el programa. No usar una variable llamada perro para referirse a un gato.

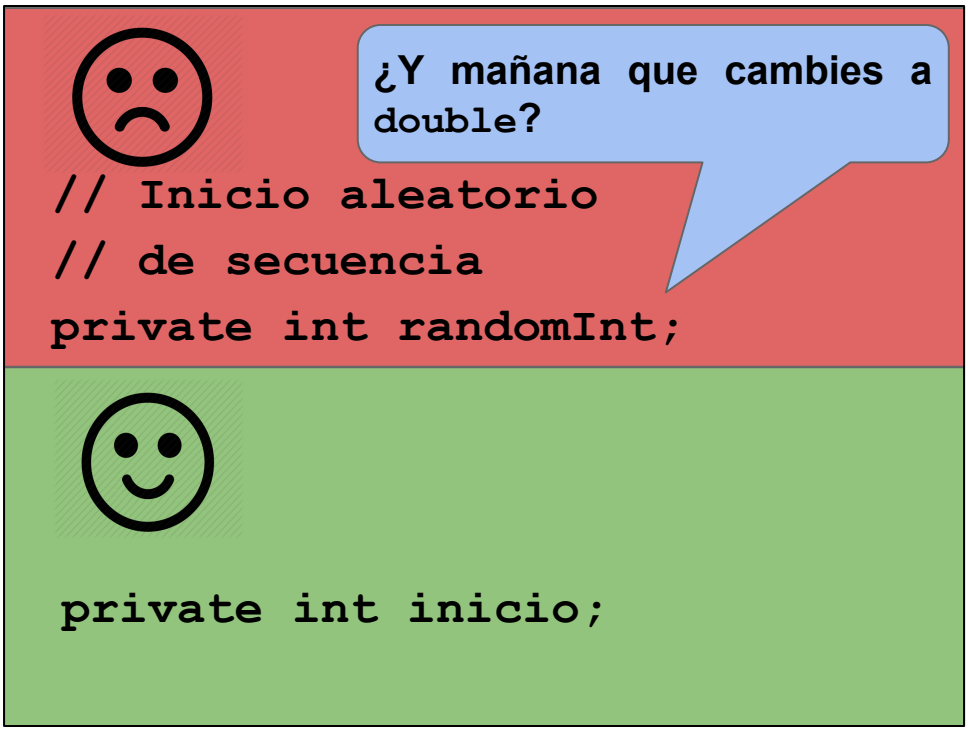


Usar nombres pronunciables. Porque llegará el momento en que alguien más trate de entender tu programa o bien a alguien tratarás de explicarlo y será conveniente que puedas referirte a las variables con palabras que puedas pronunciar.

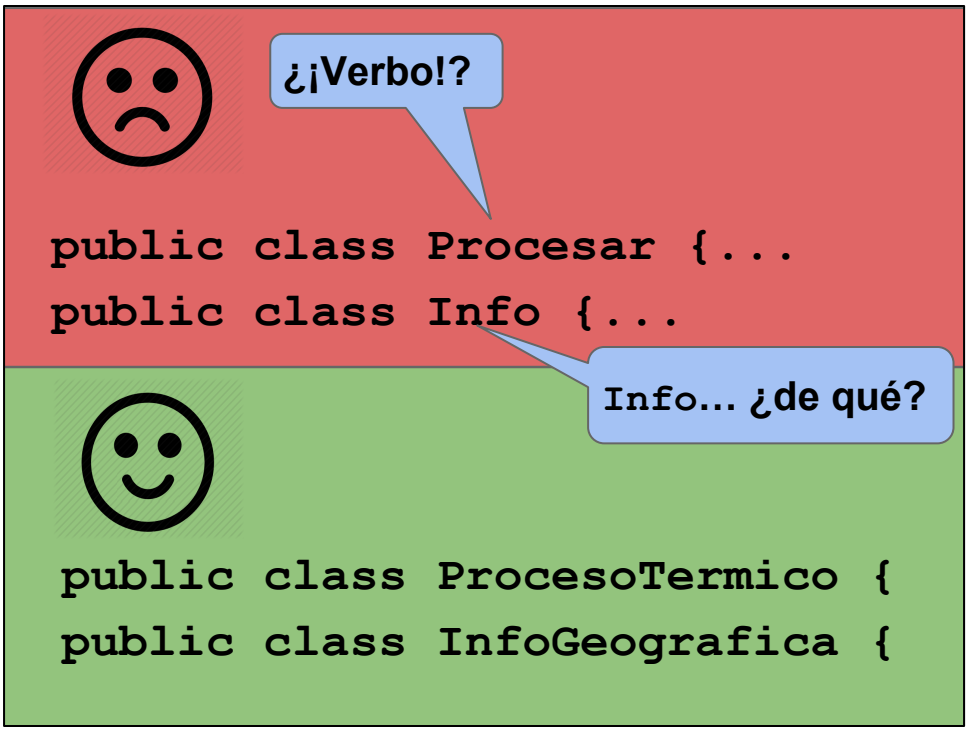
Hay que procurar que dos identificadores cualesquiera difieran en, al menos, dos caracteres y al menos uno de ellos en un extremo.



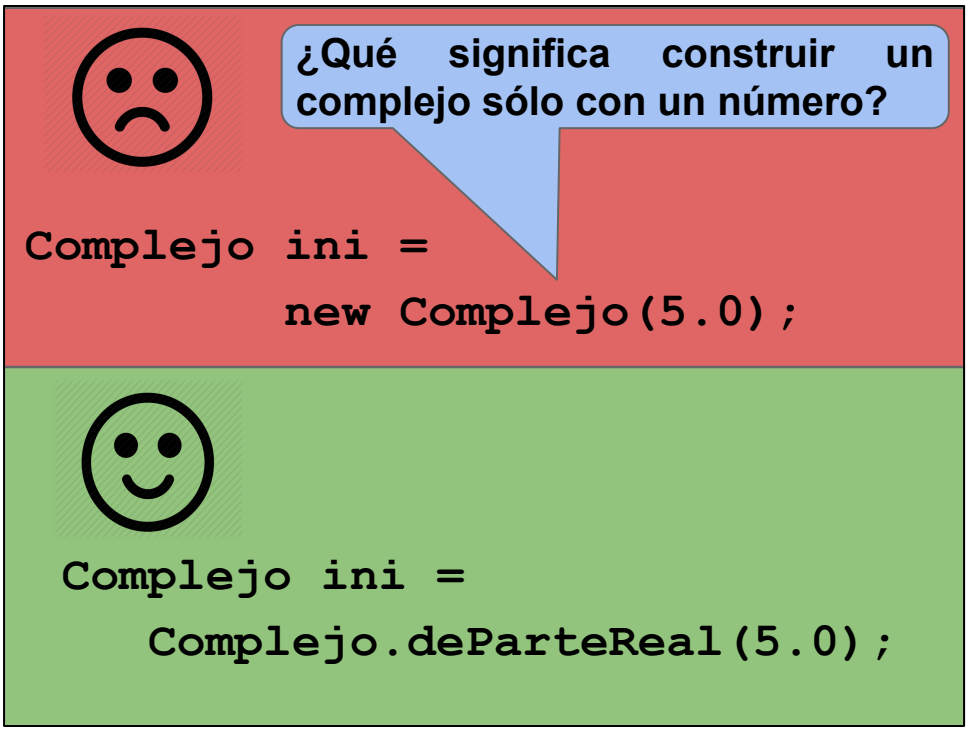
Elegir nombres que puedan ser buscados como cadenas después. ¿Cuantas veces no queremos encontrar donde se usa una cierta variable y aparecen muchas tonterías que nada tienen que ver?



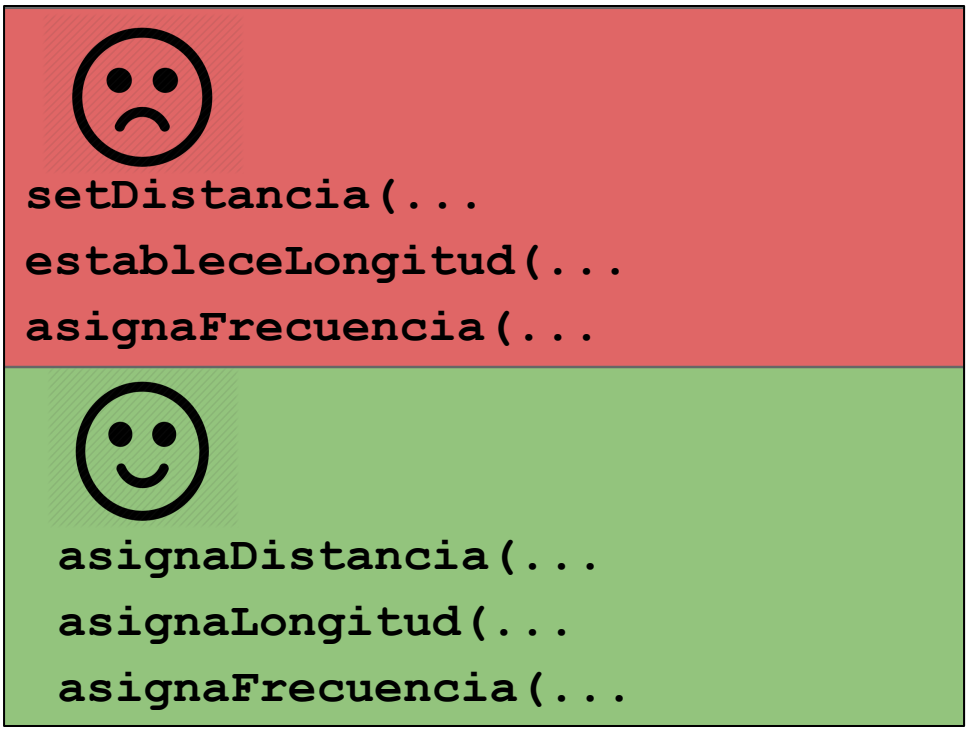
Evitar lo que se conoce como *notación Húngara*, cuando de alguna manera, el identificador de una variable hace referencia al tipo de dato que la variable almacena o refiere. No es bueno porque es frecuente que el programa se modifique, se adapte o se corrija y habrá que cambiar el tipo de la variable y entonces su nombre sugerirá algo falso, desinformará al lector.



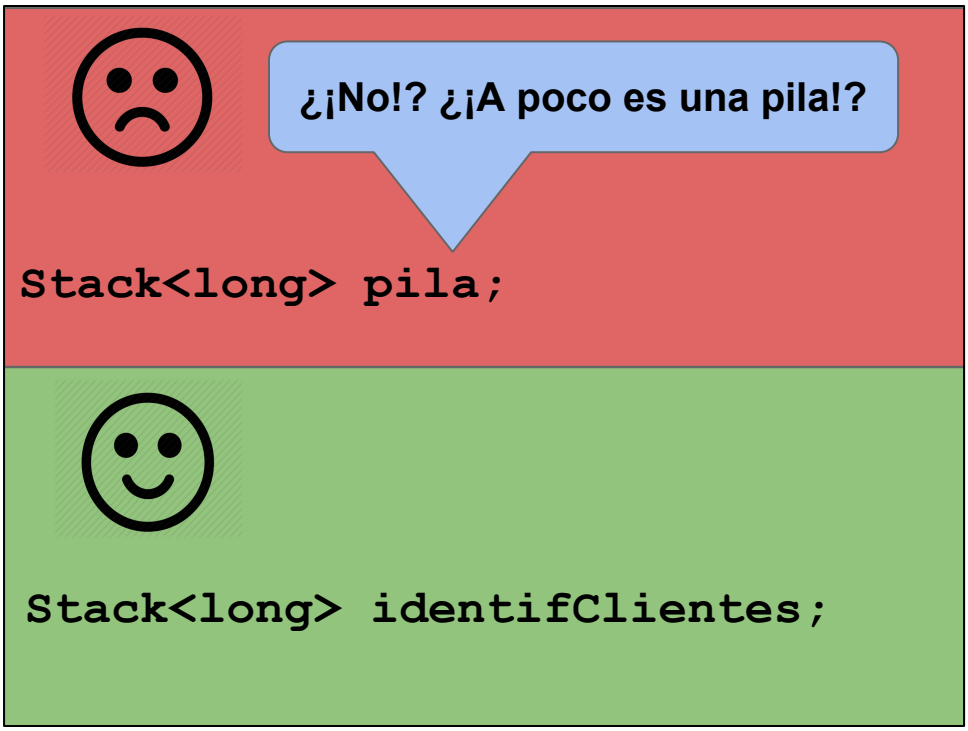
Los nombres de clases no deben ser verbos sino sustantivos. Al contrario, los de funciones o métodos sí deben ser verbos o un verbo antecediendo un sustantivo: `imprime`, `marca`, `enviaMensaje`, `encuentraPaquete` . Los nombres deben ser significativos, una clase llamada `Info` no dice nada a quien la lee acerca de cuál es su propósito. Un método o función que verifica el estado de algo y regresa un booleano (aunque sea sólo semánticamente como en C) debería tener un nombre que parezca pregunta: `esVacio`, `esSolucion`, `encontroCadena` .



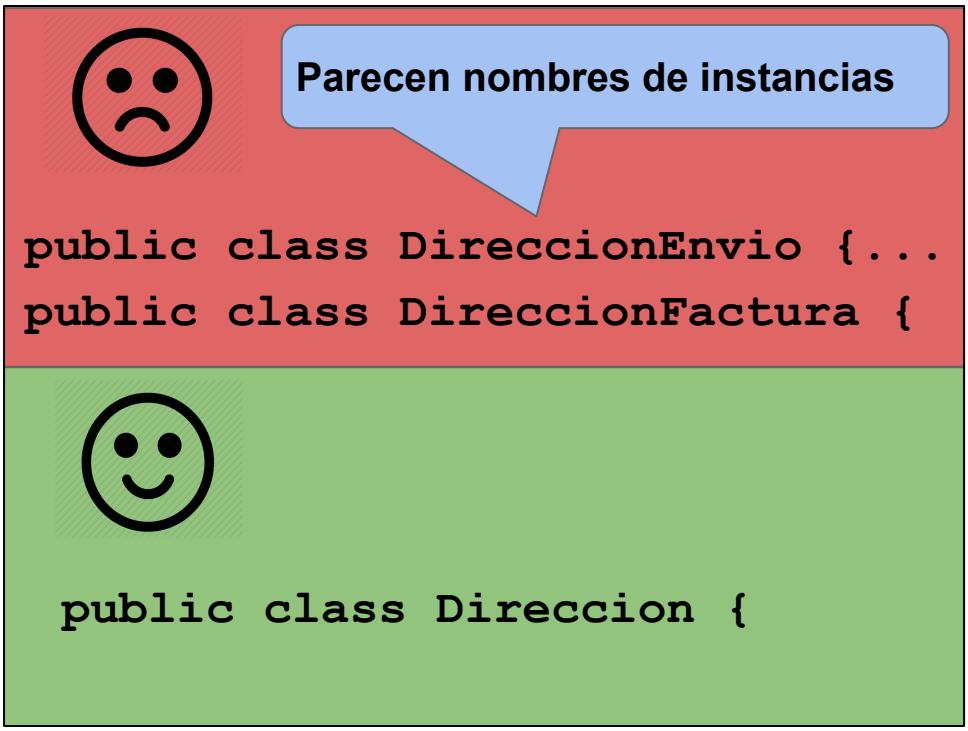
Si un constructor realiza su labor sólo con información parcial y asigna valores por omisión a las variables de estado restantes, sería bueno no llamarlo como a los constructores que sí reciben todos los datos necesarios.



Usar consistentemente la misma palabra para el mismo tipo de cosa.



Los lectores del programa son programadores, más les vale saber lo necesario. Los nombres deben referirse al dominio del problema.



Ser lo más general posible en los nombres asignados a las abstracciones que se pretenden reutilizar.

Rutinas, funciones, métodos, procedimientos

Los métodos, procedimientos, funciones, lo que en general se conoce como rutinas o subrutinas en los lenguajes de programación, deben, en la medida de lo posible, ser breves. No hay un criterio generalizado, pero Martin propone que sean:

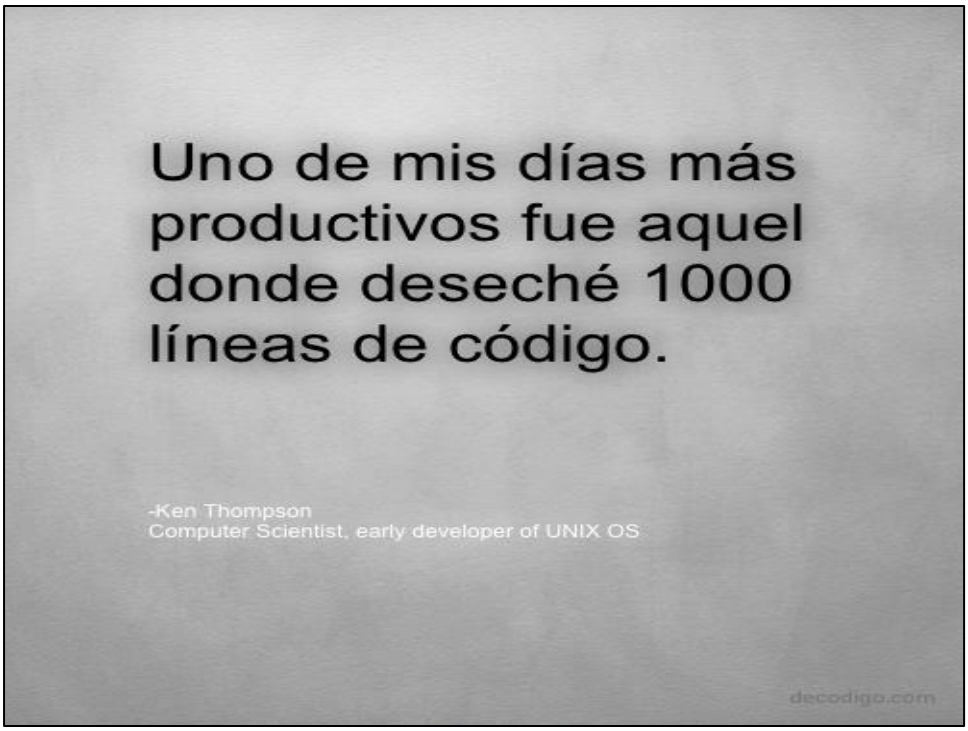
- De menos de 20 líneas de código.
- Con líneas de menos de 150 caracteres.

Lo que por cierto contraviene otras muchas recomendaciones en las que el largo de una línea debe estar acotado por 80 caracteres, que era lo que cabía en la pantalla de muchas terminales (VT100, por ejemplo) y en muchas pantallas de las viejas PC. No es sin embargo una recomendación vieja, en las convenciones de Java se especifican también 80 caracteres.

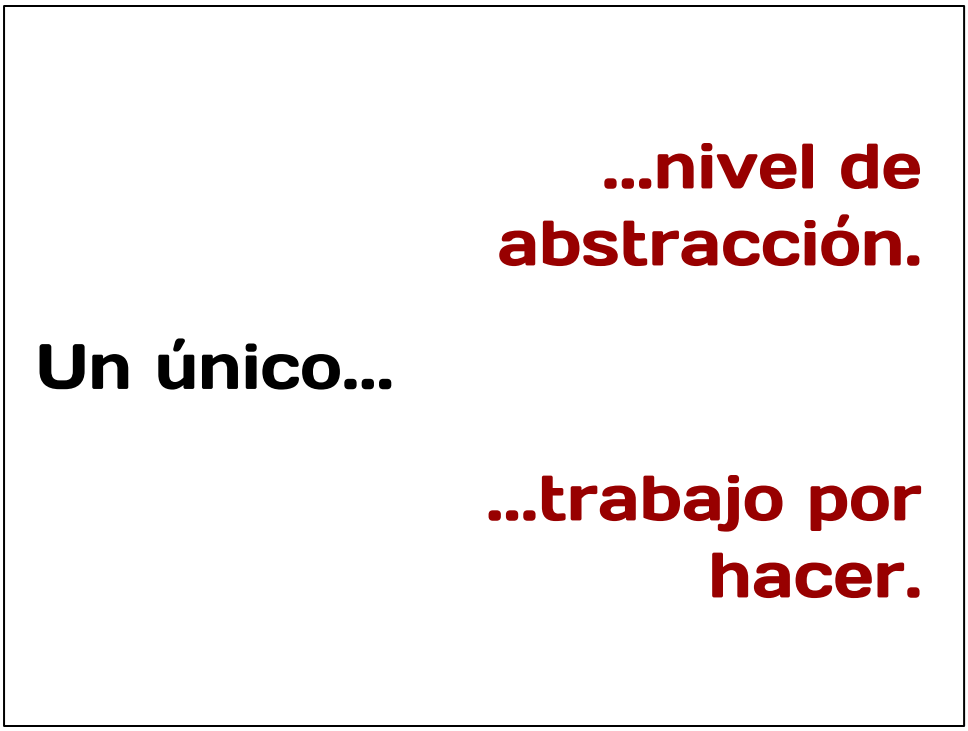
En general, alrededor de 80 caracteres es una longitud aceptable tanto para leer sin perder el hilo de lo que se hace, como para estipular una instrucción en la mayoría de los lenguajes de programación. Podríamos recomendar entonces, tratando de mediar.

- Líneas de nunca más de 100 y preferentemente, menos de 80 caracteres.
- De no más de 30 y preferentemente menos de 20 líneas de código.

Apostando por que, sí la rutina no satisface estos criterios, muy probablemente pueda partirse de modo que el resultado sí satisfaga.


Uno de mis días más productivos fue aquel donde deseché 1000 líneas de código.
-Ken Thompson
Computer Scientist, early developer of UNIX OS
decodigo.com

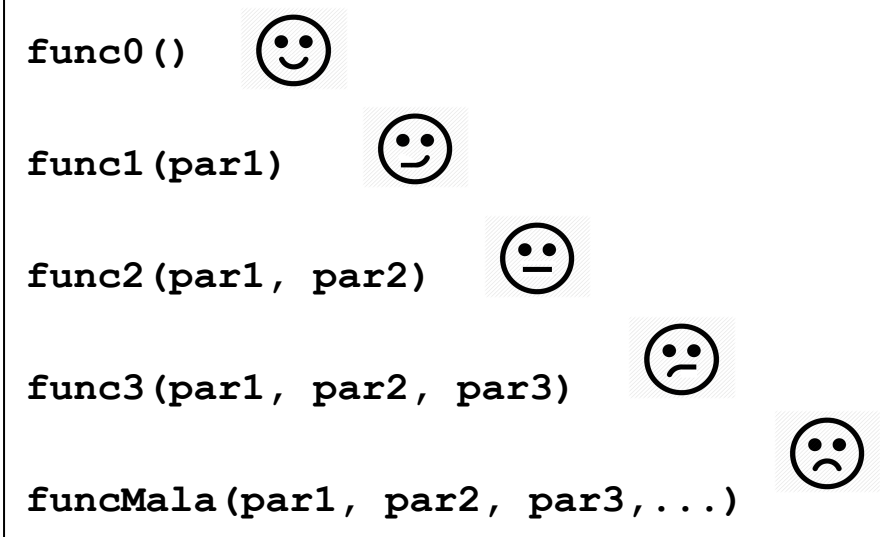
Leer la secuencia de llamadas a las rutinas o métodos que integran el programa debería ser como leer el algoritmo elegido para resolver el problema. Como la narración de la historia que culmina con la solución del problema. Claro, habría que decir que esto debe ocurrir en cada uno de los niveles de abstracción y en cada uno de los módulos en los que se ha descompuesto el sistema, es decir, tanto localmente como en general.



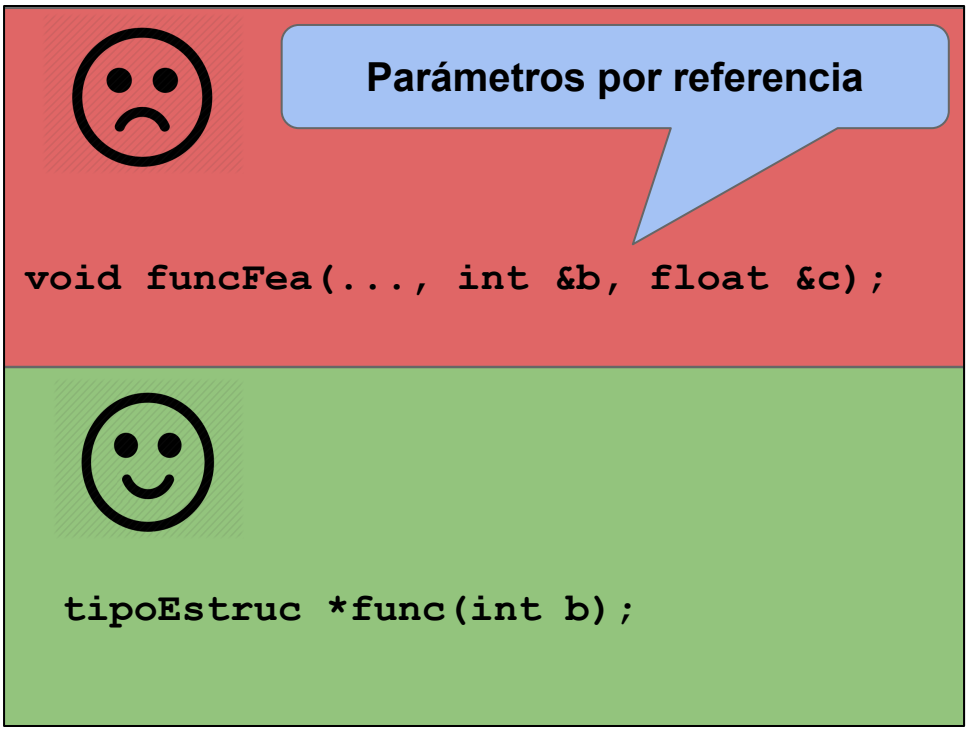
Así que es particularmente importante el mantener, en cada subrutina, un cierto nivel de abstracción fijo. No es aceptable, por ejemplo, leer un objeto de un TDA “Persona” haciendo una llamada a un método que lo hace y luego poner el código que verifica sí un cierto campo está en mayúsculas. Eso pertenece a un nivel de abstracción inferior.

Por supuesto, lo ya mencionado anteriormente, cada método o rutina debe hacer UNA ÚNICA cosa y hacerla completa (de acuerdo con el nivel de abstracción) y bien (debe probarse).

Parámetros



Deben tener pocos parámetros. Idealmente ninguno, pero es difícil, en todo caso hay que procurar mantener el número de parámetros por debajo de tres. Si se requieren más entonces deberían agruparse en una estructura para tal efecto o bien partir la rutina en dos o más partes. Si se necesitan muchos parámetros probablemente sea porque estamos pretendiendo hacer demasiadas cosas.



No se deben usar, en lo posible, parámetros pasados por referencia. Si la función o método entrega un resultado DEBE ser UN resultado.
Tampoco debe haber efectos colaterales: una variable global (que espero no se usen) modificada, el cambio de estado de otro objeto, etc.

Asimismo es importante no usar lo que se conoce como *argumentos de bandera,* o *flag arguments*, que son aquellos que al ser pasados como parámetros a la rutina hacen que esta se comporte de una forma o de otra dependiendo de su valor.

La formula especificada por Esdger Disjkstra desde los tiempos de la programación estructurada sigue siendo importante: toda rutina debe tener una sola entrada y una sola salida. Es decir, en lo posible, se debe evitar tener varios `return` en diferentes lugares del código de la rutina. En la medida de lo posible también debe tener pocos `if…else`.

Comentarios

Es lo primero que se debe escribir de toda rutina.

Frases de hacker

parcialmente
~~completamente~~ falsas

“El buen código es su propia documentación”

Es cierto en el sentido de que un código bien escrito se entiende *per se* sin necesidad de comentarlo. Pero no es cierto que el buen código no necesite comentarse; las instrucciones de uso de una rutina, el significado y los valores de sus parámetros, sus precondiciones y poscondiciones, son cosas que se deben poner en los comentarios.

“La inspiración es inversamente proporcional a la cantidad de comentarios”

Es cierta si pensamos en los comentarios que pretenden suplir las deficiencias del código mal hecho, tan malo que sabemos que no se entenderá y entonces lo comentamos profusamente tratando de hacerlo inteligible. Es falsa si pensamos en aquellas rutinas que requieren hacer explícito su contexto y su operación con lo que frecuentemente las líneas de comentarios igualan o exceden a las de código.

Pero en términos generales debería ser más sencillo comprender el código aún sin comentarios. Así que estos deben ser claves indispensables, breves.

Los indispensables

- En las rutinas: qué hace, parámetros, precondiciones, postcondiciones, valores de regreso.
- Términos de uso, licencia, datos del programador.
- Documentación de bibliotecas o interfaces de programación (API).

Los buenos

- Los que explican la parte esencial del algoritmo o la representación de datos.
- Los que clarifican cuando la sintaxis no ayuda (compareTo).
- Los que previenen consecuencias (“esto no es *thread safe*”).
- Los que dicen que falta por hacer.

Los malos

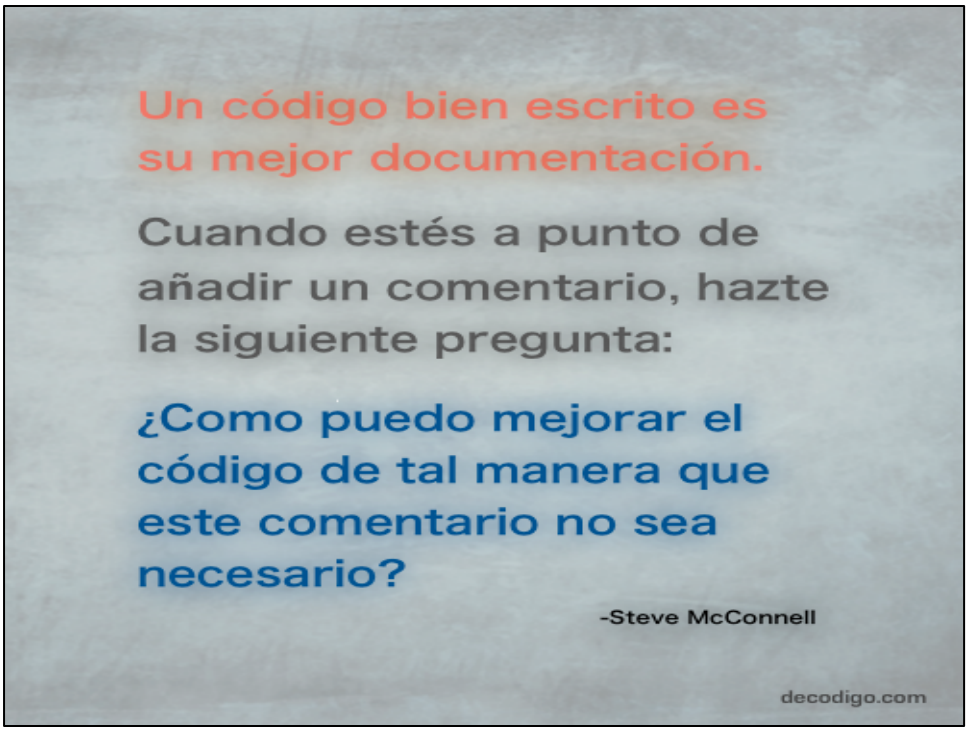
- Los que se ponen al cerrar una llave (en C, por ejemplo).
- Los de autojustificación.
- Los redundantes.
- Marcadores de posición, bitácora, código erróneo. En general los que son buenos sólo durante pruebas.

Nunca usarlos para:

- Subsanar código mal hecho.
- Eliminar código (usa control de versiones).
- Explicar algo que puedes evidenciar con un buen nombre (de rutina o variable).

La idea general de un comentario es preparar la mente del lector para comprender el código que sigue, para interpretarlo de la manera correcta.

Los comentarios correctos del código incorrecto hacen más difícil comprender y depurar el programa.


Un código bien escrito es su mejor documentación.
Cuando estés a punto de añadir un comentario, hazte la siguiente pregunta:
¿Como puedo mejorar el código de tal manera que este comentario no sea necesario?
-Steve McConnell
decodigo.com

Fin

Objetos

Ocultar información innecesaria.
Proveer servicios claros.

Estructuras de datos

La más adecuada.

Manejo de errores

Escribir código robusto.

El manejo de errores debe ser un tema per se.

Usar excepciones en vez de códigos de error de regreso.

Usar excepciones no-verificables.

No regreses NULL, mejor lanza una excepción o regresa un objeto de significado especial.

No pases NULL como parámetro. De hecho prohibelo en tu código.

Pruebas

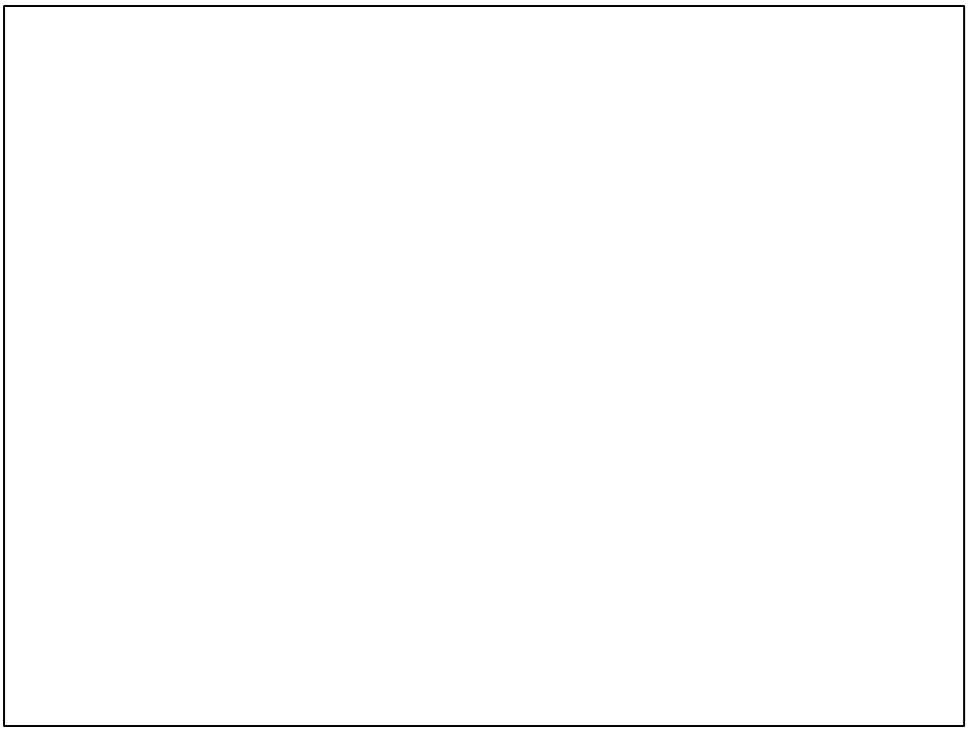
El código y las pruebas deben escribirse al mismo tiempo.

Mantener limpias las pruebas.

El código de prueba es tan importante como el de desarrollo.

Pruebas unitarias

F ast
I ndependent
R epeatable
S elf-validation
T imly



Elementos

Nombres significativos.
Comentarios útiles.
Funciones simples.
Objetos y estructuras de datos.
Manejo de errores.
Pruebas unitarias.