**Manual Buenas Practicas – Código Limpio**

Arbusta

CAPÍTULO 1 – CÓDIGO LIMPIO

Para escribir código limpio no solo es necesario conocer patrones y estilos de diseño de software y saberse la teoría, hay que practicarlo, hay que fallar y volver a practicarlo, solo así se podrá escribir buen código.

El objetivo del libro es aprender a diferenciar entre código malo y código bueno, y poder transformar código malo en bueno.

Hay quien piensa que en algún momento de nuestras vidas el código desaparecerá, que

simplemente tendremos requisitos y las máquinas los harán por nosotros, nada más lejos de la realidad, esto es improbable que suceda, dado que no existe máquina que pueda entendernos al 100% o incluso descifrar las ambigüedades de muchos requisitos. Podemos crear lenguajes que entiendan casi todos los requisitos, pero siempre será necesaria una precisión exacta que solo se puede dar programando

La única forma de alcanzar los plazos de entrega es escribiendo buen código, y esto lo

saben los grandes profesionales. Escribir código limpio y correcto es como pintar un cuadro, cualquiera puede apreciar la calidad de un cuadro, pero no por saber que está bien pintado significa que sepamos pintar.

El código limpio debe ser elegante, simple, directo, fácil de leer como un libro y de mantener, que contenga test unitarios y de aceptación, el código debe ser de un tamaño mínimo, el código limpio siempre parece que ha sido escrito por alguien que le ha importado el código, el código debe ser al leerse lo que esperamos que

sea, ¿cuándo fue la última vez que leíste un código y era lo que esperabas, o no te supuso esfuerzo comprenderlo?, son varias de las conclusiones a las que han llegado lxs grandes profesionales de software sobre lo que debe ser el código limpio.

Recuerde que cuando escribes código eres el autor de un código que tendrá lectores, pasamos mucho más tiempo leyendo código que escribiendo código en nuestras vidas.

CAPÍTULO 2 - NOMBRES CON SENTIDO

Los nombres están en todas partes, clases, variables, ficheros, etc. Elegir nombres relevantes es de suma importancia, y aunque pareciera algo de sentido común, ni es tan

común ni es tan sencillo.

Cada vez que encuentre un nombre de alguna variable que se puede mejorar, hagalo, la

gente que lea su código se lo agradecerá. Los nombres deben revelar nuestras intenciones.

Por ejemplo, si definimos una variable de tipo entero con el nombre ‘d’, para referirnos a los días que han pasado desde determinada fecha, sería incorrecto, ‘d’ no significa nada en absoluto para cualquier persona que lea el código, en su lugar algo más conveniente sería ‘daysSinceCreation’ por ejemplo.

Evite usar por ejemplo abreviaciones, como hp para hipotenusa, puede parecer correcto pero no lo es, no haga referencia a un grupo de cuentas como *accountList* porque si luego el tipo de dato cambia y no es una lista puede confundir, lo mejor sería usar *accounts* o *accountsGroup* .

No utilice tampoco nombres que se diferencian mínimamente, por ejemplo entre *XYZAlmacenamientoDeCuentas* y *XYZManejadorDeCuentas* existe una sutil diferencia que nos puede confundir o hacernos equivocar en un módulo.

Es importante utilizar nombres que se puedan pronunciar con facilidad, es un factor

importante, dado que la programación es una actividad social, y es importante comunicarse con facilidad. Había una empresa que utilizaba el nombre *genyhmds* para referirse a la generación de una fecha, año, mes, día, hora, minuto y segundo. Al final la gente acabó por pronunciarlo cómo ‘genimedemes’, pero cada vez que llegaba un nuevo programador, lo pronunciaba como le parecía, al final resultaba confuso, sería mucho más sencillo utilizar *generationTimestamp* , por ejemplo. De la misma forma, se recomienda que los nombres sean sencillos de buscar, si colocamos a una variable el nombre ‘e’, resultará muy complicado buscarlo por todo el código, pues resulta que la letra ‘e’ es la más usada en el inglés.

Los nombres de las clases deben ser nombres, nunca verbos. Por su parte, los métodos deben tener nombres de verbo, y si son modificadores utilizar get o set, como marcan los estándares

No haga juegos de palabras, imagine que tiene un método llamado ‘add’ que concatena dos valores, y está presente en varias clases, bien, pero ahora resulta que alguien quiere utilizar el nombre ‘add’ para un método que inserta un valor en una lista, puede parecer correcto, pero no lo es, porque ‘add’, ya deja de tener la misma funcionalidad que el resto de métodos, lo ideal sería llamarlo ‘insert’, por ejemplo.

Debe añadir contexto con sentido, por ejemplo las variables ‘firstName’, ‘lastName’,

‘zipcode’, ‘street’, ‘state’, podrían indicarnos que pertenecen a una dirección, pero sí ‘state’ se usará por separado podría llegar a la misma conclusión, en este caso podríamos usar algo como *addressFirstName* , *addressLastName* , etc. Aunque la mejor solución es incluir las variables en una clase llamada Address, así le damos el contexto necesario.

Lo más complicado a la hora de elegir un nombre es que se requiere habilidad descriptiva y conocimiento del lenguaje. Mucha gente tiene miedo que a la hora de cambiar los nombres otras personas se quejen, esto no es así, siempre que sean a mejor. Hágalo y obtendrá grandes resultados a corto y largo plazo.

CAPÍTULO 3 – FUNCIONES

La primera regla de la creación de funciones es que su tamaño debe ser reducido. En la década de los 80, en las pantallas cabían unas 24 líneas y 80 caracteres, y hoy en día pueden caber 100 líneas y 150 caracteres, lo correcto es que las funciones tengan aproximadamente 20 líneas, es más, las funciones deberían tener 3 o 4 líneas y contar una historia que sea sencilla de entender, esto implica que los bloques if, else, while

y similares deberían tener 1 línea que fuera la invocación a otra función. Esto conlleva a que el nivel de sangrado de las funciones sea de 1 o de 2, con lo que resulta extremadamente fácil de leer.

Otra máxima a seguir en las funciones, es que solo deben hacer una cosa, existe una frase que dice: Las funciones solo deben hacer una cosa, deben hacerlo bien y deben ser lo único que hagan. Lo complicado es saber que cosa debe hacer, una forma sencilla de saber que una función hace una sola cosa es que esa función solo realize cosas situadas un nivel por debajo del nombre de la función, si la función se llama *RenderizarPaginaYTestearPagina*, la función solo debería renderizar la página y llamar a

otra función que se encargue de testear la página, otra forma es la de que una función que solo hace una cosa no se puede dividir en secciones.

El objetivo es que el código se pueda leer de arriba hacia abajo, es lo que se denomina la

regla descendente. Que en cada función que vamos bajando se baje en 1 el nivel de abstracción, así solo quedaría un solo nivel de abstracción en cada función, es algo complicado de entender, pero es la clave para escribir pequeñas funciones.

Las sentencias switch, generalmente son grandes y hacen varias cosas, incumpliendo el

principio de responsabilidad única y el principio de abierto/cerrado en el que caso de que haya que añadir un caso más por cada tipo nuevo de empleado que añadamos por ejemplo.

La solución a esto es utilizar el patrón abstract factory y ocultar la función switch. ¿Y qué pasa con los argumentos de las funciones? El número de argumentos ideal para

una función es cero, luego uno y después dos. Siempre que sea posible hay que evitar la presencia de tres argumentos. La diferencia entre un método con argumentos y otro que no, es que cada vez que veamos dicho método tendremos que interpretar el argumento que se encuentra en otro nivel de abstracción, con 1 es sencillo pero con 2 y con 3 la cosa se vuelve compleja. Existen los denominados argumentos de indicador, que es pasar un valor booleano para que la función haga una u otra cosa en función de este valor, es totalmente desaconsejable, es preferible dividir esa función en 2.

Existe otro concepto llamado objeto de argumento, y es cuando pasamos dos o más argumentos en un solo argumento mediante un objeto, supongamos que tenemos un

CAPÍTULO 4 – COMENTARIOS

Un comentario bueno puede resultar muy útil, un comentario antiguo y que diga mentiras es un error, los comentarios se suelen usar cuando somos incapaces de expresarnos en el código, pero lo ideal es dedicar tiempo a podernos expresar correctamente en el código, porque un comentario antiguo que no haya sido actualizado puede crear confusión, debido a que el código probablemente se haya modificado con el tiempo. Aunque los comentarios sean necesarios en ocasiones, debemos dedicar nuestra energía a minimizarlos.

Una de las razones por las que se escriben comentarios es el código incorrecto, creamos

una función o una clase, vemos que es un poco confuso y lo comentamos, mejor dedique ese tiempo a crear una función o clase con un buen código, con código limpio. En ocasiones basta con crear una función que exprese exactamente lo mismo que el comentario.

Aun así, existen comentarios buenos y necesarios, aunque recuerde que el mejor

comentario es aquel que no hay que escribir.

Existe algo muy habitual como los marcadores de posición cuando escribimos un comentario del tipo // Acciones ///////////////////////////// para definir que a partir de ese punto existirán funciones que serán acciones, esto resulta al final muy molesto, quizás sean útiles pero si se usan de forma esporádica. También vemos comentarios de

cierres de llave, al final de un if o un while para indicar su cierre, esto podría tener sentido en estructuras muy grandes, pero quizás debería refactorizar su código y reducir su tamaño, como apunte personal hoy en dia los IDE modernos nos permiten ver el principio y el final de un bloque de manera visual.

No es recomendable incluir demasiada información en los comentarios, evite los comentarios de 10 o 20 líneas, o más. En definitiva, utilice los

comentarios de forma correcta y coherente.

CAPÍTULO 5 – FORMATO

El formato es algo muy importante a tener en cuenta, puede que piense que como desarrollador lo más importante es que el código funcione, pero más aún es que se lea bien, puesto que la funcionalidad puede cambiar en la siguiente versión pero la buena o mala legibilidad del código seguirá ahí.

Vamos a poner un ejemplo con un artículo de periódico, que tiene un titular que nos da una idea general y a medida que nos adentramos va añadiendo detalles, una clase es igual, el título nos debe dar una idea de si estamos en el fichero/módulo correcto, según avanzamos darnos algunos detalles más y al final de la clase ver todos los detalles, si el artículo de periódico fuera gigante y desordenado seguramente no lo leeremos, igual pasa con las clases de código.

Es importante añadir una línea en blanco entre diferentes métodos o acciones dentro de

una función, mejorará la legibilidad del código, puesto que a simple vista se pueden ver las diferentes partes de la clase o de la función.

En el tema de las funciones, si una función invoca a otra deben estar verticalmente próximas, y si es posible, la que invoca por encima de la invocada, al igual con las funciones similares conceptualmente hablando, es decir, que hagan cosas parecidas, deben estar próximas verticalmente en el código, de esta forma todo se podría leer como si de un artículo de periódico se tratara.

Veamos qué pasa ahora con el formato horizontal. ¿Cuánto ancho debe tener una línea de código?, observando varios proyectos, vemos que la media se sitúa en 40, y antiguamente existía una regla de 80, hoy en día los monitores son más anchos y con mejor resolución por lo que 100 o 120 podría estar bien, pero no más, como apunte personal, hoy en día muchos IDE te señalan cuál debería ser el ancho adecuado, podríamos ceñirnos a esa norma.

Es muy útil además rodear con espacios los operadores de asignación o las operaciones

aritméticas para una mejor legibilidad como: `int a = 2;` en vez de `int a=2;`

Es importante también respetar los niveles de indentación del código, muchas veces nos

podemos ver tentados a replegarlo en una sola línea cuando un constructor tiene una sola línea por ejemplo. Esto reduce la legibilidad, es mejor respetar el sangrado.

CAPÍTULO 6 - OBJETOS Y ESTRUCTURA DE DATOS

Si hacemos las variables privadas, ¿por qué creamos métodos de recuperación y seteo de variables como si fueran públicas? ¿Cómo solucionamos este problema? Mediante interfaces que oculten la implementación del objeto, pero esto no es todo, también debemos tener cuidado a la hora de nombrar las operaciones en las interfaces, no queremos que se sepa exactamente el tipo de dato, sino que su nombre nos diga lo que es pero de forma abstracta, por ejemplo, en vez de poner *getGallonsOfGasoline,* podemos poner *getFuelPercentageRemaining* , de esta forma no sabemos que el tipo de dato va a ser galones de gasolina, sino que es simplemente combustible en general.

Veamos ahora la diferencia entre objetos y estructuras de datos, los primeros ocultan sus datos con abstracciones y tienen operaciones con dichos datos, mientras que los segundos, muestran sus datos y carecen de funciones. A continuación en el libro se muestran una serie de ejemplos en código para entender el concepto.

Para explorar y aprender API de terceros, existen los que se conocen como pruebas de

aprendizaje, en estas invocamos a la API como lo haríamos en nuestra aplicación mediante experimentos controlados para comprobar si la entendemos. Es decir creamos con test una batería de pruebas con nuestros ejemplos, a parte de ayudarnos a entender la librería de terceros, si esta evoluciona podremos ver si sigue funcionando de la manera en la que la estamos utilizando, así que a parte de gratis, es rentable.

Como conclusión, el código en los límites requiere una separación evidente y pruebas que definan expectativas, debemos evitar que el código conozca los detalles de terceros.

CAPÍTULO 9 - PRUEBAS DE UNIDAD

Nuestra profesión ha evolucionado mucho en los últimos años en los que se ha empezado a hablar del desarrollo guiado por pruebas, antiguamente las pruebas se veían como un código desechable, hace tiempo hice un programa temporizador, e hice un test manual para ver si funcionaba y una vez comprobado deseche ese código, hoy en día hubiera probado

Las tres leyes del desarrollo guiado por pruebas son las siguientes:

1. No debe crear código de producción hasta que haya creado una prueba para ello.
2. No debe crear más de una prueba que falle, el no compilar se considera un fallo.
3. No debe crear más código de producción del necesario para superar dicha prueba.

Esto nos permitirá tener cientos de pruebas tras varios días y semanas de desarrollo, aunque el tamaño de estas pruebas también puede suponer un problema.

Cada prueba se divide en tres partes, generar, operar y comprobar. La primera crea los

datos de prueba, la segunda opera en dichos datos y la tercera comprueba que la operación devuelva los resultados esperados.

Hay que tener en cuenta también que, aunque existan cosas que jamás usaríamos en un entorno de producción, si las podríamos usar en un entorno de pruebas, cada entorno tiene sus propias necesidades.

Existe una escuela de pensamiento que dice que todas las funciones de prueba sólo deben tener una sentencia de afirmación, es decir solo un *assert* , esto es útil, porque las pruebas llegan a una misma conclusión que se entiende de forma rápida y sencilla. Aunque por mi parte, no rechazo incluir más de una afirmación si es necesario, lo mejor que podemos decir es que el número de afirmaciones de una prueba debe ser mínimo. Lo que sí es importante respetar es que cada prueba solo compruebe un solo concepto.

Además de lo mencionado, las pruebas limpias siguen otras cinco reglas, cuyas iniciales

forman las siglas FIRST en inglés:

* Rapidez: Las pruebas deben ser rápidas y ejecutarse de forma rápida. Si son lentas terminará por no ejecutarlas.
* Independencia: Las pruebas no deben depender entre ellas, una prueba no debe establecer condiciones para la siguiente.
* Repetición: Las pruebas deben poder repetirse en cualquier entorno.
* Validación automática: Las pruebas deben tener un resultado booleano: o aciertan o fallan.
* Puntualidad: Las pruebas deben crearse en el momento preciso: antes del código de producción.

Esto es solo la superficie sobre este tema tan extenso, pero son los conceptos básicos

sobre pruebas de código.

CAPÍTULO 10 – CLASES

Hasta el momento nos hemos fijado en muchas cosas, pero no tendremos código limpio

hasta que nos fijemos en los niveles superiores, las clases.

La convención de Java nos indica que una clase debe comenzar con el listado de variables, y luego las funciones públicas y si llamamos a otra función, esa función debe ir debajo de la que llama, de esta manera cumplimos también la regla descendente.

Es importante que nuestras variables y funciones sean privadas, pero no imprescindible, podemos hacerlas *protected* para que sean accesibles desde una prueba, por ejemplo.

La primera regla de las clases es que deben ser de tamaño reducido, en las funciones nos fijamos en el número de líneas, en las clases la medida serán las responsabilidades.

En el libro vemos una imagen con 70 métodos, aunque la redujeramos a 5 métodos seguirán siendo demasiados, esto es porque la clase tiene demasiadas responsabilidades.

Las clases deben tener un número reducido de variables de instancia, una clase en la que cada método utiliza cada variable tiene cohesión máxima. Esto nos permitiría conseguir que la clase actúe como un todo lógico. Cuando dividimos funciones en una clase y vemos que necesitamos extraer variables para poder usarlas y así sucesivamente, quizás estamos ante un caso en que haya que extraer eso en una clase más pequeña, cuando nuestras clases pierden cohesión cree clases más pequeñas.

RESUMEN A UTILIZAR EN LA CURSADA

| Concepto | Comentario/Referencias |
| --- | --- |
| Código en inglés | Todo lo que concierne al proyecto, está escrito en inglés.   * Directorios archivos, clases, variables   Se valora: comentarios, commits, etc. |
| Commit | * Referencia a tarea en Jira (de existir tablero) * Comentario claro y contextualizado.   por ejemplo:  *"AAW-XXX added tests about installments"* |

Reglamentación de formato y usabilidad de código

| Concepto | Descripción |
| --- | --- |
| Generales | * Camel Case * Las variables empiezan con minúsculas * Las instancias empiezan con minúscula * Los archivos empiezan con minúscula * Se utilizan comillas simples para todos los string literales y comillas dobles internas * Punto y coma en cada final de línea. * Los String se concatenan con string template o la función concat, prohibido usar el operador “+” para ello. * Es mandatario que no haya espacios entre bloques de código, solo se permite separar entre funciones o métodos * Modelos con mayúscula y singular, tablas de Data Base en minúscula y singular |
| Funciones | * Cada parámetro después de la coma se separa por un espacio * Empiezan con minúsculas |
| Variables y constantes | * Empiezan con minúsculas |
| Clases | * Se escriben con mayúsculas * Tienen métodos get para obtener propiedades de las cuales en su nombre no deben contener acciones, la acción es GET * Tienen métodos set para setear variables internas a cada clase * Las variables internas se llamarán así y empiezan con un guión bajo como regla general de encapsulamiento. * Tienen métodos que sirven para procesar algo, obtener algo, u otra operación. * Constructores para inicializar alguna variable requerida. * Los métodos get y set deben ser escritos en una línea si es posible * La organización de las clases: Constructor → Gets → Sets → Métodos * Hay un espacio de división entre cada método |
| Exports | * Todos los exports que contengan un grupo de elementos deben estar ordenados alfabéticamente. * Mandatoriamente los exports se escriben al final del archivo * Los exports pueden ser literales o exportar algo * Existe la posibilidad de hacer exports particulares para lo que se requiera aislado |

Para QA:

| Config | * Por ningún motivo se puede importar la configuración dentro del proyecto. * Custom command sirve para agregar comandos directos para procesar los elementos (conocer su estado o esperar por ellos o alguna propiedad) o extraer información de ellos, propiedades o css. * Custom command también se pueden agregar comandos de browser directamente para procesar operaciones directamente con el navegador, crear acciones o hacer alguna operación directa con un conjunto de elementos. |
| --- | --- |
| Checkers | * Los checker son funciones aisladas al cual recibe un elemento o un conjunto de ellos en forma de componente * Solo contienen expect que prueban un grupo de elementos de la web * No tienen permitido definir elementos, es decir utilizar el operador $ o $$ en los checkers. |
| PageObjects | * Representan una page * Mandatoriamente deben extender de page o algo más particular * No está permitido escribir expects en este lugar |
| Shared/ Components | * Representan un componente dentro de una page * Son clases * Contienen elementos o componentes * Existe la posibilidad de crear un componente general (common) que puede ser compartido y usado en otros componentes * No está permitido escribir expects en este lugar |
| Model | * Son los models de los DTO, solo contienen GET y SET * Se llaman en base a lo que representan * Son simples y no pueden contener métodos de procesamiento de datos |

| Concepto | Descripción |
| --- | --- |
| Data | * Son los generadores de datos, que pueden o no usar models |
| Helpers | * Son un conjunto de herramientas de utilización aislada para solucionar problemas particulares y complejos en donde se requiera * La herramienta date para manejar todo el sistema fecha dentro de los tests, no puede haber otra herramienta que haga esto. * El formateador de fecha sirve para generar texto en base a una fecha y formatear la herramienta date |
| Constantes | * Contiene las constantes de todo el proyecto y están subdivididas * Se agrupan dependiendo de la página en un objeto con nombres descriptivos * Iconos se escriben con nombres con significado unívoco. * Mensajes se crea un archivo en el lugar correspondiente con el nombre de la page, tanto así como el nombre de los elementos representan una constante dentro del archivo en cuestión. |
| Specs | * Ubicación física de todos los test del framework * los nombres de los archivos tienen el formato de {nombre}.{case}.spec.js |

Gabriel Muñoz

Estas buenas prácticas fueron creadas en base a experiencias en grupos de desarrollo, principalmente basado en el libro Clean Code.

Historial de cambios

| Versión | Fecha | Descripción | Modificado por: |
| --- | --- | --- | --- |
| 1 | agosto/2022 | Creación del documento | Gabriel Muñoz |
|  |  |  |  |