1ª edición

Clean Code.

Las mejores prácticas para un código limpio.

eBooks
Paradigma

Autor: Antonio José García.
Ilustraciones: Marta Ruiz.
Maquetación: David Mota.

Paradigma

Contenido:

Introducción:

Como desarrolladores/as, nos hemos encontrado en muchas ocasiones frente a un código que nos cuesta entender, mantener, que genera errores, fricciones en los equipos... ¿Cómo podemos asegurarnos de no llegar a este punto? Existe una serie de recomendaciones y técnicas que permiten tener un código limpio, pero llegar a esta maestría exige dos cosas: conocimiento y trabajo.

Tenemos que aplicar una buena teoría a nuestro trabajo diario para tener un código limpio. Durante este proceso cometeremos errores, aprenderemos de ellos e iremos siendo mejores artesanos de código. A lo largo de este ebook vamos a ver diversas definiciones y técnicas de clean code que están muy relacionadas entre sí. Pero primero empecemos por definir qué es código limpio.

—01

¿Qué es código limpio?

No existe una definición exacta de código limpio, pero vamos a ver una serie de definiciones que nos van a ayudar a entender el concepto:

> Cualquier idiota puede hacer código que compila, pero solo un buen programador puede hacer código que otros entiendan.
>
> 
>
> **Martin Fowler.**

El código es para seres humanos, no para máquinas, por lo que debe ser entendible. Es tan importante que sea legible como ejecutable.

> Me gusta que mi código sea elegante y eficiente. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser mínimas para facilitar el mantenimiento, el tratamiento de errores completo y sujeto a una estrategia articulada y el rendimiento debe ser óptimo para que no se tienda a estropear el código con optimizaciones sin sentido.
>
> El código limpio hace bien una cosa.
>
> 
>
> **Bjarne Stroustup, inventor de C++.**

Aquí se abordan varios temas: evitar acoplamientos y responsabilidad única (patrón SOLID). Nuestro código debe hacer una cosa y hacerla bien.

> El código limpio es simple y directo. Se lee como un texto bien escrito. No oculta la intención del diseñador sino que muestra nítidas abstracciones y líneas directas de control.

Grady Booch, autor de Object Oriented Analysis and Design with Applications.

Volvemos a reincidir en hacer solo una cosa y que no haya comportamientos ocultos. El código no es una película de suspense, debe ser claro y directo.

> El código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que hacer para mejorarlo. El autor pensó en todos los aspectos posibles y, si intentamos imaginar alguna mejora, volvemos al punto de partida y solo nos queda disfrutar del código que alguien a quien le importa realmente nos ha proporcionado.

Michael Feathers, autor de Working Effectively with Legacy Code.

Como buenos artesanos de código, debemos comprometernos con nuestro desarrollo y hacerlo lo mejor que podamos. Es muy importante la ACTITUD, y debemos mejorar la forma en la que nos desarrollamos e ir mejorando poco a poco.

> El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas unitarias y de aceptación, nombres con sentido. Ofrece una y no varias formas de hacer algo. Sus dependencias son mínimas y ofrece un API claro. El código debe ser culto en función del lenguaje, ya que no toda la información necesaria se puede expresar de forma clara en el código.
>
> 
>
> **Dave Thomas, fundador de Eclipse.**

Aquí se incide sobre el buen mantenimiento del código, el nombrado correcto (que veremos más adelante), bajo nivel de acoplamiento (al tener pocas dependencias) y una legibilidad clara del código.

Trabajando es la forma en que obtendremos la madurez en el desarrollo. Como buenos artesanos de software, debemos preocuparnos de hacer código de la mejor manera que seamos capaces: de forma elegante, sencilla, óptima y bonita, teniendo siempre en cuenta el sentido común.

Entonces... ¿Qué es código limpio?

Como ya hemos comentado, debemos ser artesanos de software: involucrarnos en nuestro código y aplicar el sentido común. Tiene que ser sencillo, fácil de entender, de mantener, hacer una única cosa y óptimo.

—02

Principios del clean code.

Ley del Boy Scout.

Se resume en dejar el campamento más limpio que cómo lo hemos encontrado. De esta forma, cada vez que tengamos que hacer un nuevo desarrollo, intentaremos mejorar el código existente. Así, en cada iteración, el código va quedando mejor.

Teoría de las ventanas rotas.

Si en un edificio existe una ventana rota, lo más probable es que los vándalos vayan rompiendo más y más. Ocurre lo mismo con la basura en la calle: si no se va recogiendo, lo más seguro es que vaya apareciendo más. Si lo llevamos a nuestro mundo, nos encontramos clases, módulos mal diseñados y difíciles de mantener que sabemos que existen y debemos eliminarlos lo antes posible. Quizás en el momento actual no podamos acometer la tarea, pero añadamos la tarea al backlog y acometámosla lo antes posible.

—03

Code style.

El 80% de las modificaciones se realizan durante el mantenimiento y por personas que no desarrollaron esa pieza, por lo que cobra gran importancia la mantenibilidad. Un lenguaje bien estructurado, indentado y con coherencia nos ayuda a entenderlo más rápido, ser más ágiles y eficientes en el desarrollo.

Es cierto que hay lenguajes como Python que precisan que la indentación sea correcta, pero para el resto de lenguajes existen formateadores y plantillas que nos ayudan con esta tarea, independientemente del IDE de desarrollo que estemos usando. Además, si como equipo utilizamos la misma plantilla, los cambios al hacer un pull request son menores.

—04

Nomenclatura.

Al final, nosotros somos humanos y el código binario lo entienden las máquinas. Merece la pena dedicar tiempo a pensar el orden de nuestro código, y no empezar a desarrollar pensando solo en que funcione. Si llevamos a cabo esta tarea mejoraremos en claridad, generalidad, mantenibilidad y en facilitar su comprensión.

El idioma en el que desarrollamos el código y los comentarios que le añadimos debe ser un acuerdo del equipo. Es cierto que si desarrollamos en inglés, las palabras son más cortas y encajan mejor con los frameworks que utilizamos. Lo mismo nos ocurre con los comentarios. Pero apliquemos el sentido común y, si hay algo que es difícil de expresar, podemos usar el idioma en el que quede mejor reflejado. Dicho esto, intentemos evitar desarrollos con mezcla de idiomas en el código, ya que conseguiremos que el conjunto sea más complejo de entender.

Nombrado con coherencia.

El nombrado es de vital importancia y debe ser representativo. Da igual que sea una clase, un método, un argumento... Tenemos que ponerle un nombre coherente que evite las pistas falsas sobre qué es, para qué sirve o cómo se usa. Debe ser un nombrado representativo y distintivo.

✗ `int d;`

¿Para qué se usa? ¿Qué hace? No nos aporta ninguna pista.

✓ `int numberOfDaysElapsed`

Sé su propósito y el valor que va a guardar.

No usar nombres con variaciones mínimas.

❌
```
public List<Account> getPublicAccount
```

❌
```
public List<Account> getPublicAccounts
```

Aquí realmente estoy forzado a ver la implementación para entender qué está ocurriendo. Estos dos nombres me generan desinformación: ¿Qué diferencia hay entre las dos funciones?

Por el contrario si tenemos:

✅
```
public List<Account>
getPublicAccountsForActiveUsersInThisBank
```

✅
```
public List<Account>
getPublicAccountsForInactiveUsersInThisBank
```

Lo anterior también aplica a los argumentos en las funciones.

❌
```
public void copyChars(Char a, Char a1);
```
¿Cuál es el origen y cuál el destino? En este caso de nomenclatura, estoy obligado a ver la implementación.

✅
```
public void copyChars(Char source, Char destination);
```
Ya no es necesario que vaya a ver el código de la función, con esta nomenclatura queda claro.

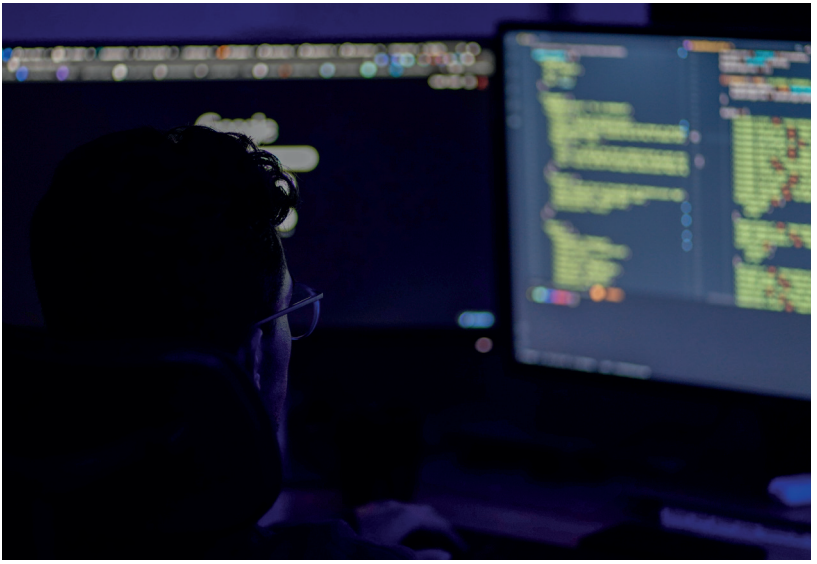
Merece la pena dedicar tiempo a pensar el orden de nuestro código.

Usemos nombres que podamos pronunciar y no caigamos en nombres divertidos o frikis.

Si no es posible pronunciarlo, va a ser más difícil poder explicar o inferir su propósito. Lo mismo ocurre si usamos algún nombre divertido que en ese momento parecía adecuado. La persona que venga detrás seguramente no piense como la persona que lo creó, y no le estamos haciendo ningún favor a la hora de entender el código.

✗ `private String cMarcaAc, dDescMod, desEmpre;`
Tenemos que investigar cada una, ya que con esta nomenclatura no entendemos qué es o a qué se refiere.

✓ `private String activeBrandName, moduleDescription, companyName`
Nombres distintivos.



Tipo y nombres coherentes.

La idea es no generar pistas falsas.

✗ `private String accountList;`
¿Plural y es sólo un string?¿Almacena una o varias account concatenadas? Tengo que investigar cómo se asigna el valor y su uso.

✓ `private String account;`
Singular, almacena una sola cuenta.

✓ `private List<String> accountList;`
Es una lista de cuentas y es coherente con el nombre.

Prefijos y sufijos que no aportan nada.

✗ `Product = ProductInfo = ProductData` (es lo mismo)
En este ejemplo nos estamos refiriendo a "Product", y los sufijos no aportan nada salvo desorientación.

Otro ejemplo de prefijos que no aportan nada es añadir el tipo a la variable (notación húngara) a su nombre. Esto nos obligaría, además, a cambiar la variable si cambiamos de tipo. Existen lenguajes fuertemente tipados donde esta técnica tiene todavía menos sentido.

✓ `private int Icount; boolean bIsActive;`

Es cierto que existen casos donde un prefijo nos aporta el contexto de la variable con la que estamos trabajando. Si estamos trabajando con el atributo status de la clase Product en otra parte, productStatus nos ayuda a identificar claramente a qué estamos haciendo referencia.

Si todas nuestras clases tienen el mismo prefijo, realmente ese prefijo no aporta nada y nos dificulta la búsqueda de clases. Un ejemplo claro es cuando prefijamos todas las clases con el nombre de una empresa; quizás esas clases puedan ser extraídas a una librería.

Ocurre lo mismo si prefijamos las interfaces con una I. No está aportando nada y estamos perdiendo abstracción. Pero por el contrario, si a una implementación le añadimos Impl al final de la clase (CacheServiceImpl, CacheServiceRedisImpl), indicamos claramente su propósito.

De la misma forma, añadir el sufijo cuando trabajamos con patrones nos aporta semántica.

✓ `ProductFactory, AccountFacade, CustomerPublisher`

Como regla general, sería recomendable no usar en clases:

- Solo un verbo (Manager, Procesor, Get...) que no nos identifica claramente su propósito.
- Ni palabras generalistas como Data, Info, General...

Pero sí nombres o frases como ProductorParser, CustomerValidator...

Los métodos hacen acciones, por lo que deben llevar un verbo como parte de su nombre y también indicar sobre qué actúa.

✓ `getAccount(), generateProduct(…), setName(…)`

—05

Comentarios.

Los comentarios deben aportar el conocimiento necesario donde no hemos podido expresarnos con el código.

A su vez, un comentario implica una responsabilidad de mantenimiento según se vaya actualizando esa pieza de código y nunca, nunca, nunca, debe servir para comentar código. El código comentado nos genera dudas y, para esa función, tenemos los repositorios de código.

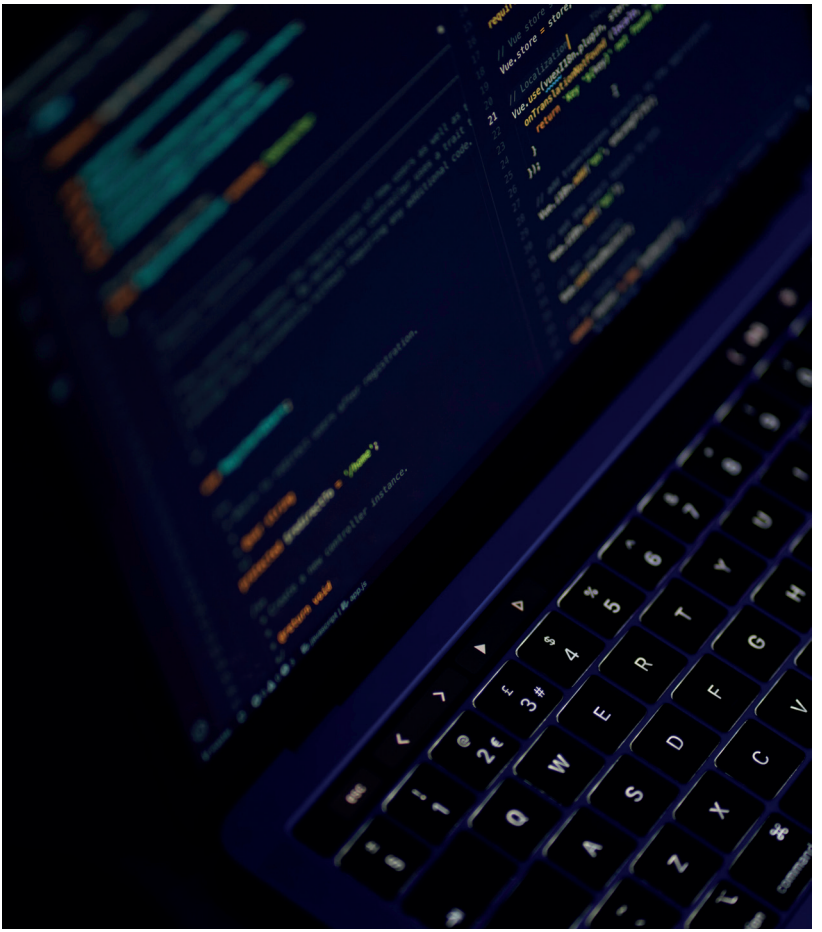
En qué casos nuestros comentarios no tienen sentido:

- Comentarios con cambios por fecha. Para eso tenemos los repositorios de código, changelog, Jira...
- Metadatos: autores, fecha...
- Justificaciones de decisiones.
- Explicación demasiado profunda.
- Comentarios obsoletos. O lo actualizamos o tenemos que eliminarlo.
- Código comentado. Tenemos que eliminarlo porque nos genera más desinformación.
- Código redundante, como podría ser el getter o el setter de un campo.
- Menciones a otros usuarios que quizás no formen parte ya del equipo de desarrollo.
- Usar comentarios para separar bloques o señalar llaves de apertura o cierre.

Los comentarios deben aportar el conocimiento necesario donde no hemos podido expresarnos con el código.

Los comentarios son notas técnicas y de diseño que merece la pena leer, y se deben mantener en el futuro.

Deben tener un lenguaje claro y una ortografía correcta. El desarrollo es una actividad social y todo lo que podamos ayudar y guiar será de gran utilidad. Debemos tener cuidado con pensar que nuestro código es autocomentado y expresar con palabras donde nuestro código no puede llegar.



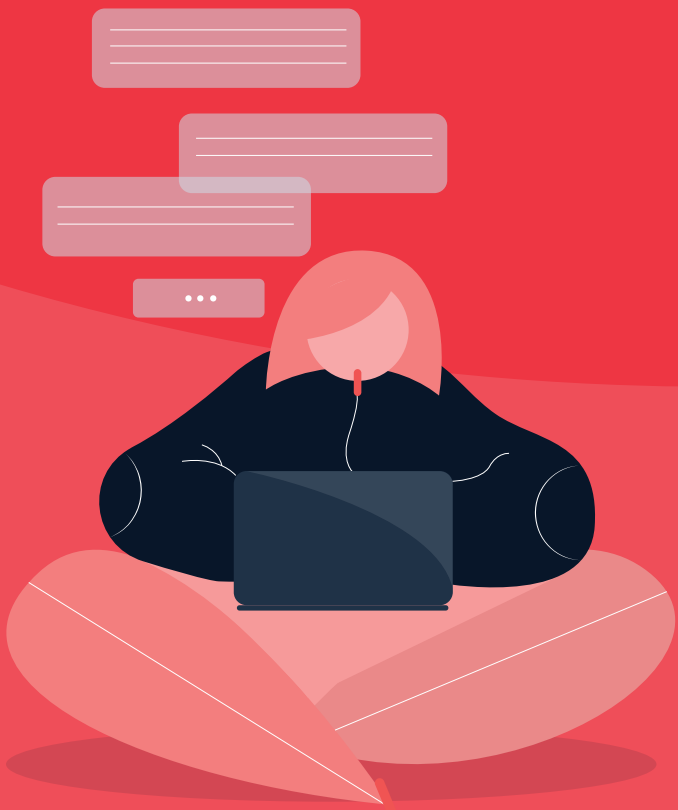
Un ejemplo de un buen comentario.

```
/**
* Servicio principal desde el cual se orquestn tres llamadas a
* los subsistemas de obtención de información extendida de cada
* uno de los productos resultantes de la consulta inicial a la
* base de datos
*
* <pre>
*  Las tres llamadas son:
*   - Obtencion de precios
*   - Obtención de la taxonomía del producto
*   - Obtencion de manuales de usuario
*
* </pre>
*
* @param ProductSearchParameterIDTO
* @return ProductFlowODTO
*/

@Override
public ProductDocumentRSRDTO
searchProduct(ProductSearchParameterIDTO
productSearchParameterIDTO) {
```

—06

Funciones.



Hace muchos años, las funciones eran subrutinas compuestas por miles de líneas. Después, vinieron los lenguajes que limitaban el tamaño a unas cuantas líneas. Pero, ¿cómo deberían ser nuestras funciones?

Deberían ser concretas, claras y con una única responsabilidad (esto conectaría con los principios SOLID). Además, al ser concreta, su tamaño es pequeño y esto nos favorece a la hora de poder testear nuestro código. Cuanto más pequeña es la función, normalmente es más fácil de entender y no oculta ningún efecto secundario.

El lenguaje y cómo están escritas cobra mucha importancia. Si el nombre que elegimos describe su propósito, no será necesario entrar en un principio a su implementación. Además, estamos acostumbrados a leer de izquierda a derecha y en párrafos. Una buena indentación y separar cambios u operaciones en párrafos es de gran ayuda. Las llaves, estructuras de código, agrupar algunas operaciones... nos pueden ayudar a ello.

Una función debe ser concreta, clara y con una única responsabilidad.

A continuación podemos ver una función bien escrita que se puede entender de un simple vistazo:

```
public ProductDocumentRSRDTO
searchProduct(ProductSearchParameterIDTO
productSearchParameterIDTO) {

  /*Transformamos a UniqueSearchParameterIDTO*/
  UniqueSearchParameterIDTO uniqueSearchParameterIDTO = this.
  productServiceTransformer
    .toUniqueSearchParameterIDTO(ProductSearchParameterIDTO);

  /*Buscamos las referencias por cada producto*/
  ProductFlowODTO productFlowODTO =
  this.productRestIntegrationService
    .findReferencesInUniqueSearch(uniqueSearchParameterIDTO);

  /*Si tenemos referencias vamos a completar la documentación*/
  if (productFlowODTO.getReferencias() != null) {
    productFlowODTO = completeDocumentation(productFlowODTO);
  }

  ProductDocumentRSRDTO productDocumentRSRDTO = this.
  productServiceTransformer
    .toProductDocumentRSRDTO(productFlowODTO.getDocuments());

  return productDocumentRSRDTO;

}
```

Cuando estemos codificando y tengamos un switch (de hecho Python no tiene), se nos debería encender una alarma y revisar qué estamos haciendo, porque en muchos casos esa función hace más de una cosa y estamos violando el principio de responsabilidad única.

Un mal ejemplo (se están haciendo muchas cosas en la misma función):

```
switch (value) {
  case 1: return sendEmail();
  case 2: return calculatePercentage();
  case 3: return callToCics();
```

Normalmente, la solución es hacer una factoría abstracta que haga visible el comportamiento, pero no hay una regla escrita, depende del caso. La solución es favorecida mediante el uso del polimorfismo.

Sobre el número de argumentos, la regla sería que cuantos menos, mejor (que nos favorece a testearlas con mayor facilidad).

- Lo ideal sería tener funciones monádicas (consultas y transformaciones sobre un objeto).
- Diádicas, no son el mal pero a veces son necesarias (como en las que tenemos un expected, actual). Aunque si podemos transformarlas en monádicas, mucho mejor.
- Evitemos en lo posible funciones triádicas.
- A partir de aquí (poliádicas) la recomendación sería encapsularlas en un objeto. Aquí ocurren varios factores: no nos acordamos de los argumentos y su orden, si fuera necesario otro argumento impactaría más en el código y es más fácil de utilizar y probar.

Los nombres de los argumentos deben ser identificativos.

Una función bien escrita se puede entender de un simple vistazo.

Si tenemos un argumento que es un booleano, puede ser indicativo que nuestra función hace más de una cosa. Porque dependiendo del valor del booleano hace una u otra cosa.

```
public committeeODTO saveCommittee(CommitteeIDTO committeeIDTO,
Boolean isUnique);
```

Las funciones que no se utilizan (muertas) deben ser eliminadas, al igual que todo el código que se encuentre comentado, porque para eso tenemos los repositorios de código.

Dentro del código hay que minimizar el acoplamiento (lo que sabe una clase de otra). Para ello existe la ley de Demeter, que básicamente la podríamos resumir en que una clase habla con sus amigos y no con desconocidos:

- Accede a tus atributos (simples o compuestos).
- Accede a tus propiedades.
- Accede a los atributos de las funciones.

De esta forma minimizamos los cambios en caso de un cambio de una estructura. Pensemos qué ocurre si cambiamos los atributos de la función doSomething en el código de la izquierda, fuertemente acoplado respecto con el de la derecha. Además es probable que si no somos conscientes, el código de la izquierda estará repartido por muchas clases en vez de manejar la clase Z.

```
getX().getY().getZ().doSomething();
```

```
X xVariable = getX();
Y yVariable = xVariable getY();
Z zVariable = yVariable getZ();
zVariable.doSomething();
this.url = url;
```

Si poseemos un objeto que contiene otros objetos, no deberíamos expandir el acceso a ellos en profundidad desde la raíz sino desde el propio atributo (acceso a los inmediatos).

Otra recomendación es minimizar los condicionales. Si tenemos más de dos condicionales, deberíamos extraer en un método y refactorizar. El código de debajo es más entendible.

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
} else {
  charge = quantity * summerRate;
}
```

```
if (notSummer(date)) {
  charge = winterCharge(quantity);
} else {
  charge = summerCharge(quantity);
}
```

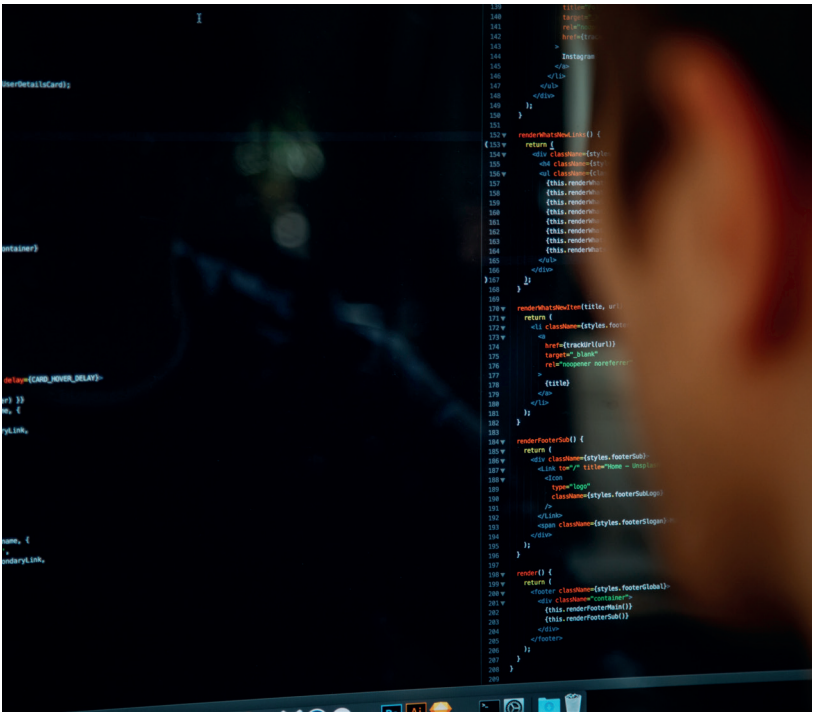
Debemos evitar los efectos secundarios que se producen:

- Cuando una función altera variables locales o de clase.
- Cuando se crean dependencias con respecto al orden de ejecución de los métodos.

Además, cuando esto ocurre es porque la función hace más de una cosa. Es importante el orden en el que se ejecuta y tener errores donde no corresponde.

¿Por qué se inicializa la base de datos en una función que hace login?

```
Boolean login(String user, String password) {
  if (validLogin(user, password)){
    initDatabase();
    return true;
  }
  return false;
}
```



¿Cómo debería ser el proceso de creación de una función? Aquí dejamos unas recomendaciones.

- Estructuremos bien nuestras ideas y lo que queremos hacer.
- Generemos un borrador de cómo lo vamos hacer.
- Iteremos nuestro borrador hasta que estemos seguros que es la mejor solución.

Dicho esto, a la hora de codificar intentemos:

- Hacer funciones claras y pequeñas.
- Pensar una buena nomenclatura y buscar una buena organización.
- Eliminar duplicidad, abstraer, generalizar y usar todas las buenas prácticas que conozcamos.
- Repetir todo lo anterior e intentar mejorarnos como desarrolladores/as.

—07

DRY (Don’t repeat yourself).

Simplemente se trataría de no repetir código. Cuando tengamos la sensación de estar haciendo lo mismo, merece la pena pararse y mirar si podemos hacer una librería, una funcionalidad genérica o cualquier técnica con la podamos reutilizar el código.

Debemos estar comprometidos con el desarrollo y refactorizar cuando sea necesario, buscando una mayor mantenibilidad y claridad. Si no, estamos generando una ventana rota y habrá que hacer un doble esfuerzo en el futuro para mantener ese código.

—08

Tratamientos de errores.

En primer lugar, es mejor devolver excepciones que códigos que luego sean capturados y tratados por un handler. Con excepciones queda más claro y más mantenible, y si trabajamos con aspectos (AOP) la excepción se podría tratar fácilmente.

No devolvamos null y fallemos lo antes posible justo donde ha ocurrido el error. Al final, delegamos ese tratamiento del null en otra parte que no es responsable.

Es importante no ocultar los errores, registrar en los logs y fallar lo antes posible para poder ver qué está ocurriendo y dónde con mayor exactitud.

No devuelvas null y falla lo antes posible, no ocultes los errores, busca por qué ha fallado y añade pistas que faciliten el trabajo.

Contextualicemos los errores y añadamos pistas que nos faciliten nuestro trabajo, ver por qué ha fallado, incluso datos para poder reproducirlo.

Por último y para evitar acoplamientos, cuando consumimos un API o librería de terceros no deberíamos propagar sus excepciones y encapsularlas en excepciones propias. De esta forma, nuestro código está más preparado para futuros cambios.

—09

Clases y objetos.

Dependiendo del lenguaje, existe un code style con un orden de declaración de los elementos que conforman una clase. Por ejemplo, en Java suele ser: atributos estáticos, métodos estáticos, atributos públicos, atributos privados, constructores, métodos públicos, métodos privados...

Como regla general, debemos pensar que nuestras clases deben tener un tamaño reducido. Así podemos asegurarnos de que tiene una única responsabilidad y no es una clase todopoderosa (God Object Antipattern), que significa que conoce más de lo debe (de su responsabilidad).

Hay dos conceptos claves: el acoplamiento y la cohesión.

Acoplamiento :

Es el grado en el cual una clase sabe acerca de otras clases. Un acoplamiento fuerte significa que las clases están muy relacionadas y que necesitan saber mucho unas de las otras, lo que provoca que los cambios se propaguen por todo el sistema. Posiblemente sea más difícil de mantener y de entender.

Para ello, debemos fomentar el bajo acoplamiento que nos permite: entender una clase sin leer otras, cambiar una clase sin afectar a otras y mejorar la mantenibilidad del código.

Cohesión :

Es el grado de cómo una clase está diseñada para interactuar consigo misma. Es individual por cada clase, y nos ayuda a ver si tiene un propósito bien enfocado.

Debemos tender a minimizar el acoplamiento y aumentar la cohesión.

Normalmente trabajamos con aplicaciones que tienen N capas o módulos independientes. Para conseguir un desacoplamiento alto existen DTO (Data transfer Object), que son clases de intercambio entre capas. Son objetos simples que no tienen lógica.

Existen librerías que nos ayudan a mantener la claridad de nuestros objetos y liberarnos de ese código que forma parte de nuestra clase pero aporta poco (getters, setters, toString, constructores...). El uso de Lombok o el uso de Java Records (JDK >= 14) nos puede ayudar para ese propósito.

Además, para transformaciones de objetos tenemos la librería de Mapstruct, que nos facilita mucho el trabajo (compatible con Lombok).

Nuestras clases deben tener un tamaño reducido. Así podemos asegurarnos de que tienen una única responsabilidad.

—09.01

STUPID Antipattern.

Cuando no cumplimos estas recomendaciones, podemos estar cayendo en el antipatrón STUPID. Es importante reinventarse cada día y mejorar como desarrollador/a para no caer en estas malas prácticas, a veces sin ser conscientes. STUPID son las siglas de:

- **Singleton:** Debemos evitarlos por varias razones: son difíciles de testear, guardan un estado global de la aplicación, son difíciles de mantener, ocultan dependencias y suelen estar muy acoplados.
- **Tight coupling:** Esto ocurre cuando muchas clases conocen mucho sobre otras clases que no deberían conocer. Al final, un cambio en una clase impacta en muchas otras, empeorando la mantenibilidad y disminuyendo la capacidad de reutilizar. Un acoplamiento ligero es necesario, ya que tenemos interfaces de integración entre diferentes módulos.
- **Untestability:** Si nuestro código es difícil de testear seguramente esté fuertemente acoplado, o algo hemos hecho mal.
- **Premature optimization:** Sería refactorizar algo que realmente no es necesario y que nos puede llevar a situaciones más complejas. Por eso, siempre debe priorizarse el sentido común.
- **Indescriptive naming:** El código es para humanos, no usemos abreviaturas ni nombres impronunciables. Busquemos una buena nomenclatura ya sea para clases, funciones, paquetes...
- **Duplication:** Este punto ya lo vimos en el apartado de DRY. Escribamos nuestro código una vez y reutilicémoslo.

—09.02

SOLID pattern.

Para luchar contra STUPID tenemos el patrón SOLID, que es una colección de principios de diseño de buen código recogidos por Rober C. Martin. Si cumplimos el patrón, el código será legible, reutilizable y mantenible. Estos principios son:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsability: Una clase debe tener una única responsabilidad y una razón para cambiar. Si se asume más de una responsabilidad, el código quedará acoplado y será más sensible al cambio. Evitemos las clases todopoderosas (god classes) y dividamos las responsabilidad (divide y vencerás).

Open/Closed: El resumen es que una clase debe estar abierta para extender (heredar), pero cerrada (se puede reutilizar) para modificaciones.
Los componentes de nivel superior (ancestros) están protegidos de cambios de los de nivel inferior (descendientes). La funcionalidad se ordena de manera jerárquica.
Como regla general, todos los atributos de una clase serán privados y tendremos los getters y setters que tengan sentido.

Liskov Substitution: Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

Interface Segregation: Muchas interfaces de cliente específicas son mejores que una interfaz de propósito general. Es decir, cuando desarrollamos y queremos implementar algo específico no debemos estar obligados a implementar más métodos de los que realmente necesitamos. Imaginemos una navaja suiza: es mejor hacer una cosa bien, que tener que hacer muchas si solo quiero cortar.

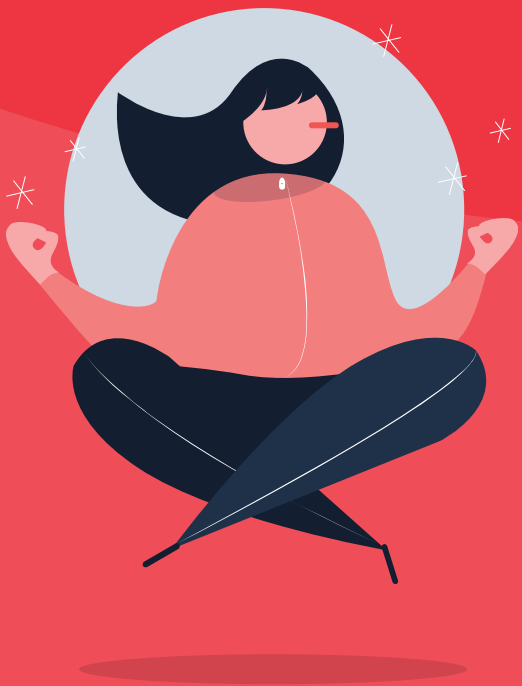
Dependency Inversion: Se basa sobre todo en dos puntos:

- Las abstracciones no deben depender de las implementaciones (deberíamos poder cambiar de implementación y nuestro código seguiría funcionando).
- Las implementaciones dependen de las abstracciones.

No añadamos clases concretas al contrato de la interfaz y usemos interfaces en nuestras clases. Lo que buscamos es reducir el acoplamiento y aumentar la reutilización.

—10

Introducción al testing.



Test de UI:

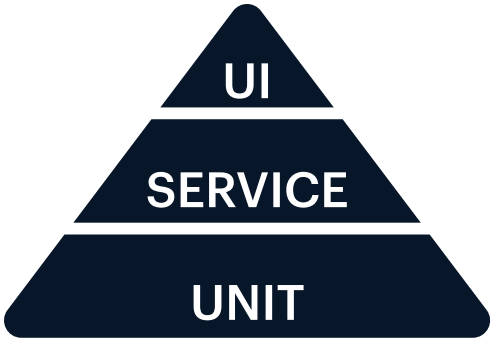
- Se ejecutan sobre la capa de UI.
- Son lentas de ejecutar.
- Son difíciles de ejecutar en un entorno local.
- Son de alto nivel y suelen requerir licencias.

Test de Servicio:

- Se ejecutan sobre la capa de API.
- Son end to end, obviando la capa de UI.
- Son pruebas de alto nivel funcional.
- Si detectamos un error, seguramente nos falten test unitarios.

Test Unitarios:

- Se ejecutan sobre partes concretas del código.
- Son muy rápidas de ejecutar.
- Garantizan que los errores no se van a reproducir.



$$$

Ç

Test Unitarios:

Nuestros test deben cumplir las siguientes características.

- **Fast:** Cuanto más rápidos sean, más veces los ejecutaremos.
- **Isolated:** El orden de ejecución no importa porque son test aislados.
- **Repeteable:** Somos capaces de repetirlos y siempre obtienen el mismo resultado.
- **Self-validating:** Son autovalidados, no requieren de ninguna acción manual para validarlos y el resultado es un booleano.
- **Timely:** El test debe ser diseñado antes del código que queremos probar. Esto está en conexión directa con el desarrollo guiado por los test (TDD), que tendría que ser la forma en la que deberíamos desarrollar por todos los beneficios que nos aporta.

“

Clean code always looks like it was written by someone who cares.

Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship.

Autor:

Antonio José García.

Estrategia.

Soy un apasionado de las últimas tecnologías, siempre con ganas de aprender y mejorar. Me encanta cualquier deporte y soy socio del Real Madrid. También me gusta la cerveza, el cine y los videojuegos.

Think Big.

V.1.0 - Septiembre de 2022

info@paradigmadigital.com

eBooks
Paradigma