Clean Code

Rubén Gómez García

Clean Code

- El código se lee en proporción 10:1 con respeto a lo que se escribe, cada línea de código que se escribe normalmente exige una lectura del contexto para asegurarse de que lo que escribimos es coherente, luego hay que pensar en la codificación como en un libro que otros leerán.
- Cuando se habla de clean code, se habla de
  - Código que es fácilmente interpretable, por lo que tiene que ser claro y legible, es deseable que sea corto para no perder el hilo.
  - Código que debe estar desacoplado de sus dependencias, que los cambios en ellas le afecten lo menos posible.
  - Código que es testable.
  - Código que sigue el principio de responsabilidad única.

Nombres con sentido

- Los nombres han de revelar una intención. No importa si a lo largo del tiempo decidimos cambiar un nombre porque hayamos encontrado otro que sea mas adecuado.
- Los nombres no han de revelar cosas que no son. Hay que tener cuidado con el significado de las siglas o de las palabras en el contexto.
- El compilador no cuenta como lector, sus reglas son demasiado básicas, pongo un carácter mas a un nombre y el ya no protesta.
- Los nombres han de ser pronunciables, nada de acrónimos que sean susceptibles de reinterpretarse al ser pronunciados.
- No emplear prefijos innecesarios, Interfaces sin “I”

Nombres con sentido

- Los nombre de clases deben ser sustantivos.
- Los nombres de métodos deben ser verbos.
- Loas métodos de factoría deben hacer referencia a los parámetros empleados en la construcción (From).
- Hay que mantener los conceptos a lo largo del código, no emplear sinónimos en distintas partes para representar el mismo concepto, o peor usar una misma palabra para representar conceptos distintos.
- Sin juegos de palabras.
- Dar contexto a los conceptos genéricos con prefijos o asociándolos a una entidad superior (Clase, paquete)

Funciones

- Tamaño reducido. Las funciones no deberían ocupar mas de 20 líneas, lo ideal es que ocupen menos de 10.
- Las funciones solo deben hacer una cosa. Principio de la responsabilidad única. Evitar por tanto parámetros boléanos, lleva a pensar que la función hace dos cosas.
- Las funciones principales, deben aparecer antes que las funciones secundarias, leemos de arriba a abajo.
- Evitar funciones con mas de 2 parámetros, dificultan la lectura y las pruebas.
- Añadir al nombre en lo posible el orden de los parámetros para facilitar la lectura.

Funciones

- Evitar los efectos secundarios (side effects), aquellas cosas que se hacen sin avisar en el nombre.
- Evitar parámetros de salida, para eso están los retornos.
- Evitar funciones que hacen algo y además retornan algo.
- Usar excepciones en lugar de códigos de error.
- Extraer el cuerpo de un try-catch a una función para mejor legibilidad.
- Procesar errores es una responsabilidad, luego no se debe hacer nada mas.
- Refactorizar los códigos duplicados.

Comentarios

- No se debería tener que perder el tiempo en mantener los comentarios.
- Los comentarios no compensan el código, si el código necesita ser comentado es que no es bueno.
- Explicar la intención en el código buscando semántica.
- Emplear comentarios TODO, para indicar que falta por hacer algo.
- Evitar
  - javadocs en código no publico.
  - Demasiados comentarios.
  - Código comentado
  - Redundancia de comentarios.
  - Comentarios no explicativos.
  - Comentarios que indican que cambios y cuando se hicieron, para eso esta el SCM.

Formato

- Debe ser claro, coherente y mantenerse en todo el proyecto, por lo que hay que acordarlo con el equipo.
- Las clases no deberían superar el 1% del tamaño total en numero de líneas del proyecto, siendo recomendable 0.5%.
- Líneas en blanco para separar conceptos.
- Juntar líneas que estén asociadas.
- Los conceptos (métodos) relacionados deben mantenerse juntos.
- Las líneas deben ocupar en torno a los 45 caracteres, no mas de 100.
- Separar los operadores y los parámetros con espacios en blanco.
- Aplicar sangrado a los ficheros para enfatizar a que nivel corresponde.

Estructuras de datos

- No añadir get y set básicos por que sí, pensar en como se quiere ofrecer la información al exterior.
- Ley de Demeter, un método solo debe invocar métodos de objetos que se le han pasado por parámetros, que sean atributos de la misma clase, que pertenezcan a la misma clase donde esta definida o un objeto que instancie el propio método.

Excepciones

- No se debe invadir la representación de la lógica con el tratamientos de los errores.
- Mejor excepciones uncached que cached.
- Encapsular excepciones de librerías de terceros para minimizar el acoplamiento con la librería a aquel código que realmente la necesita.
- No retornar null.
- No contemplar pasar null a los métodos, evita tener que comprobar y permite identificar un mal uso de un método.

Limites

- Definir adaptadores en aquellos casos en los que los tipos manejados (APIs de terceros) ofrezcan mas funcionalidades de las deseadas o sean susceptibles de cambiar para minimizar su impacto en el software.
- Realizar **pruebas de aprendizaje** para comprender los APIs de terceros, esto es, realizar ejemplos básicos del uso que se pretende del API para validar que lo entendemos, serán de gran ayuda ante cambios de versión de esos APIs.
- Emplear Interfaces para establecer limites de nuestro código.

Pruebas

- Deben seguir las mismas reglas de estilo que el código, dado que han de ser mantenidas, deben por tanto ser Claras y Simples.
- Aplicar la convención **given-when-then**.
- Se debe tener una afirmación por prueba, en caso de necesitar varias, se crea otra.
- Probar un solo concepto en cada prueba.

Clases

- Encapsulación de campos (privado), salvo que por fuerza mayor, por ejemplo para poder hacer una prueba, tenga que ser accesible.
- Tamaño reducido.
- Principio de Responsabilidad única. El nombre de las clases debe reflejar su única responsabilidad. Evitar el uso de Processor, Manager, Super, …
- Cohesión, los atributos de una clase, han de ser empleados por los métodos de la clase, eso indica que no podrían estar separados, cuando no se cumple habrá que separar en varias clases.
- Organizar los cambios. Si se sigue el principio de responsabilidad única, los cambios de una clase no afectan a otra, luego cuando esto no suceda, quizás es hora de dividir en mas clases, aplicando herencia, …
- Aislar las clases de las implementaciones de sus dependencias con interfaces.

Sistema

- Separar la construcción de los objetos que forman en sistema del uso que se hará de ellos. No se han de instanciar los objetos a emplear en el mismo lugar donde se emplearán, lo objetos se mandarán crear a otros (Factorías) o ya deberán de estar creados (DI).
- El sistema tiene que ser capaz de crecer sin tener que reconstruir todo, se pueden aplicar conceptos de AOP, proxies, …
- Lenguaje de dominio.

Diseño Emergente

- Sistema testable, que de confianza para realizar refactorización.
- No contiene duplicados.
- Expresa la intención del programador.
- Minimiza el numero de clases y métodos.

Concurrencia

- Es difícil.
- Añade carga tanto en el rendimiento como en la codificación.
- Errores difíciles de reproducir.
- Cambios en el diseño.
- El código de concurrencia debe ser independiente del de negocio. Principio de Responsabilidad Única.
- Se han de minimizar los datos compartidos, por ejemplo creando copias.
- Un solo método sincronizado por clase.
- Evitar bloqueos cerrando los flujos (timeouts,…)
- Realizar pruebas que verifiquen la concurrencia.

SOLID

SOLID

- Son las siglas de
  - Single Responsibility
  - Open-closed
  - Liskov Substitution
  - Interface segregation
  - Dependency inversion

SOLID

- Introducido por Robert C. Martin a comienzos del 2000
- Con estos cinco principios, es más probable que un desarrollador cree un sistema mas fácil de mantener y ampliar
- Permiten eliminar código sucio
- Debe ser utilizado junto con TDD

SOLID: Single Responsibility Principle

- Un objeto solo debe tener una única responsabilidad
- Cada módulo de clase debe tener responsabilidad de una sola parte de la funcionalidad proporcionada
- Todos los servicios deben realizarse en base a esa responsabilidad
  - Una clase debe tener una sola razón de cambiar
  - Cada responsabilidad es el eje del cambio
  - Para contener la propagación del cambio, debemos separar responsabilidades
  - Si una clase asume más de una responsabilidad será sensible al cambio
  - Si una clase asume más de una responsabilidad, las responsaibilidades se acoplan

SOLID: Open-closed

- Las entidades de software deben estar abiertas a extensión, pero cerradas a modificación
- Se debe poder extender el comportamiento sin modificar el código fuente
- Originalmente utilizado por Bertrand Meyer
- Se considera un módulo abierto si podemos agregar nuevos campos a la estructura de datos, o nuevas funcionalidades de comportamiento
- Un modulo queda cerrado si es utilizable por otros módulos, luego posee una descripción estable y bien definida

SOLID: Liskov Substitution Principle (LSP)

- Los objetos deben poder sustituirse por instancias de subtipos sin alterar el programa
- Cada clase que hereda de otra puede usarse como su padre sin conocer las diferencias entre ellas

SOLID: Interface Segregation Principle (ISP)

- Muchas interfaces cliente específicas son mejor que una general
- Los clientes de un programa dado solo deben conocer los métodos que usan realmente
- Se aplica a una interfaz amplia y compleja y se separa en muchas pequeñas y específicas para que cada cliente use solo lo que necesite
- A estas interfaces se les llama interfaces de rol
- Se trate de mantener el código desacoplado.

SOLID: Dependency Inversion Principle

- Se debe depender de abstracciones, no de implementaciones
- Dependency Injection es una posible solución
  - Patrón orientado a objetos que suministra objetos a una clase en lugar de construirse en la propia clase
  - Estos objetos cumplen contratos que necesitan nuestras clases para poder funcionar
  - Los objetos son inyectados por una clase distinta a la clase que los necesita.
  - Término usado por primera vez por Martin Fowler