# Entornos de Desarrollo

04 Clean Code: Optimización, Refactorización y Documentación

José Luis González Sánchez

# Contenidos

¿Qué voy a aprender?

# Contenidos

# Clean Code

No pienses solo en ti, si no en los demás

# Clean Code

- Clean Code, o Código Limpio en español, es un término al que ya hacían referencia desarrolladores de la talla de Ward Cunningham o Kent Beck, aunque no se popularizó hasta que Robert C. Martin, también conocido como Uncle Bob, publicó su libro “Clean Code: A Handbook of Agile Software Craftsmanship24” en 2008.
- “Código Limpio es aquel que se ha escrito con la intención de que otra persona (o tú mismo en el futuro) lo entienda.”
- Los desarrolladores solemos escribir código sin la intención explícita de que vaya a ser entendido por otra persona, ya que la mayoría de las veces nos centramos simplemente en implementar una solución que funcione y que resuelva el problema.
- Tratar de entender el código de un tercero, o incluso el que escribimos nosotros mismos hace tan solo unas semanas, se puede volver una tarea realmente difícil. Es por ello que hacer un esfuerzo extra para que nuestra solución sea legible e intuitivaes la base para reducir los costes de mantenimiento del software que producimos.
- A continuación veremos algunas de las secciones del libro de Uncle Bob que más relacionadas están con la legibilidad del código.

# Clean Code: Como nombrar a las variables

- Usa **variables** si su valor va a cambiar a menudo a lo largo del programa. Debes usar para crear su iidentificador el modelo CamelCase, pero siempre con la primera letra en minúscula. Por ejemplo userAge.
- Se usan **constantes** para aquellas varibeles cuyo valor una vez asignado no cambia a lo largo de la ejecución del program,e s decir, siempre sesle mismo, es constante. Usa para crear su identificaror sintaxis kamel case, pero todoe n mayúscula y con guiones bajos. Por ejemplo NUMERO_PI.
- En ambos casos los indetificadores, imprescindiblemente en inglés, deben ser pronunciables. Esto quiere decir que no deben ser abreviaturas
- En los **arrays** son una lista iterable de elementos, generalmente del mismo tipo. Es por ello que pluralizar el nombre de la variable puede ser una buena idea, por ejemplo fruitNames[].
- Los **booleanos** solo pueden tener 2 valores: verdadero o falso. Dado esto, el uso de prefijos como is, has y can ayudará inferir el tipo de variable, mejorando así la legibilidad de nuestro código. Ejemplo isOpen, isSolution
- Para los **números** es interesante escoger palabras que describan números, como min, max o total: totalUsers.

- **CONSEJO:** Piensa que leyendo el identificador debes saber exacatmente qué información almacena y de qué tipo

# Clean Code: Como nombrar a los métodos

- Los nombres de las **funciones o métodos** deben representar acciones, por lo que deben construirse usando el verbo que representa la acción seguido de un sustantivo. Estos deben de ser descriptivos y, a su vez, concisos. Esto quiere decir que el nombre de la función debe expresar lo que hace, pero también debe de abstraerse de la implementación de la función. Ejemplo createUser(), deleteFruitList().
- En el caso de las funciones de acceso, modificación o predicado, el nombre debe ser el prefijo get, set e is, respectivamente. Ejemplo **isValidUser()**

- **CONSEJO:** Piensa que leyendo el nombre del método deberías adivinar el tipo de acción que realiza.

# Clean Code: Como nombrar a las clases

- Las clases y los objetos deben tener nombres formados por un sustantivo o frases de sustantivo como User, UserProfile, Account o AdressParser. Debemos evitar nombres genéricos como Manager, Processor, Data o Info.
- Hay que ser cuidadosos a la hora de escoger estos nombres, ya que son el paso previo a la hora de definir la responsabilidad de la clase. Si escogemos nombres demasiado genéricos tendemos a crear clases con múltiples responsabilidades.
- **CONSEJO:** Piensa que leyendo el nombre de la clase debes saber exacatmente qué información encapsula, y que acciones realiza.

# Clean Code: Como nombrar a los paquetes o ficheros de módulos

- Siempre lo haremos en minúscula
- **CONSEJO:** Piensa que leyendo el nombre de la clase debes saber exacatmente qué información encapsula, y que acciones realiza.

# Clean Code: Ámbito de variables

- El ámbito global debería estar prohibido, salvo circustancia vies argumentada y la imposibilidad de hacerlo por otro medio.
- Usar las variables según el bloque al que corresponde. Si la variable i las usas constantemente en bucles for para recooresr arrays, inicia esa variable en ese bucle, así evitarás cambiar su valor por error.

# Clean Code: Resumen

- Nombre de Package: siempre en minúsculas.
- Nombre de las clases o interfaces: UpperCamelCase.
- Nombre de los métodos: lowerCamelCase. Suelen ser verbos o frases.
- Nombres de constantes: CONSTANT_CASE. Todo el mayúsculas, separando con barra baja.
- Variables locales, atributos de la clase, nombres de parámetros: lowerCamelCase.

- **CONSEJO:** Simepre intenta que al leerlas sepas lo que almacenas o las acción que realizas.

# Documentación

Cuando debemos aportar algo más

# Documentación

- “No comentes el código mal escrito, reescríbelo”. – Brian W. Kernighan
- Cuando necesitas añadir comentarios a tu código es porque este no es lo suficientemente autoexplicativo, lo cual quiere decir que no estamos siendo capaces de escoger buenos nombres o tu funciones tienen bastantes líneas. Cuando veas la necesidad de escribir un comentario, trata de refactorizar tu código y/o nombrar los elementos del mismo de otra manera.
- En todo caso, si necesitas hacer uso de los comentarios, lo importante es comentar el porqué, más que comentar el qué o el cómo. Ya que el cómo lo vemos, es el código, y el qué no debería ser necesario si escribes código autoexplicativo. Pero el por quéhas decidido resolver algo de cierta manera a sabiendas de que resulta extraño, eso sí que deberías explicarlo.

# Documentación

- “No comentes el código mal escrito, reescríbelo”. – Brian W. Kernighan
- Cuando necesitas añadir comentarios a tu código es porque este no es lo suficientemente autoexplicativo, lo cual quiere decir que no estamos siendo capaces de escoger buenos nombres o tu funciones tienen bastantes líneas. Cuando veas la necesidad de escribir un comentario, trata de refactorizar tu código y/o nombrar los elementos del mismo de otra manera.
- En todo caso, si necesitas hacer uso de los comentarios, lo importante es comentar el porqué, más que comentar el qué o el cómo. Ya que el cómo lo vemos, es el código, y el qué no debería ser necesario si escribes código autoexplicativo. Pero el por quéhas decidido resolver algo de cierta manera a sabiendas de que resulta extraño, eso sí que deberías explicarlo.

# Documentación

- “El buen código siempre parece estar escrito por alguien a quien le importa”. Michael Feathers
- En todo proyecto software debe existir una serie de pautas sencillas que nos ayuden a armonizar la legibilidad del código de nuestro proyecto, sobre todo cuando trabajamos en equipo. Algunas de las reglas en las que se podría hacer hincapié son:
  - **Problemas similares, soluciones simétricas**. Es capital seguir los mismos patrones a la hora de resolver problemas similares dentro del mismo proyecto. Por ejemplo, si estamos resolviendo un CRUD de una entidad de una determinada forma, es importante que para implementar el CRUD de otras entidades sigamos aplicando el mismo estilo.
  - **Densidad, apertura y distancia vertical**. Las líneas de código con una relación directa deben ser verticalmente densas, mientras que las líneas que separan conceptos deben de estar separadas por espacios en blanco. Por otro lado, los conceptos relacionados deben mantenerse próximos entre sí.
  - **Lo más importante primero**. Los elementos superiores de los ficheros deben contener los conceptos y algoritmos más importantes, e ir incrementando los detalles a medida que descendemos en el fichero.
  - **Indentación**. Por último, y no menos importante, debemos respetar la indentación o sangrado. Debemos indentar nuestro código de acuerdo a su posición dependiendo de si pertenece a la clase, a una función o a un bloque de código.

**CONSEJO:** Ten en cuenta que le mero hecho de escribir código implica que debes hacerlo de manera autopducmentada y entendible

# Documentación

- La documentación añade explicaciones de la función del código, de las características de un método, etc. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.
- Comentario de varias líneas /* Comentario */:  Se utilizan al principio de clases y métodos para explicar a nivel general algun aspecto especial de éstas que no puesa ser entendido por el propio uso o nombre que se le da a sus elementos, o simplemente para acotar y explicar parte de su solución.
- Comentarios de una línea // Comentario: Se utiliza si queremos remarcar una senttencia o un fragmento por interés especila o al que se le debe prestar atención en la acción que realiza.

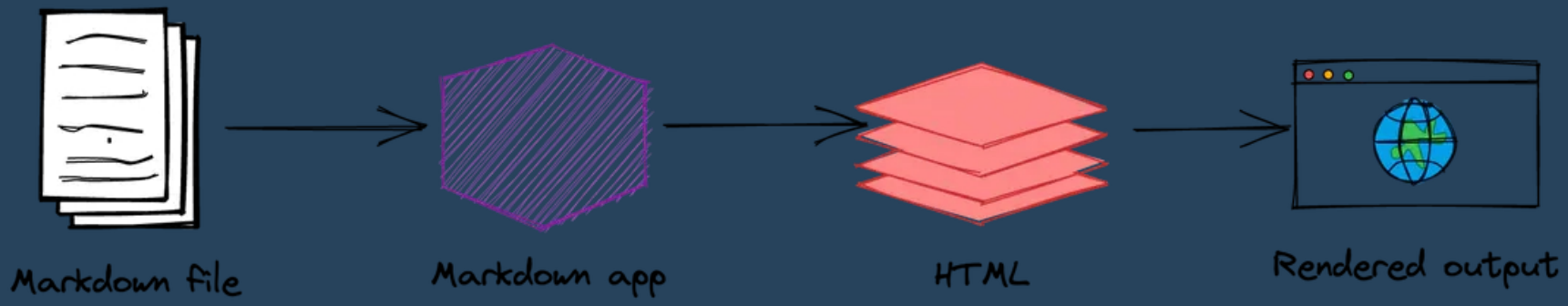
# Documentación: JDoc y JSDoc

- Comienzan con /** y terminan */. Deben tener una estructura estrablecida.
- **Los comentarios de una clase** deben comenzar con /** y terminar con */ . Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas @autor y @version, donde @autor identifica el nombre del autor o autora de la clase y @version, la identificación de la versión y fecha. También se suele añadir la etiqueta @see, que se utiliza para referenciar a otras clases y métodos.
- En Métodos, debemos usar  @param nombreParámetro descripció, para describir un parámetro de un método y @return descripción. para describir el valor de salida de un método.

- https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

# Documentación: Markdwon

- **Markdown** es un lenguaje de marcado ligero que puede usar para agregar elementos de formato a documentos de texto sin formato. Creado por John Gruber en 2004, Markdown es ahora uno de los lenguajes de marcado más populares del mundo. Se uytiliza par aescribir documentación técnica que puede mostrarse como páginas web, o simplemente para dar información de nuestros repositorios.
- Para escribir documentos en Markdown solo debemos tener un editor de textos.




- https://www.markdownguide.org/

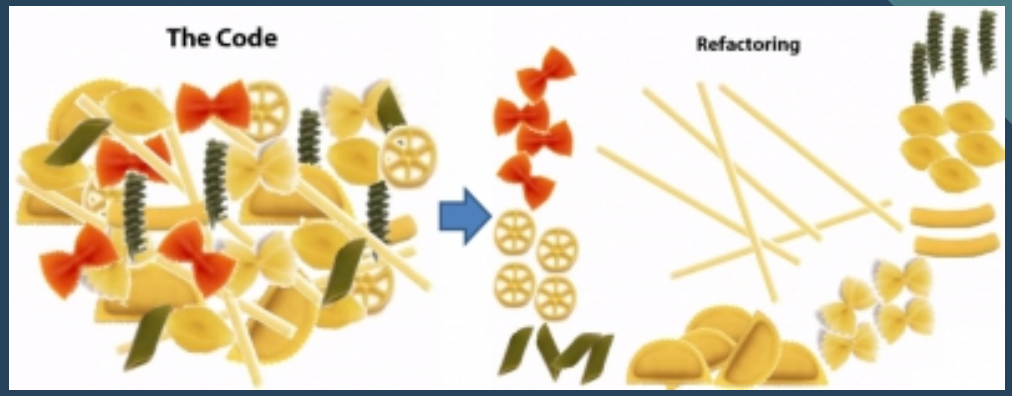
# Refactorización

Mejorando nuestro código

# Refactorización

- El término refactorizar dentro del campo de la Ingeniería del Software hace referencia a la modificación del código sin cambiar su funcionamiento. Se emplea para crear un código más claro y sencillo, facilitando la posterior lectura o revision de un programa. Se podría entender como el mantenimiento del código, para facilitar su comprensión, pero sin añadir ni eliminar funcionalidades. Refactorizar código consiste en crear un código más limpio.
- La refactorización debe ser un paso aislado en el diseño de un programa, para evitar introducir errores de código al reescribir o modificar algunas partes del mismo. Si después de refactorizar hemos alterado el funcionamiento del código, hemos cometido errores al refactorizar.
- Se refactoriza para:
  - Limpieza del código, mejorando la consistencia y la claridad.
  - Mantenimiento del código, sin corregir errores ni añadir funcionalidades.
  - Elimina el código “muerto”, y se modulariza.
  - Facilita el futuro mantenimiento y modificación del código.
  - Los siguientes apartados están inevitablemente relacionados entre si,

    ya que todas las técnicas o reglas persiguen el mismo fin.



# Refactorización: Code smell

- Se conoce como Bad Smell o Code Smell (mal olor) 1) a algunos indicadores o sintomas del código que posiblemente ocultan un problema más profundo. Los bad smells no son errores de código, bugs, ya que no impiden que el programa funcione correctamente, pero son indicadores de fallos en el diseño del código que dificultan el posterior mantenimiento del mismo y aumentan el riesgo de errores futuros. Algunos de estos síntomas son:
  - **Código duplicado** (Duplicated code). Si se detectan bloques de código iguales o muy parecidos en distintas partes del programa, se debe extraer creando un método para unificarlo.
  - **Métodos muy largos** (Long Method). Los método de muchas lineas dificultan su comprensión. Un método largo probablemente está realizando distintas tareas, que se podrían dividir en otros métodos. Las funciones deben ser los más pequeñas posibles (3 lineas mejor que 15). Cuanto más corto es un método, más fácil es reutilizarlo. Un método debe hacer solo una cosa, hacerla bien, y que sea la única que haga.
  - **Clases muy grandes** (Large class). Problema anterior aplicado a una clase. Una clase debe tener solo una finalidad. Si una clase se usa para distintos problemas tendremos clases con demasiados métodos, atributos e incluso instancias. Las clases deben el menor numero de responsabilidades y que esté bien delimitado.
  - L**ista de parámetros extensa (Long parameter list)**. Las funciones deben tener el mínimo número de parámetros posible, siendo 0 lo perfecto. Si un método requiere muchos parámetros puede que sea necesario crear una clase con esa cantidad de datos y pasarle un objeto de la clase como parámetro. Del mismo modo ocurre con el valor de retorno, si necesito devolver más de un dato.

# Refactorización: Code smell

- **Cambio divergente (Divergent change).** Si una clase necesita ser modificada a menudo y por razones muy distintas, puede que la clase esté realizando demasiadas tareas. Podría ser eliminada y/o dividida.
- **Cirugía a tiros (Shotgun surgery)**. Si al modificar una clase, se necesitan modificar otras clases o elementos ajenos a ella para compatibilizar el cambio. Lo opuesto al smell anterior.
- **Envidia de funcionalidad (Feature Envy)**. Ocurre cuando una clase usa más métodos de otra clase, o un método usa más datos de otra clase, que de la propia.
- **Legado rechazado (Refused bequest)**. Cuando una subclase extiende (hereda) de otra clase, y utiliza pocas características de la superclase, puede que haya un error en la jerarquía de clases.

# Refactorización: Consejos

- **Evita siempre que puedas el uso de else. Debes** priorizar el estilo declarativo, hacer uso de las cláusulas de guarda cuando uso estructuras condicionales o reemplazo las estructuras if/else por el operador ternario, lo que da lugar a un código mucho más comprensible y expresivo: isRunning ? stop() : run()
- **Prioriza las condiciones asertivas**. Las frases afirmativas suelen ser más fáciles de entender que las negativas, por esta razón deberíamos invertir, siempre que sea posible, las condiciones negativas para convertirlas en afirmativas.
- **Principio DRY (don't repeat yourself)**. Este principio, que en español significa no repetirse, nos evitará múltiples quebraderos de cabeza como tener que testear lo mismo varias veces, además de ayudarnos a reducir la cantidad de código a mantener. Para ello lo ideal sería extraer el código duplicado a una clase o función y utilizarlo donde nos haga falta.
- **Bajo Acomplamiento y Alta Cohesión**: La cohesión hace referencia a la relación entre los módulos de un sistema. En términos de clase, podemos decir que presenta alta cohesión si sus métodos están estrechamente relacionados entre sí. Un código con **alta cohesión** suele ser más self-contained, es decir contiene toda las piezas que necesita por lo tanto también suele ser más sencillo de entender. No obstante, si aumentamos demasiado la cohesión, podríamos tender a crear módulos con múltiples responsabilidades. El **acoplamiento**, en cambio, hace referencia a la relación que guardan entre sí los módulos de un sistema y su dependencia entre ellos. Si tenemos muchas relaciones entre dichos módulos, con muchas dependencias unos de otros, tendremos un grado de acoplamiento alto. En cambio, si los módulos son independientes unos de otros, el acoplamiento será bajo. Si favorecemos el bajo acoplamiento, obtendremos módulos más pequeños y con responsabilidades más definidas, pero también más dispersos. En el equilibrio está la virtud, es por ello que debemos tratar de favorecer el bajo acoplamiento pero sin sacrificar la cohesión.
- **Principio SRP. El principio de la Responsabilidad Única,** viene a decir que una clase/método debe tener tan solo una única responsabilidad. Las funciones deben hacer una única cosa y hacerla bien. Este principio lo usamos para refactorizar funciones de gran tamaño en otras más pequeñas, pero esto no aplica a la hora dediseñar clases o componentes, donde debemos analizar su responsabilidad dependiendo de su rol. Tener más de una responsabilidad en nuestras clases o módulos hace que nuestro código sea difícil de leer, de testear y mantener. Es decir, hace que el código sea menos flexible, más rígido y, en definitiva, menos tolerante al cambio.
- Siempre que puedas **usa Patrones de Diseño** y código estandarizado.

# Refactorización: Técnicas y Patrones

- Los métodos de refactorización, también llamados patrones de refactorización, nos permiten plantear casos y previsualizar las posibles soluciones que se nos ofrecen. Podemos seleccionar diferentes elementos para mostrar su menú de refactorización ( una clase, una variable, método, bloque de instrucciones, expresion, etc ). A continuación se muestras algunos de los métodos más comunes:
  - **Rename**: Es la opción empleada para cambiar el identificador a cualquier elemento (nombre de variable, clase, método, paquete, directorio, etc). Cuando lo aplicamos, se cambian todas las veces que aparece dicho identificador.
  - **Move**: Mueve una clase (fichero .java)de un paquete a otro y se cambian todas las referencias. También se realiza la misma operación arrastrando la clase de un paquete a otro en el explorador de eclipse.
  - **Extract Constant**: Convierte un número o cadena literal en una constante. Se puede ver donde se realizarán los cambios, y también el estado antes y después de refactorizar. Después, todas las apariciones de esa cadena se sustituyen por el nombre de la constante. Esto se utiliza modificar el valor en un solo lugar.
  - **Extract Local Variable**: Convierte un número o cadena literal en una variable de ámbito local. Si esa misma cadena de texto existe fuera del bloque o del método, no se aplica el cambio. Parecido al patrón anterior, pero para aplicar dentro de método o bloques de código entre llaves { }.
  - **Convert Local Variable to Field**: Convierte una variable local en un atributo privado de la clase. Después de aplicar el patrón de refactorización, todos los usos de la variable local se sustituyen por el atributo.

https://refactoring.guru/refactoring/techniques

# Refactorización: Técnicas y Patrones

- **Extract Method**: Convierte un bloque de código en un método, a partir de un bloque cerrado por llaves { }. Eclipse ajusta las parámetros y el retorno del método. Es muy útil cuando detectamos bad smells en métodos muy largos, o en bloques de código que se repiten.
- **Change Method Signature**: Permite cambiar el nombre del método y los parámetros que recibe. Se actualizarán todas las dependencias y llamadas al método dentro del proyecto actual.
- **Inline**: Nos permite ajustar una referencia a una variable o método en una sola linea de código.
- **Extract Interface:** Este patrón de refactorización nos permite seleccionar los métodos de una clase para crear una Interface. Una Interface es una plantilla que define los métodos acerca de lo que puede hacer una clase. Define los métodos de una clase (Nombre, parámetros y tipo de retorno) pero no los desarrolla.
- **Extract Superclass:** Permite crear una superclase (clase padre) con los métodos y atributos que seleccionemos de una clase concreta. Lo usamos cuando la clase con la que trabajamos podría tener cosas en común con otras clases, las cuales serían también subclases de la superclase creada.

# Optimización

Mejorando aun más si podemos

# Optimización

- La optimización de código nos ayuda a mejorar lka calidad del mismo. Todo pasa por tu proceso de Refactorización y de generación de código límpio.
- Si has llegado aquí, seguro que tu código esta bastante optimizado, pero vamos a darle otra vuelta con estos consejos:
  - Evita los swicth, si es posible usar Mapas y una función que leyendo las claves obtenga el valor que corresonde
  - Si un condicional o bucle tiene muchas condiciones, usa guardas y si es posible calculalas fuera en una función, con ello evitaras sentencias anidadas.
  - Usa siempre sentencias antes que condicionales evitando el else, por ejemplo el operador ternario: resultado = (condicion)?valor1:valor2. Ejemplo mayor=(x>y)?x:y;
  - Osar siempre que puesas foreach (variante de for) para recorrer los arrays, evitando problemas con el indice o que te equivoques con alguna condición.
  - No pidas nunca el valor size o lenght de una colección en la condición del bucle, siempre debe calcularla en cada iteración, si no cambia déjala fuera.
  - Siempre usa un return pero si es necesario y no puedes sal de la función pero analizando que no dejes código muerto.

# Conclusiones

- Estamos comenzando y poco a poco veremos nuevas tecnicas. Progrmar es un arte y requere de paciencia y esfuerzo. Escribe código limpio, usa comentarios sabiamente, refactorizar y optimizar es una obligación.
- Te recomiendo que poco a poco escribas código límpio y recuerda:

> *“Programa siempre tu código como si el tipo que va a tener que mantenerlo en el futuro fuera un violento psicópata que sabe dónde vives”. Martin Goldin*

El buen código es su mejor documentación"

-- Steve McConnell

# Recursos

- Twitter: https://twitter.com/joseluisgonsan
- GitHub: https://github.com/joseluisgs
- Web: https://joseluisgs.github.io
- Discord: https://discord.gg/WKKvSJCS
- Aula Virtual: https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=246

# Gracias

José Luis González Sánchez