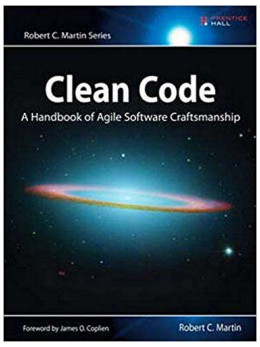
Clean Code: A Handbook of Agile Software Craftsmanship
(Robert C. Martin)



El código limpio (clean code), no es algo recomendado o deseable, es algo **vital para las organizaciones y los programadores**. La razón es que cada vez que alguien escribe código enmarañado y sin pruebas unitarias (código no limpio), otros tantos programadores pierden mucho tiempo intentando comprenderlo. Incluso el propio creador de un código no limpio si lo lee a los 6 meses será incapaz de comprenderlo y por tanto de evolucionarlo.

Para evitar este problema el autor Robert C. Martin propone seguir una serie de **guías y buenas prácticas a la hora de escribir el código**. En concreto se centra en el más bajo de los niveles, por ejemplo, en el formato, estilo, nomenclaturas, convenciones, etc. El libro no trata casi nada sobre patrones de diseño, arquitecturas ni tecnologías concretas. Aunque los ejemplos están en Java, se puede aplicar perfectamente a otros lenguajes.

Pese a ser un libro de referencia y no una novela, el libro tiene 2 partes diferenciadas más un capítulo final recopilatorio:

- Capítulos del 1 al 13: Explicación de buenas prácticas de código limpio, qué hay que hacer para obtener código limpio.
- Capítulos del 14 al 16: Plantean ejemplos de situaciones reales en las que aplicar lo explicado en la teoría.
- Capítulo 17: Recopilación de síntomas y heurísticas para detectar código no limpio, donde se repite gran parte del contenido de la teoría.

1. Introducción

La responsabilidad del código limpio es 100% de los programadores, su labor es defenderlo y negarse a hacer código incorrecto (¿dejaría un médico que le pidieran que no se lavara las manos antes de una operación?). El código limpio es la única alternativa para avanzar.

El código limpio se caracteriza por ser elegante, eficaz, legible, mínimo, hacer solo una cosa bien de una única manera y tener pruebas unitarias. Metáfora de las ventanas rotas, el código incorrecto es el principio del desastre. **Leemos 10:1 más código del que escribimos,** no

podremos escribir código si el de alrededor es ilegible. Regla del Boy Scout, dejar el código más limpio de lo que me lo encontré.

2. Nombres con sentido

Todos los nombres deben ser intencionados y descriptivos. Evita abreviaciones, prefijos, usar secuencias de números en variables y las palabras redundantes (the-, a-, -object, -info, -data). Usa nombres que se puedan buscar (evita variables a, e, l… usar i, j, k solo para bucles cuyo contexto sea muy acotado).

Es mejor usar un código en la implementación (que será privada y usada menos veces) que la interfaz => IShapeFactory, ShapeFactory —> ShapeFactory, ShapeFactoryImp

Nombres de clases: Evitar sufijos Manager o Processor. No deben ser verbos.

Nombres de métodos: Utiliza verbos. Usa get y set para métodos de acceso e is para booleanos. Usa métodos estáticos con nombres con el tipo de argumento en lugar de sobrecargar constructores (new Complex(23.0) –> Complex.FromRealNumber(23.0))

No usar juegos de palabras, chistes o sutilezas. Intentar usar siempre las mismas palabras para lo mismo (get, fetch, retrieve?). No usar una misma palabra para cosas distintas (add es sum o insert?).

Usa nombres técnicos cuando la intención sea técnica (Factory, Visitor, Facade, …) y nombres de dominio para conceptos de dominio.

Añade contexto a las variables, por ejemplo, agrupándolas en clases.

Los nombres cuanto más cortos mejor, siempre que sean explícitos y claros.

3. Funciones

Deben ser reducidas (~20 líneas) y con nombres descriptivos (no importan si son un poco largos). Debemos evitar el anidamiento excesivo (~complejidad ciclomática). Solo deben hacer una cosa. Para adivinar si una función hace más de una cosa intentamos describirla en una frase: “PARA (nombredefunción) […]”. Ej: “Para RenderPageWithSetup comprobamos si la página es de prueba y en caso afirmativo añadimos configuración y detalles. En cualquier caso, renderizamos en HTML”. Todo el contenido de una función debería estar al mismo nivel de abstracción. De igual forma, el siguiente nivel de abstracción debería estar en la función que sigue. De esta forma una clase se puede leer secuencialmente de arriba hacia abajo.

Evitar instrucciones switch en funciones escondiéndolas en clases abstractas (patrón Factory+strategy).

No deberían tener más de 2 parámetros, 3 ya son muchos y más de 3 una excepción que se debe justificar. Evitar los parámetros de salida pues son confusos, mejor funciones que retornen valor o que se llame a una función de clase del objeto que se cambia. Argumentos booleanos evitarlos, son síntoma de que la función hace al menos 2 cosas (1 si es true y otra si es false), en este caso mejor hacer 2 funciones, una para cada caso.

Funciones con 2 argumentos son válidas si los 2 argumentos están relacionados naturalmente, si no tienen nada que ver resultará confuso. Peor aún 3. Si podemos relacionarlos mejor hacer clases específicas con ellos que tengan sentido (Ej: Circle makeCircle(float x, float y, float radius) –> Circle makeCircle(Point p, float radius)).

Funciones sin efectos secundarios, que hagan lo que dice su nombre y nada más oculto (Ej: checkPassword que también llama a initializeSession). Normalmente asociado a hacer una sola cosa.

Funciones de comando o de consulta, pero nunca combinadas (solo una cosa).

Mejor devolver excepciones que códigos de error siempre (no implica capturar las excepciones inmediatamente y el código resulta más simple).

Si hay un bloque try/catch mejor simplificarlos extrayendo funciones para cada uno de los bloques (ej: try { hagoAlgoQuePuedeFallar(); } catch (Excepcion ex) { Logger.log(ex.message); }). Si hay try/catch en una función no debería haber nada más, si no seguro se está haciendo más de una cosa.

Todas las reglas que se han descrito son muy complicadas de seguir a la primera, la primera vez que se escribe una función es común violar la mayoría de reglas, aunque con trabajo iterativo adicional siempre se puede mejorar.

4. Comentarios

Los comentarios solo están justificados cuando no somos capaces de expresarnos con el código. En general, basta con escribir y encapsular todo en una función que se llame como lo que hay en el comentario.

Solo algunos comentarios son positivos:

- Comentarios legales, de derechos de autor
- Comentarios informativos de lo que devuelve una función, aunque también se pueden eliminar si en el nombre de la función especificamos lo que se devuelve
- Explicación de la intención, decisión tomada o advertencia
- Cuando utilizamos librerías de terceros, que no podemos modificar los nombres de funciones
- Comentarios TODO, aunque no deben ser excusa para dejar código incorrecto
- Comentarios en API públicas (Javadoc)

Comentarios incorrectos: Que dejan dudas en la explicación, redundantes, obligatorios en javadoc, de registro de cambios (para ello está el control de código fuente), código comentado y comentarios sobre cosas que no están en el código adyacente.

5. Formato

El tamaño de los ficheros no debería superar las 200 líneas de media, con un límite máximo en 500.

Metáfora del periódico:

- Una clase comienza con un título descriptivo y una descripción sin detalles que explica el contenido.
- Después vienen los detalles.
- El módulo o programa es una compilación de artículos grandes y pequeños.
- Podemos entender la clase con los métodos superiores sin necesidad de ahondar en los detalles
- Separación de pensamientos y bloques con líneas en blanco (como separar en párrafos).

La distancia vertical entre elementos relacionados debe ser mínima, y como consecuencia, evitar separar elementos relacionados en 2 ficheros distintos (razón por la cual no se deben usar variables protected)

Las variables se deben declarar lo más cerca posible a su uso. Las variables de clase (de instancia) en la parte superior, ya que se usan en toda la clase.

La anchura de las líneas de código, entre 80 y 120 caracteres, no deberíamos hacer scroll horizontal para leer código.

Los espacios en blanco se usan para separar conceptos que queremos remarcar o que están desconexos con sus antecesores y predecesores. Ej: double determinant(double a, double b, double c) { return b*b – 4*a*c; }

Intentar no romper el sangrado aunque la longitud de la función o while sea mínima, lo importante es la claridad.

Cualquier equipo debería tener unas reglas convenientemente consensuadas. Es importante seguirlas por parte de todos, el estilo y formato debe ser siempre el mismo ya que el código es compartido.

6. Objetos y estructuras de datos

Abstracción de los datos, se debe esconder la implementación de los datos de una clase y generar una interfaz (normalmente mediante métodos) para acceder y establecerlos, siempre escondiendo la implementación concreta.

Clase (esconde su implementación interna) vs estructura de datos (los expone tal cual)

Ley de Demeter. Solo debe invocar funciones de:

1. Sí mismo
2. Variables locales
3. Un argumento
4. Una variable de instancia

Y no debe invocar funciones de objetos devueltos por llamadas a otras clases
(ej: a.getX().getY().getValue();)

Cuando tengamos esta necesidad hay que crear un método que devuelva el objeto de terceros que necesitamos (a.getXYValue();), o mejor, preguntarnos para qué necesito estos datos y traspasarle quizás la responsabilidad de ejecutar la acción al objeto a usar (a.doSomethingWithXYValue();). La ley de Demeter solo aplica a objetos, no a estructuras de datos simples.

DTO = ejemplo de estructuras de datos útiles

Active Record = DTO con métodos save y find, siguen sin ser objetos, son meras representaciones de un origen de datos, si queremos añadir métodos con lógica de negocio debemos crear clases aparte con variable de instancia interna de DTO

7. Procesar errores

No usar códigos de error ya que confunden el flujo de ejecución y obligan al invocador a procesarlos inmediatamente.

En los errores incluir información que nos dé contexto de dónde se ha producido el fallo.

Al usar APIs de terceros siempre envolver excepciones (patrón **Facade**).

Crear clases para los casos especiales en lugar de dejar al código cliente procesar el caso excepcional (patrón caso especial, Fowler).

En general no es recomendable devolver null, en su lugar es mejor devolver una excepción o un objeto de caso especial.

Tampoco se debe pasar null como parámetro, a no ser que una librería de terceros espere un null. Al no haber una forma racional de controlar null para parámetros, evitarlo por convención es la mejor solución posible.

8. Límites

No es conveniente utilizar clases genéricas del sistema como valores de retorno de una de nuestras API. Normalmente tienen un exceso de funcionalidad que nuestro cliente no necesitará. Además, si la clase genérica cambia (poco probable pero no imposible), tendremos que cambiar el código de todos los clientes. Es mejor encapsular la implementación con la clase genérica en una clase propia que será la que usen los clientes.

Si usamos código de terceros, generar unit tests de sus interfaces para comprobar que entendemos cómo funciona y se comporta como esperamos. Además, esto nos sirve para comprobar si nuevas versiones mantienen compatibilidad con lo que usamos de esa librería.

Para usar cualquier librería de terceros usar el patrón **Adapter** (además permitirá comenzar la implementación sin tener aún el sistema con un FakeAdapter y hacer unit tests).

Todo esto va enfocado a tener el mínimo de puntos a tocar si se produce algún cambio en el límite.

9. Pruebas unitarias

3 leyes de **TDD**:

1. No hay que crear código hasta que haya fallado un Unit Test (test unitario)
2. No hay que crear nunca más de una prueba que falle
3. El código creado debe ser el mínimo para que la prueba pase

~30 segundos en hacer los 3 pasos aprox.

Las pruebas al tener que evolucionar al mismo ritmo que el código, deben ser igualmente mantenibles y respetar las mismas reglas de código limpio.

Las pruebas son la clave del desarrollo ágil, permiten hacer cambios sin temor a romper nada que funcione.

En las pruebas es todavía más importante la legibilidad que en el código de producción. Evitar métodos muy largos con todos los detalles de implementación, es mejor que se lea claramente la estructura Arrange-Act-Assert de las pruebas escondiendo los detalles en métodos. Es común refactorizar código de unit tests y acabar en una API de pruebas. También es común penalizar el rendimiento a favor de la legibilidad ya que las pruebas nunca se ejecutarán en entorno productivo.

Está permitido hacer más de un assert en una prueba, pero sí se debe cumplir que el número de asserts sea mínimo. Lo que sí se debe cumplir siempre es que solo se prueba una cosa en cada test.

5 reglas para pruebas limpias: FIRST

1. **F**ast
2. **I**ndependent, si son dependientes provocarán un fallo en cascada
3. **R**epetition, se deben poder repetir en cualquier entorno, incluso sin red
4. **S**elf-Validating, o aciertan o fallan
5. **T**imely, se hacen antes del código porque si la haces después te dará pereza y acabarás por no probar

10. Clases

Orden dentro de la clase: Contantes estáticas, variables estáticas, variables de instancia y funciones. De todo ello, primero lo público y después lo privado.

El tamaño debe ser reducido, debe tener una única responsabilidad, la que indica su nombre. Nombres a evitar son Manager, Processor, Super ya que denotan muchas responsabilidades.

Single Responsibility Principle, una clase debe tener un único motivo para cambiar.

Cuando organizamos la complejidad del software, es mejor organizarla en cajones pequeños bien etiquetados que no en cajones de sastre enormes.

Cohesión = grado de utilización de las variables de instancia por parte de las funciones. Queremos clases cohesionadas. Cuando se reduce el tamaño de las funciones se aumenta el tamaño de variables de instancia (para no pasarlas como parámetro a las subfunciones) y se pierde cohesión. En ese caso lo mejor es dividir en subclases.

Open/Closed Principle = las clases deben estar abiertas a extensión y cerradas a modificación. Los cambios mejor que se hagan extendiendo o introduciendo nuevas clases, no modificando las existentes.

Dependency Inversión Principle = las clases dependen de abstracciones, no de detalles concretos –> Imprescindible para unit testing.

11. Sistemas

Metáfora de la ciudad como el sistema, las responsabilidades se separan y una persona no puede controlar absolutamente todo y conocer todos los detalles. Las ciudades además se construyen poco a poco, las carreteras y demás servicios se van ampliando y acomodando a la demanda. Esto en software es todavía más fácil.

Separar el proceso de construcción e inicio del uso del sistema o clase.

Necesitamos un sitio centralizado donde abordar la construcción y resolver las dependencias. Una estrategia posible es generar todo desde main para después pasárselo a la aplicación (ya que la configuración es un aspecto global va a main).

Patrón **Factoría Abstracta** –> Separar en una clase la responsabilidad de crear un objeto concreto para esconder los detalles de la creación.

Inyección de dependencias –> Un objeto no es responsable de instanciar sus dependencias, lo delega a un sistema autorizado (normalmente main o un contenedor autorizado).

AOP –> Modularizar aspectos transversales como persistencia, caché, excepciones… (Ejemplos, Spring, AspectJ)

Una correcta separación de aspectos y modularidad es esencial para empezar proyectos software en pequeño e ir creciendo a medida que lo necesitemos.

Arquitectura óptima = dominios implementados con **POJO** y conectados y armonizados con aspectos transversales mínimamente invasivos.

12. Emergencia

Según Ken Beck un diseño es sencillo cuando:

- Ejecuta todas las pruebas: Si se puede probar estamos seguros que hace lo que debe. Además, el diseñar muchas pruebas nos fuerza a hacer clases más pequeñas y con menos responsabilidades. Cuantas más pruebas, mejor diseño simple.
- No contiene duplicados (REFACTORIZAR): Patrón de Template Method es muy usado para eliminar código duplicado.
- Expresa la intención del programador (REFACTORIZAR)
- Minimiza el número de clases y métodos (REFACTORIZAR): Es la menos importante de las 4, queremos sistemas simples o por ello con menos clases y métodos, pero no a costa de perder pruebas, aumentar duplicados o perder expresividad.

En la fase de refactorización es donde realmente aplicamos todos los aspectos vistos de código limpio: aumentar cohesión, dividir clases, modularizar aspectos, elegir nombres adecuados… Es imposible hacer un código limpio a la primera, así que es necesario refactorizar después de que el código funcione, y es necesario hacerlo antes de pasar a la siguiente tarea.

13. Concurrencia

El hecho de separar el qué del cuándo en programación concurrente hace que tengamos que tomar precauciones para no tener problemas. Las prácticas a adoptar son:

- SRP, un único motivo para cambiar en cada clase –> separar el código que controla la concurrencia del código de la aplicación
- Encapsula datos y protégelos con synchronized cuando se compartan entre procesos
- Intentar minimizar la compartición de datos entre procesos, los procesos deben ser independientes
- Utiliza las clases del entorno de desarrollo específicas de concurrencia (ej. Java.util.concurrent)

Distintos modelos: productor-consumidor, lector-escritor, la cena de los filósofos (condiciones de carrera), …

Planificar y probar concienzudamente el código de cierre de un proceso, para evitar bloqueos en nuevos procesos.

Recomendaciones:

- Primero que funcione sin procesos, esto permite identificar los fallos que no tienen que ver con concurrencia

- No ignorar los fallos que no se pueden reproducir
- Se tienen que poder probar, mejor diseñar con esto en mente

14. Refinamiento sucesivo

Ejemplo de **refactoring** de una aplicación que parsea args pasados a main de 3 tipos (boolean, string e integer).

El refactor consiste en hacer cambios pequeños poco a poco y que en cada paso pequeño pasen todos los tests.

Los pasos concretos del refactor:

- Identificar el código o patrón de código que se repite. En este caso para cada tipo teníamos un parser de schema, un parser del valor y una función getBoolean(), getInteger()…
- Creamos la clase ArgumentMarshaler como clase base, por ahora la clase no es abstracta ya que para empezar tendrá los métodos de tipo boolean (cambio más pequeño posible)
- Introducimos instancias de la clase ArgumentMarshaler en lugar de los valores parseados de boolean y hacemos los retoques para que pase el compilador. Todo sigue funcionando.
- Añadir los métodos de integer y string en ArgumentMarshaler. Aunque sea la clase base lo primero es que el programa siga funcionando y antes de extraer clases heredadas de la clase base necesito toda la lógica en la clase base. Todo el marshaling está ahora en una sola clase, la clase base ArgumentMarshaler.
- Siguiente paso es pasar el código de boolean a BooleanArgumentMarshaler. Para ello hacemos ArgumentMarshaler clase abstracta y el parsing lo pasamos a funciones abstractas get y set. Cambiamos las instancias de ArgumentMarshaler por BooleanArgumentMarshaler en los casos de boolean.
- Al pasar a abstracta necesito devolver Object en el método get() y hacer un Cast. Añado el control de la excepción ClassCastException.
- Repetimos el proceso para integer y string.
- Nos disponemos a eliminar los mapas de args por tipo, booleanArgs, stringArgs e intArgs. Para ello añado un mapa nuevo para los marshalers.
- Puedo eliminar parámetros de funciones con este mapa, ya que con el id del arg puedo recuperar del mapa el marshaler

- Paso parte del código de try catch de los diferentes marshalers de tipo a la clase base para no repetir código. En el marshaler de tipo mantengo la gestión de excepciones propia de cada tipo.
- Queremos eliminar el switch de tipos en setArgument y cambiarlo por Marshaler.set. Primero se hace con boolean y cuando funciona con string e integer.
- Al no quedar ningún método abstracto en la clase base podemos convertirla a interfaz.
- Para comprobar que el código es limpio y escala, añadimos el tipo double.
    - Primero hacemos un test que pase el happy path y añadimos la clase DoubleArgumentMarshaler y los cambios mínimos para que pase el test.
    - A continuación, hacemos un test por cada caso que pueda fallar y en cada uno vamos añadiendo código de control del error.
- El número de excepciones posible ha crecido mucho. Es mejor separar toda la lógica de excepciones en la clase ArgsException.
- Finalmente movemos cada nueva clase a su archivo.

15. Aspectos internos de JUnit

Ejemplo de refactor con un módulo llamado ComparisonCompactor. Los cambios más importantes que hacemos son:

- Mejorar los nombres a las variables
- Extraer una función de un condicional if para que nuestra intención quede clara (ej: if (a && b ==0 && c<30) –> if (shouldBeCompacted()). También se cambia el sentido del if para que la condición sea positiva (if (shouldNotBeCompacted()) –> if (shouldBeCompacted())
- Renombrar función, ya que no solo compacta sino también formatea (compact() –> formatCompactedComparison(string message)).
- Separamos la función en 2. formatCompactedComparison() formatea y compactExpectedAndActual() compacta
- Vemos que findCommonPrefix y findCommonSuffix se tienen que llamar en orden. Para marcar esto hacemos que findCommonSuffix reciba preffixIndex como parámetro.
- Nuevamente partimos findCommonPrefixAndSuffix() en 3 funciones, uno con una condición de if para que quede más claro y las otras 2 para extraer
- Cambiamos las variables preffixIndex y suffixIndex por preffixLength y suffixLength para que sea más claro, y movemos la resta -1 a los lengths a la función charAtEnd() que es donde se utilizan los lengths (para la diferencia de base 0 y base 1).

- Parece que sobran 2 ifs que comprueban longitud 0. Comentamos las líneas, ejecutamos pruebas y efectivamente en verde. Las eliminamos.

16. Refactorización de SerialDate

Ejemplo de refactor de la clase SerialDate de Jcommon, que implementa una clase más sencilla que java.util.Date ya que no incorpora información de las horas. Los pasos más importantes son:

- Cambiar el nombre de la clase por DayDate para esconder su implementación (la fecha se codifica con un número de serie)
- Eliminar la herencia de MonthConstants y crear mi propio enum. Implica muchos cambios, pero elimino comprobaciones de códigos incorrectos y una función.
- Existen 2 constantes MINIMUM_YEAR_SUPPORTED y MAXIMUM_YEAR_SUPPORTED públicas que no deben existir, ya que es una clase abstracta y no debe mostrar detalles de su implementación. El problema es que se usa en un método getFollowingDayOfWeek() que devuelve una clase concreta que hereda de DayDate, SpreadsheetDate. Para evitar que una clase abstracta deba conocer detalles de sus clases usuarias, utilizamos abstract factory DayDateFactory para que se encargue de gestionar la creación de instancias de DayDate y de los detalles de implementación.
- Simplificamos y aumentamos expresividad de algunas funciones
- Cambio addDays ya que es estática y opera variables de instancia, la pasamos a no estática.
- Ídem con addMonths, y además lo hacemos más explícito con la explicación de variables temporales. Ídem addYears.
- Cambio addDays y demás por plusDays para denotar que la instancia no sufre ningún cambio (solo se calcula el valor)
- Cambiamos weekInMonthToString por la función toString dentro de una nueva enumeración weekInMonth
- Cambiamos el método abstracto toDate que convierte a java.util.Date, no debería ser abstracto porque no depende de la implementación concreta. Lo subimos a DayDate.
- Eliminamos comentarios de demás clases abstractas
- Finalmente muevo todos los enums y clases nuevas a ficheros independientes
- También las funciones estáticas resultantes las moveremos a una clase DateUtil.

- También sustituimos el número mágico 1 por Month.JANUARY.toInt() y Day.toSunday.toInt().

17. Síntomas y heurística

Listado de olores que identifican problemas en el código:

Comentarios:

- Informativos
- Redundantes
- Obsoletos
- Mal escritos
- Código comentado

Entorno:

- La generación de la solución (build) desde el control de código fuente es compleja, requiere más de un paso
- Las pruebas requieren igualmente más de un paso, debería ser algo rápido y sencillo o no lo ejecutará con la frecuencia necesaria

Funciones:

- Demasiados argumentos, 0, 1, 2 o como mucho 3
- Argumentos de salida
- Argumentos de indicador (booleanos), son indicadores de que la función hace más de una cosa
- Función muerta (nadie la usa)

General:

- Varios lenguajes en un mismo archivo de código
- Comportamiento evidente no implementado, cuando un usuario espera que una función haga algo y ésta no lo hace se pierde la confianza y obliga a mirar qué hace el método por dentro
- Comportamiento incorrecto de la función (en los límites, en casos extremos, excepciones…)
- Medidas de seguridad canceladas temporalmente, nunca desactivar los tests, avisos del compilador, …

- Duplicación (DRY): Los códigos repetidos se pasan a clases abstractas, los switchs con polimorfismo, y los algoritmos repetidos se cambian con el patrón plantilla o estrategia.
- Código en un nivel de abstracción incorrecto (clase base abstracta sabe de lo general, clase heredada conoce los detalles)
- Clases base que dependen de sus variantes
- Exceso de información, es necesario ocultar la información de dentro de las clases, no queremos conocer los detalles, queremos saber lo mínimo y poder hacer lo máximo (interfaces pequeñas)
- Código muerto (condiciones if o catch que nunca se dan)
- Separación vertical estrecha
- Incoherencia (si usas una convención o nombre para algo mantenlo en todo el código igual)
- Desorden, elimina variables sin usar, ordena las declaraciones de variable, …
- Conexiones artificiales, por comodidad dejar las variables fuera de sitio, funciones, …
- Envidia de características, cuando una clase accede a get y set de una instancia de otra clase más de lo necesario es que envidia esa clase, es candidata a pasar a la clase envidiada, aunque a veces no hay más remedio
- Argumentos boolean (selectores) que denotan varios comportamientos en una función (mejor separar en n funciones)
- Intención desconocida o escondida, el código debe dejar clara la intención del autor
- Responsabilidad desubicada, dejar los elementos donde el lector espera encontrarlos
- Elementos estáticos incorrectos, una función será estática solo si no accede a variables de instancia o si no tiene sentido que sea polimórfica
- Variables explicativas, usar variables temporales con nombres descriptivos para explicitar nuestra intención, aunque el código quede más largo
- Nombres claros de función, si tenemos que leer documentación o mirar el código para ver qué hace, es candidata a cambiar de nombre
- Comprender el algoritmo, más allá que pase las pruebas asegurarnos de que es correcto
- Convertir dependencias lógicas en físicas (tiene que ver con funciones o constantes desubicadas)

- Polimorfismo antes que if o switch, como regla general solo puede haber un switch por cada tipo de selección, si se repite el switch en otra parte del código es síntoma claro de que necesita polimorfismo
- Seguir las normas de código
- Sustituir números mágicos por constantes con nombres correctos, también aplica a strings
- Precisión a la hora de tomar decisiones (utilizar float o int, return a[0], …)
- Encapsular condicionales, evitar poner en los if cadenas de booleanos, mejor una subfunción con un nombre explícito
- Evitar condicionales negativas
- Funciones solo deben hacer una cosa
- Las conexiones temporales deben ser visibles, por ejemplo, pasándole como parámetro el resultado de la primera función a la segunda, …
- Evitar la arbitrariedad, las cosas en su sitio y que se vea claro por qué están es ese sitio
- Las funciones solo deben bajar un nivel de abstracción
- Mantener los datos configurables en los niveles superiores (las constantes por ejemplo)
- Evitar desplazamientos transitivos (Ley de Demeter)

Java:

- Evitar usar comodines
- No heredar constantes (es complicado después saber de dónde vienen, mejor declararlas en una clase como estáticas y llamarlas con el prefijo apropiado, o usar importación estática)
- Usar enums en lugar de constantes

Nombres:

- Usar nombres descriptivos
- Elegir nombres correctos para el nivel de abstracción
- Usar nomenclatura estándar (usar nombres de patrones, nombres de funciones existentes tipo clone o toString, nombres del dominio…)
- Nombres inequívocos para funciones, mejor largos pero que expresen lo que hacen
- Usar nombres extensos para ámbitos extensos

- Evitar codificaciones en los nombres
- Deben describir efectos secundarios

Tests:

- Pruebas suficientes, se debe probar todo lo que pueda fallar
- Usar herramientas de cobertura
- No ignorar pruebas triviales, sobre todo por su labor de documentación
- Usar pruebas ignoradas (@Ignore) para preguntar sobre ambigüedades
- Probar condiciones de límite
- Si detecta un error en una función, pruébela exhaustivamente porque es probable que haya más
- Los patrones de fallo de las pruebas (también los de cobertura) son indicadores de una posible solución
- Las pruebas deben ser rápidas

"Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code. But it doesn't have to be that way. Noted software expert Robert C. Martin presents a revolutionary paradigm with Clean Code: A Handbook of Agile Software Craftsmanship. Martin has teamed up with his colleagues from Object Mentor to distill their best agile practice of cleaning code "on the fly" into a book that will instill within you the values of a software craftsman and make you a better programmer-but only if you work at it. What kind of work will you be doing? You'll be reading code-lots of code. And you will be challenged to think about what's right about that code, and what's wrong with it. More importantly, you will be challenged to reassess your professional values and your commitment to your craft. Clean Code is divided into three parts. The first describes the principles, patterns, and practices of writing clean code. The second part consists of several case studies of increasing complexity. Each case study is an exercise in cleaning up code-of transforming a code base that has some problems into one that is sound and efficient. The third part is the payoff: a single chapter containing a list of heuristics and "smells" gathered while creating the case studies. The result is a knowledge base that describes the way we think when we write, read, and clean code. Readers will come away from this book understanding. How to tell the difference between good and bad code How to write good code and how to transform bad code into good code How to create good names, good functions, good objects, and good classes How to format code for maximum readability How to implement complete error handling without obscuring code logic How to unit test and practice test-driven development. This book is a must for any developer, software engineer, project manager, team lead, or systems analyst with an interest in producing better code."