CLEAN CODE

{

RESUMEN EN ESPAÑOL

}

INTRODUCCIÓN

En la portada de este capítulo hay una imagen divertida, que dice: ‘The only valid measurement of code quality is WTFs/minute. Y se ven 2 puertas una que dice ‘Good code’ y otra que pone ‘Bad code’, y nos hace una pregunta: ¿En qué puerta está nuestro código o nuestro equipo o empresa?

Nos habla acerca de que la única manera de escoger la puerta correcta es mediante la maestría, que no es más que conocimiento y trabajo. Para escribir código limpio no solo es necesario conocer patrones y estilos de diseño de software y saberse la teoría, hay que practicarlo, hay que fallar y volver a practicarlo, solo así se podrá escribir buen código.

Yo haré un resumen teórico del libro, pero si te gusta este resumen te recomiendo que te compres el libro y empieces a practicar los casos descritos en él para convertirte en un buen desarrollador.

Comenzamos con el capítulo 1.

CAPÍTULO 1 - CÓDIGO LIMPIO

Este libro trata sobre programación correcta, este libro es para todos los programadores y para todos los que quieran ser mejores programadores. El objetivo del libro es aprender a diferenciar entre código malo y código bueno, y poder transformar código malo en bueno.
Hay quien piensa que en algún momento de nuestras vidas el código desaparecerá, que simplemente tendremos requisitos y las máquinas los harán por nosotros, nada más lejos de la realidad, esto es improbable que suceda, dado que no existe máquina que pueda entendernos al 100% o incluso descifrar las ambigüedades de muchos requisitos. Podemos crear lenguajes que entiendan casi todos los requisitos, pero siempre será necesaria una precisión exacta que solo se puede dar programando.

Una empresa en la década de 1980 comercializó una gran aplicación de software que fue muy vendida, pero poco a poco las actualizaciones fueron distanciando y la gente dejando de usar el producto, ¿por qué? La razón que dió uno de los empleados, fue que comercializaron el software antes de tiempo con fallos en el código, poco después la empresa desapareció y fue a raíz del mal código, de ahí la importancia del buen código.

Si se trabaja en un equipo de desarrollo y se avanza muy rápido sin prestar atención al código, puede que al final avancen a paso tortuga, cada cambio afecta a demasiadas cosas y con el paso del tiempo el software se vuelve inmantenible, por tanto la productividad del equipo baja, y ¿que hace el director? lo único que sabe, aumentar la plantilla con el fin de aumentar la productividad, pero estos nuevos programadores no entienden el sistema y menos el código que han escrito los demás y aunque lo comprendan se encontrarán en la misma situación pero con menos tiempo, por lo que la productividad del equipo bajaría a cero. Dedicar tiempo a que el código sea correcto, no solo es rentable, es una cuestión de supervivencia profesional.

¿Y de quién es la culpa? ¿De los jefes, directores o comerciales? No, la culpa es nuestra la de los programadores por no ser buenos profesionales, ¿y los plazos de entrega y los requisitos? Nosotros como profesionales debemos informar a nuestros jefes de la situación, ellos también quieren código correcto aunque tengan que preocuparse por los plazos de entrega también, nosotros debemos defender a capa y espada nuestra calidad de código, o ¿le parecería correcto que un médico no se lavara las manos antes de una operación porque pierde mucho tiempo? A que no, pues lo mismo ocurre con el código.

La única forma de alcanzar los plazos de entrega es escribiendo buen código, y esto lo saben los grandes profesionales. Escribir código limpio y correcto es como pintar un cuadro, cualquiera puede apreciar la calidad de un cuadro, pero no por saber que está bien pintado significa que sepamos pintar. El código limpio debe ser elegante, simple, directo, fácil de leer como un libro y de mantener, que contenga test unitarios y de aceptación, el código debe ser de un tamaño mínimo, el código limpio siempre parece que ha sido escrito por alguien que le ha importado el código, el código debe ser al leerse lo que esperamos que sea, ¿cuándo fue la última vez que leíste un código y era lo que esperabas, o no te supuso esfuerzo comprenderlo?, son varias de las conclusiones a las que han llegado los grandes profesionales de software sobre lo que debe ser el código limpio.

Recuerde que cuando escribe código eres el autor de un código que tendrá lectores, pasamos mucho más tiempo leyendo código que escribiendo código en nuestras vidas.
Existe una regla de los boy scouts de norteamérica, que dice que hay que dejar el campamento más limpio de lo que lo encontramos, de esta forma, siguiendo esta sencilla regla, si cada vez que vemos código incorrecto, lo hacemos más legible, cambiamos el nombre de una variable, agregamos un test, etc. Estaremos contribuyendo a mejorar el código, este libro no es la verdad absoluta y solo presenta técnicas que por sí solas no lo convertirán en un buen programador, para ello deberá practicar.

CAPÍTULO 2 - NOMBRES CON SENTIDO

Los nombres estan en todas partes, clases, variables, ficheros, etc. Elegir nombres relevantes es de suma importancia, y aunque pareciera algo de sentido común, ni es tan común ni es tan sencillo.
Cada vez que encuentre un nombre de alguna variable que se puede mejorar, hagalo, la gente que lea su código se lo agradecerá. Los nombres deben revelar nuestras intenciones.

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

Evite usar por ejemplo abreviaciones, como hp para hipotenusa, puede parecer correcto pero no lo es, no haga referencia a un grupo de cuentas como *accountList* porque si luego el tipo de dato cambia y no es una lista puede confundir, lo mejor sería usar *accounts* o *accountsGroup*. No utilice tampoco nombres que se diferencian mínimamente, por ejemplo entre *XYZAlmacenamientoDeCuentas* y *XYZManejadorDeCuentas* existe una sutil diferencia que nos puede confundir o hacernos equivocar en un módulo.

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

En el libro, también se recomienda evitar usar la I mayúscula, como prefijo para las interfaces, por ejemplo, supongamos que tenemos la interfaz *ShapeFactory*, es preferible colocar en la implementación *ShapeFactoryImpl*, ¿tú qué opinas?

Evite asignaciones mentales, es decir, generalmente se usan las variables i, j, y k para los contadores de bucles, esto es correcto si el ámbito es muy pequeño, pero es mucho mejor utilizar palabras que el programador pueda asociar, no hay peor motivo para usar el nombre ‘c’, que el de que ‘a’ y ‘b’ ya estuvieran asignados. Los programadores inteligentes les gusta presumir de su capacidad de memoria, pero los profesionales saben que la claridad es lo que importa.

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

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

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

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

CAPÍTULO 3 - FUNCIONES

En este capítulo veremos cómo crear funciones, el autor nos muestra un ejemplo en código de una función grande y con código duplicado y nos invita a intentar comprenderla, es tan grande que por mucho esfuerzo que hagamos no conseguimos comprenderla, luego nos muestra la misma función, pero, extrayendo trozos de código en métodos y renombrandolos, ahora sí se puede entender.

Y aquí comenzamos, la primera regla de la creación de funciones es que su tamaño debe ser reducido.En la década de los 80, en las pantallas cabían unas 24 líneas y 80 caracteres, y hoy en día pueden caber 100 líneas y 150 caracteres, lo correcto es que las funciones tengan aproximadamente 20 líneas, es más, las funciones deberían tener 3 o 4 líneas y contar una historia que sea sencilla de entender, esto implica que los bloques if, else, while y similares deberían tener 1 línea que fuera la invocación a otra función. Esto conlleva a que el nivel de sangrado de las funciones sea de 1 o de 2, con lo que resulta extremadamente fácil de leer.

Otra máxima a seguir en las funciones, es que solo deben hacer una cosa, existe una frase que dice: Las funciones solo deben hacer una cosa, deben hacerlo bien y deben ser lo único que hagan. Lo complicado es saber que cosa debe hacer, una forma sencilla de saber que una función hace una sola cosa es que esa función solo realize cosas situadas un nivel por debajo del nombre de la función, si la función se llama *RenderizarPaginaYTestearPagina*, la función solo debería renderizar la página y llamar a otra función que se encargue de testear la página, otra forma es la de que una función que solo hace una cosa no se puede dividir en secciones.

El objetivo es que el código se pueda leer de arriba hacia abajo, es lo que se denomina la regla descendente. Que en cada función que vamos bajando se baje en 1 el nivel de abstracción, así solo quedaría un solo nivel de abstracción en cada función, es algo complicado de entender, pero es la clave para escribir pequeñas funciones.

Las sentencias switch, generalmente son grandes y hacen varias cosas, incumpliendo el principio de responsabilidad única y el principio de abierto/cerrado en el que caso de que haya que añadir un caso más por cada tipo nuevo de empleado que añadamos por ejemplo.
La solución a esto es utilizar el patrón abstract factory y ocultar la función switch.

¿Y qué pasa con los argumentos de las funciones? El número de argumentos ideal para una función es cero, luego uno y después dos. Siempre que sea posible hay que evitar la presencia de tres argumentos. La diferencia entre un método con argumentos y otro que no, es que cada vez que veamos dicho método tendremos que interpretar el argumento que se encuentra en otro nivel de abstracción, con 1 es sencillo pero con 2 y con 3 la cosa se vuelve compleja. Existen los denominados argumentos de indicador, que es pasar un valor booleano para que la función haga una u otra cosa en función de este valor, es totalmente desaconsejable, es preferible dividir esa función en 2.
Existe otro concepto llamado objeto de argumento, y es cuando pasamos dos o más argumentos en un solo argumento mediante un objeto, supongamos que tenemos un

método al que le pasamos el valor x e y de un punto más el radio, pues los argumentos x e y podrían combinarse en un objeto de tipo punto y tan solo se pasarían 2 argumentos.
Como hemos dicho anteriormente el uso de los nombres es muy importante, y sobretodo el jugar con los verbos y los nombres, por ejemplo el método *write(name)* al que le pasamos como argumento name nos dice que sea lo que sea name se escribe, es más, quedaría mejor de la siguiente forma: *writeField*(name), que nos indica además que name es un campo.
No es recomendable que las funciones tengan efectos secundarios, imaginemos que tenemos una función llamada checkPassword en la que comprueba la contraseña y además inicia sesión, este inicio de sesión borraría la sesión anterior, por tanto cada vez que alguien invocará el método pensando que solo va a comprobar la contraseña también borraría la sesión actual, lo ideal sería que la función se llamará *checkPasswordAndInitializeSession*, pero entonces ya no haría solo una cosa, ¿lo ve?

Las funciones deben hacer algo, o devolver algo, pero no ambas cosas, sería confuso. Imagínese que tenemos el código *if(set("username"))*, ¿qué significa? es confuso, podría referirse a si el seteo de la variable ha funcionado o a si se ha seteado previamente, podríamos cambiar el nombre de la función, pero sería también algo confuso, mejor sería separar la función en 2, una que compruebe si existe la variable y otra que setee la variable en tal caso.

También es preferible utilizar excepciones en lugar de códigos de error, esto nos permite separar el código y darle mayor claridad, dado que comprobar un código de error, nos podría complicar excesivamente el código, es preferible siempre un bloque try/catch. En el libro hay un ejemplo en código muy bueno. Pero es ideal extraer el código de dichos bloques en funciones separadas, así aumentamos la legibilidad y mantenimiento del código.

Escribir funciones de esta forma no ocurre de repente, primero se escriben las ideas como un borrador, quizás con nombres no adecuados y funciones largas, pero poco a poco refactorizando se va llegando al objetivo final, como un buen libro.

CAPÍTULO 4 - COMENTARIOS

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

Una de las razones por las que se escriben comentarios es el código incorrecto, creamos una función o una clase, vemos que es un poco confuso y lo comentamos, mejor dedique ese tiempo a crear una función o clase con un buen código, con código limpio. En ocasiones basta con crear una función que exprese exactamente lo mismo que el comentario.

Aun así, existen comentarios buenos y necesarios, aunque recuerde que el mejor comentario es aquel que no hay que escribir. Por ejemplo comentarios legales, que informen sobre derechos de autor, o explicar la intención, como cuando logra una solución pero no cree que sea la mejor y explicar qué es lo que mejor se le ocurrió, o advertir las consecuencias de ejecutar un método que tarda demasiado tiempo, como un test largo, aunque hoy en día usamos `@Ignore` de Junit.
También podemos usar comentarios TODO, que son tareas pendientes, aunque no es recomendable tener el código lleno de estos comentarios. Ahora veamos ejemplos de comentarios incorrectos.

Existen comentarios de tipo balbuceo, en la que el redactor del comentario no se ha expresado bien, un comentario como hemos visto puede resultar útil, pero es necesario que esté bien redactado, si nos obliga a navegar por el código para entender el significado del comentario, es un mal comentario, tenemos también comentarios redundantes, que vienen a decir lo mismo que dice el código, existen comentarios confusos, en los que se dicen cosas que no son coherentes con el código y llevan a confusión al lector, también comentarios obligatorios, como los javadoc antes de las funciones, esto solo ensucia el código y quizás no estén contando la verdad. Muchas veces en lugar de comentar el código podemos extraer los datos en variables para que luego quede más legible y evitar los comentarios de esta forma. Existe algo muy habitual como los marcadores de posición cuando escribimos un comentario del tipo // Acciones ////////////////////////////// para definir que a partir de ese punto existirán funciones que serán acciones, esto resulta al final muy molesto, quizás sean útiles pero si se usan de forma esporádica. También vemos comentarios de cierres de llave, al final de un if o un while para indicar su cierre, esto podría tener sentido en estructuras muy grandes, pero quizás debería refactorizar su código y reducir su tamaño, como apunte personal hoy en dia los IDE modernos nos permiten ver el principio y el final de un bloque de manera visual. Hay algo horroroso como son los comentarios de código, no comente código y lo deje ahí, puesto que nadie se atreverá a eliminarlo por pensar que es muy importante y se irá acumulando. No es recomendable incluir demasiada información en los comentarios, evite los comentarios de 10 o 20 líneas, o más. En definitiva, utilice los comentarios de forma correcta y coherente.

CAPÍTULO 5 - FORMATO

Cuando los usuarios ven nuestro código queremos que se asombren de lo bien hecho que esta, que parezca un trabajo hecho por profesionales, y no solo un masa amorfa y desordenada de código.

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

Empecemos por el tamaño vertical, por el tamaño de los archivos, que está muy relacionado con el tamaño de las clases, en el libro se ve una imagen con varios proyectos y sus medias de líneas de código por archivo, por lo que vemos un tamaño adecuado podría ser entre 200 y 500 líneas como máximo, aunque no debe ser una regla, y recuerde que entre más pequeño mejor se entiende.

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

Es importante añadir una línea en blanco entre diferentes métodos o acciones dentro de una función, mejorará la legibilidad del código, puesto que a simple vista se pueden ver las diferentes partes de la clase o de la función, en el libro hay un ejemplo que vale la pena observar.

¿Alguna vez ha tenido que recorrer una clase saltando de un lado a otro buscando variables y métodos? Esto es molesto para el desarrollador, los aspectos comunes deben estar juntos verticalmente en el código, esto no aplica cuando un método está en otro archivo, por eso es importante tener una razón de peso para separar acciones comunes en archivos diferentes. Siguiendo con lo mismo, las variables deben declararse próximas a su uso, como las funciones deben ser pequeñas, las variables de ámbito local deben declararse al inicio de la función. Pero, ¿y qué pasa con las variables de instancia? Lo normal en Java es declararlas al inicio de la clase, y en C++ al final, no importa el lugar mientras se declaren todas en un mismo punto conocido, no vale declarar 4 arriba, 2 en medio, y 3 al final.

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

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

Es muy útil además rodear con espacios los operadores de asignación o las operaciones aritméticas para una mejor legibilidad como: `int a = 2;` en vez de `int a=2;`

Es importante también respetar los niveles de sangrado del código, muchas veces nos podemos ver tentados a replegarlo en una sola línea cuando un constructor tiene una sola línea por ejemplo. Esto reduce legibilidad, es mejor respetar el sangrado.

Aunque cada programador tiene su estilo de formato, cuando se trabaja en equipo hay que adaptarse a ese estilo y formato, para que el código tenga un formato coherente y no parezca escrito por desarrolladores enfrentados, para terminar el capítulo se muestra en un ejemplo de código las reglas que utiliza el autor del libro.

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

Veamos ahora la diferencia entre objetos y estructuras de datos, los primeros ocultan sus datos con abstracciones y tienen operaciones con dichos datos, mientras que los segundos, muestran sus datos y carecen de funciones. A continuación en el libro se muestran una serie de ejemplos en código para entender el concepto. (Pág. 123)

Ahora veamos la ley de Demeter, que nos dice que un módulo no debe conocer los entresijos de los objetos que manipula, de este modo un método *f* de una clase *C* sólo debe invocar los métodos de *C,* un objeto creado por *f,* un objeto pasado como argumento a *f,* un objeto en una variable de instancia de *C.* Es decir no hable con desconocidos, sólo con amigos. El siguiente ejemplo de código incumple esta ley:

```
final String word = ctx.method1().method2().method3();
```

Al anterior código también se le denomina choque de trenes, y es algo que puede denotar descuido en el código y que se debe evitar.

Por ejemplo una mejor implementación sería:

```
final String word = ctx.method1();
final Object1 obj1 = word.method2();
final Object2 obj2 = obj1.method3();
```

Pero entonces nuestra función ya conoce demasiado, ¿pero incumple la ley de Demeter?, todo depende de si son objetos o estructuras de datos. Los objetos la incumplirían, mientras que las estructuras de datos no.

Muchas veces existe confusión y lleva a crear híbridos, estructuras mitad objetos y mitad estructuras de datos, evite a toda costa esto.

En conclusión, los objetos muestran comportamiento y ocultan datos, esto facilita la inclusión de nuevos objetos sin necesidad de cambiar los comportamientos existentes, pero también dificulta la inclusión de nuevos comportamientos en objetos existentes. Las estructuras de datos muestran datos y carecen de comportamiento significativo, esto facilita

la inclusión de nuevos comportamientos, pero dificulta la inclusión de nuevas estructuras de datos en funciones existentes. En un sistema, a veces necesitaremos uno u otro, los buenos programadores de software entienden estos problemas sin prejuicios y eligen el enfoque más apropiado.

CAPÍTULO 7 - PROCESAR ERRORES

El control de errores es algo que todos tenemos que hacer al programar, las entradas pueden ser incorrectas y los dispositivos pueden fallar. No obstante la conexión con el código limpio debe ser evidente. El control de errores es importante, pero si oscurece la lógica, es incorrecto. En este capítulo, vamos a ver diversas técnicas para crear código limpio y robusto, que procese los errores de manera limpia y con estilo.

Es recomendable usar excepciones en lugar de códigos devueltos, en el pasado los lenguajes carecían de excepciones y los errores se comprobaban mediante códigos de error, el problema de esto es que había que comprobar el código mediante condicionales y oscurecía la lógica, con las excepciones todo es más sencillo y no hace falta hacer comprobaciones. A su vez, también es aconsejable usar la sentencia *try-catch* debido a que de esta forma sabemos que el código que se ejecute en *try* puede ser parado en cualquier momento y finalizado de forma coherente.

Durante años se ha pensado que las excepciones comprobadas eran necesarias para crear software robusto, pero se ha visto que no es así, además incumplen uno de los principios SOLID de abierto/cerrado, si un método tiene en su firma la comprobación de una excepción todos los métodos que lo invoquen deberán también tener su comprobación y esto resultaría en una cascada de cambios dependiendo de lo grande que sea el software.

Es importante además ofrecer contexto en las excepciones, que nos indique en que caso y a que se debe dicha excepción, para ello es aconsejable redactar mensajes de error informativos y pasarlos junto a las excepciones.

Otro consejo que nos da el libro y que se explica mejor en un ejemplo de código en la página 140, es que si 2 excepciones tratan el mismo mensaje de error quizás es mejor envolverlas en una sola clase, de esta forma eliminamos elementos duplicados y nos aseguramos de que devuelven un tipo de excepción común. Es importante también no devolver NULL, si siente la tentación de hacerlo retorne mejor un objeto especial o una excepción, puede devolver una lista vacía por ejemplo, cuánto código hay con miles de comprobaciones de NULL, así minimizará la presencia de NullPointerException y su código será más limpio.

Peor aún es pasar NULL a un método, si una función espera 2 objetos y uno de ellos se pasa como NULL, pues se generará una excepción obviamente. Podriamos crear una nueva excepción para controlar esta o realizar comprobaciones de *assert* en el método, pero esto no solucionaría el problema, lo mejor es diseñar código intentando que no se pueda pasar NULL y que esto indica un problema.

En conclusión el código limpio es legible, pero también debe ser robusto.

CAPÍTULO 8 - LÍMITES

Lo normal es que no todo el software de nuestros sistemas sea nuestro, usaremos paquetes de terceros, o componentes de otros equipos de nuestra empresa, en este tema veremos cómo definir los límites de nuestro software.

Muchas veces al utilizar código de terceros nos gustaría que tuviera un funcionalidad concreta para nuestro sistema o lenguaje, pero el proveedor prefiere que sea todo global para atraer al mayor número de usuarios posible. Veamos un ejemplo con *Map*, imaginemos que queremos pasar un mapa y no queremos que se borren datos ni se inserten nuevos datos, pero la interfaz de *Map* nos ofrece métodos para ello, entonces ¿que sería lo ideal? Pues encapsular la interfaz en una clase propia para usar nosotros solo con lo que necesitamos, por ejemplo imagine que almacenamos sensores en nuestro mapa. Quedaría así:

```
public class Sensors {
        private Map sensors = new HashMap();

        public Sensor getById(String id) {
                return (Sensor) sensors.get(id);
        }
}
```

Como ven, si pasamos una instancia de esta clase no se podrán ni eliminar ni añadir objetos, solo tendremos la funcionalidad que necesitamos.

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

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

CAPÍTULO 9 - PRUEBAS DE UNIDAD

Nuestra profesión ha evolucionado mucho en los últimos años en los que se ha empezado a hablar del desarrollo guiado por pruebas, antiguamente las pruebas se veían como un código desechable, hace tiempo hice un programa temporizador, e hice un test manual para ver si funcionaba y una vez comprobado deseche ese código, hoy en día hubiera probado hasta el más mínimo detalle, para asegurarme de que cualquiera pudiese utilizar el código, TDD y Agile han hecho mucho por esto, pero aún queda bastante camino.

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

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

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

Hace unos años estuve dirigiendo un equipo que había decidido que sus pruebas no tenían porque ser código limpio, que fueran rápidas y directas, el problemas fue cuando cambiaron el código de producción y llegó un momento en que era imposible actualizar las pruebas para que siguieran pasando, terminaron por eliminarlas y al final se quedaron sin red de seguridad para realizar cambios, la moraleja es que la calidad de las pruebas es tan importante como la calidad del código de producción. Las pruebas son las que permiten que su código sea flexible, mantenible y reutilizable.

¿Que hace que una prueba sea limpia? La legibilidad, esta es más importante en las pruebas que en el código de producción, sea claro, simple, y debe decir mucho con el menor número de expresiones posible. A continuación en el libro se expone un caso de pruebas no limpias (Pág. 161)

Cada prueba se divide en tres partes, generar, operar y comprobar. La primera crea los datos de prueba, la segunda opera en dichos datos y la tercera comprueba que la operación devuelva los resultados esperados.

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

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

Además de lo mencionado, las pruebas limpias siguen otras cinco reglas, cuyas iniciales forman las siglas FIRST en inglés:

Rapidez: Las pruebas deben ser rápidas y ejecutarse de forma rápida. Si son lentas terminará por no ejecutarlas.

Independencia: Las pruebas no deben depender entre ellas, una prueba no debe establecer condiciones para la siguiente.

Repetición: Las pruebas deben poder repetirse en cualquier entorno.

Validación automática: Las pruebas deben tener un resultado booleano: o aciertan o fallan.

Puntualidad: Las pruebas deben crearse en el momento preciso: antes del código de producción.

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

CAPÍTULO 10 - CLASES

Hasta el momento nos hemos fijado en muchas cosas, pero no tendremos código limpio hasta que nos fijemos en los niveles superiores, las clases.

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

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

La primera regla de las clases es que deben ser de tamaño reducido, en las funciones nos fijamos en el número de líneas, en las clases la medida serán las responsabilidades.
En el libro vemos una imagen con 70 métodos, aunque la redujeramos a 5 métodos seguirán siendo demasiados, esto es porque la clase tiene demasiadas responsabilidades.

El nombre de una clase nos ayudará a saber si tiene demasiadas responsabilidades, si es ambiguo, o no es conciso, es que la clase es demasiado grande, otra forma es describiendo la clase sin utilizar las palabras *si, o, y* o *pero,* por ejemplo, la clase X nos permite hacer Z cosas **y** monitorizar H, ese *y* nos indica que la clase tiene demasiadas responsabilidades.

Otra regla a tener en cuenta es el principio de responsabilidad única, que nos dice que una clase o un módulo solo debe tener un único motivo para cambiar, si una clase tiene varios motivos para cambiar, es que tiene demasiadas dependencias, una manera de solucionarlo es extraer esos métodos en clases pequeñas que se encarguen únicamente de eso, puede pensar que tener muchas clases pequeñas puede dificultar la comprensión del código, pero ayudará a una mejor estructuración y a no conocer aspectos innecesarios del programa. Un software debe contener varias clases pequeñas, en lugar de pocas clases de gran tamaño.

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

Algo a tener en cuenta en las clases es el principio abierto / cerrado, que nos dice que una clase debe estar abierta a su ampliación pero cerrada a su modificación. En un sistema ideal, incorporamos nuevas funciones ampliándolo, no modificando el código existente, en la página 183 hay un ejemplo en código muy bueno.

Otra manera de protegernos frente al cambio es la de creación de interfaces, de esta manera también cumpliriamos el principio de inversión de dependencias, que dice que las clases deben depender de abstracciones y no de detalles concretos.

CAPÍTULO 11 - SISTEMAS

Las ciudades son demasiado grandes para ser administradas por una sola persona, para ello tenemos varios equipos que se encargan del transporte, alcantarillado, alumbrado, etc.
En el software los equipos también suelen organizarse de esta forma, pero los sistemas no suelen contar con la misma separación de aspectos y niveles de abstracción, en este capítulo veremos cómo mantener la limpieza en el sistema.

No es lo mismo la construcción que el uso que se le va a dar al sistema, son procesos totalmente diferentes, cuando construyen un hotel tenemos grúas y obreros, cuando este acabado no habrá grúas y los trabajadores serán diferentes. Los sistemas de software deben separar el proceso de inicio, de la lógica de ejecución que toma el testigo tras el inicio.

El proceso de inicio es un aspecto que toda aplicación debe abordar. La separación de aspectos es una de las técnicas de diseño más antiguas e importantes de nuestra profesión.

Una forma de separar la construcción del uso consiste en trasladar todos los aspectos de la construcción a *main* o módulos involucrados por *main*. La función principal crea los objetos y se los pasa a la aplicación para que los utilice. En ocasiones, la aplicación tendrá que ser responsable de la creación de algún objeto, en este caso podemos usar el patrón de factoría para que la aplicación controle cuando crear el objeto, pero manteniendo los detalles de dicha construcción separados del código de la aplicación.

Un potente mecanismo para separar la construcción del uso el la Inyección de Dependencias. En este contexto un objeto no debe ser responsable de instanciar dependencias, sino que debe delegar ese mecanismo en otro autorizado.

En el software no podemos conseguir sistemas perfectos a la primera, debemos implementar hoy, y refactorizar y ampliar mañana. No podemos crear un sistema para 100.000 peticiones si sólo tenemos 100 usuarios, suponiendo que creceremos, debido a que ese gasto es inasumible, de esta forma debemos evolucionar nuestro sistema. Los sistemas de software son únicos si los comparamos con los sistemas físicos. Sus arquitecturas pueden crecer incrementalmente, si mantenemos la correcta separación de los aspectos.
A continuación hay un ejemplo en el libro, en la página 194, que es recomendable ver.

Hablemos de los proxies de Java, estos son útiles en casos sencillos, como envolver invocaciones de métodos en objetos o clases concretas. Sin embargo, los que ofrece JDK solo funcionan con interfaces. Para aplicarlos a clases debe usar una biblioteca de manipulación de código, como CGLIB, ASM, o Javassist. A continuación hay un ejemplo de código en el que se define una interfaz, que se envuelve en el proxy y un POJO, que implementa la lógica. Pero los proxies dificultan la creación de código limpio, además no ofrecen un mecanismo para especificar puntos de ejecución globales del sistema, imprescindibles para una verdadera solución orientada a aspectos.

La separación a través de enfoques similares a aspectos no se puede menospreciar. Si puede crear la lógica de dominios de su aplicación mediante POJO, sin conexión con los aspectos arquitectónicos a nivel del código, entonces se podrá probar realmente la arquitectura. De esta forma se podrá evolucionar y no habrá que crear un buen diseño por adelantado (BDUF). Aunque el software se rige por una física propia, es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz. Para recapitular:

Una arquitectura de sistema óptima se compone de dominios de aspectos modularizados cada uno implementado con POJO. Los distintos dominios se integran mediante aspectos o herramientas similares mínimamente invasivas. Al igual que en el código, en esta arquitectura se pueden realizar pruebas.

La modularidad y separación de aspectos permite la descentralización de la administración y la toma de decisiones. En un sistema suficientemente amplio, no debe existir una sola persona que tome todas las decisiones.

Otro aspecto a tener en cuenta, es que debe usar estándares solo cuando añadan valor demostrable. Muchos equipos usaron la arquitectura EJB2 por ser un estándar, y se olvidaron de implementar el valor para sus clientes. También debe usar lenguajes específicos del dominio. Los lenguajes específicos del dominio permiten expresar como POJO todos los niveles de abstracción y todos los dominios de la aplicación, desde directivas de nivel superior a los detalles más mínimos.

En conclusión, los sistemas también deben ser limpios, en todos los niveles de abstracción, los objetivos deben ser claros. Independientemente de que diseñe sistemas o módulos individuales, no olvide usar los elementos más sencillos que funcionan.

CAPÍTULO 12 - EMERGENCIA

Imagine tener cuatro sencillas reglas para crear diseños de calidad, estas son las cuatro reglas de Kent Beck para crear un software bien diseñado.
Según Kent Beck, un diseño es sencillo si cumple estas cuatro reglas:

1. Ejecuta todas las pruebas.
2. No contiene duplicados.
3. Expresa la intención del programador.
4. Minimiza el número de clases y métodos.

Ahora vamos a describir estas reglas, en orden de importancia.

Ejecutar todas las pruebas:

Un diseño debe generar un sistema que actúe de la forma prevista. Un sistema puede tener un buen diseño, pero si no existe una forma sencilla de probarlo, el esfuerzo sobre el papel es cuestionable, y un sistema que no se puede verificar no debe implementarse.

Crear sistemas testables hace que diseñemos clases de tamaño reducido y un solo cometido. Cuantas más pruebas diseñemos más nos acercaremos a elementos más fáciles de probar, del mismo modo ocurre con la inyección de dependencias. Por tanto, la creación de pruebas conduce a obtener mejores diseños.

Refactorizar:

Una vez tenemos las pruebas, debemos mantener limpio el código. La presencia de pruebas hace que perdamos el miedo a refactorizar y que resulte el código dañado. En esta fase podemos aumentar la cohesión, reducir las conexiones, modularizar aspectos del sistema. Aquí también aplicamos las tres últimas reglas del diseño correcto: eliminar duplicados, garantizar la capacidad de expresión y minimizar el número de clases y métodos.

Los duplicados son los mayores enemigos de un sistema bien diseñado. Suponen un esfuerzo adicional, riesgos añadidos y una complejidad a mayores innecesaria. Por ejemplo, en una colección podríamos tener 2 métodos, *size()* y *isEmpty()*, size podría tener un contador y isEmpty un booleano, pero en vez de tener implementaciones distintas podríamos vincularlos:

```
boolean isEmpty() {
      return 0 == size()
}
```

Debemos eliminar duplicados aunque sean unas pocas líneas de código, al extraer podemos detectar incumplimientos del principio de responsabilidad única.

En cuanto a la expresividad, es fácil crear código que entendamos, pero los encargados de mantener el código no lo comprenderán de la misma forma. El principal coste de un proyecto de software es su mantenimiento a largo plazo, para minimizar los costes es fundamental que comprendamos el funcionamiento del sistema. Por tanto el código debe expresar con claridad la intención de su autor. Puede expresarse si reduce el tamaño de funciones y clases, si mejora los nombres, usar nomenclatura estándar, patrones de diseño.
Las pruebas bien escritas también son expresivas, uno de los principales objetivos de una prueba es servir de documentación mediante ejemplos.

Incluso conceptos tan básicos como la eliminación de código duplicado, la expresividad y el principio de responsabilidad única pueden exagerarse. Por ello, la última regla también sugiere minimizar la cantidad de funciones y clases.

Una gran cantidad de clases y métodos suele indicar un dogmatismo sin sentido.
Nuestro objetivo es reducir el tamaño general del sistema, pero recuerde que esta regla es la de menor prioridad de las cuatro. Por ello aunque sea importante reducir la cantidad de clases y funciones, es más importante contar con pruebas, eliminar duplicados y expresarse correctamente.

En conclusión, estas prácticas no reemplazan la experiencia, pero la práctica del diseño correcto anima y permite a los programadores adoptar principios y patrones que en caso contrario tardarían años en aprender.

CAPÍTULO 13 - CONCURRENCIA

La creación de programas concurrentes limpios es complicada, muy complicada. En este capítulo analizaremos la necesidad de la programación concurrente y sus dificultades.

Mitos e imprecisiones:

La concurrencia siempre mejora el rendimiento:
En ocasiones lo hace, pero solo cuando se puede compartir tiempo entre varios procesos o procesadores.

El diseño no cambia al crear programas concurrentes:
De hecho, el diseño de un algoritmo concurrente puede ser muy distinto al de un sistema de un solo proceso.

No es importante entender los problemas de concurrencia al trabajar con un contenedor Web o EJB: En realidad, debe saber lo que hace su contenedor y protegerlo de problemas de actualizaciones concurrentes y bloqueo.

Otros aspectos relacionados con la concurrencia:

- Genera cierta sobrecarga
- Es compleja, incluso para problemas sencillos
- Los errores no se suelen repetir, de modo que se ignoran.
- Suele acarrear un cambio fundamental de la estrategia de diseño.

¿Qué hace que la programación concurrente sea tan complicada?

Imagine una clase en la que tenemos un entero y un método que devuelve ese valor, lo inicializamos a 43.

Se pueden dar varios casos:
El primer proceso lo lee, y obtiene 44, el segundo, 43 y el valor final es 44.
El primer proceso lo lee y obtiene 43, el segundo 44, y el valor final es 44.
El primer proceso lo lee y obtiene 43, el segundo 43, y el valor final es 43.

Como vemos, el tercer resultado es sorprendente, y esto se debe a que se pueden adoptar varias rutas posibles en una línea de código y algunas generan resultados incorrectos. Existen aproximadamente unas 12.870 rutas diferentes para el ejemplo que acabamos de mencionar, si fuera long en vez de int ascendería a 2.704.156. Muchas generan resultados inválidos, pero algunas no lo hacen.

Principio de defensa de la concurrencia

A continuación mencionamos una serie de principios y técnicas para proteger a sus sistemas de los problemas del código concurrente.

El principio de responsabilidad única, estables que un método, clase o componente debe tener un motivo para cambiar. El diseño de concurrencia, es lo suficientemente complejo, como para ser un motivo de cambio con derecho propio, nuestra recomendación es que separe el código de concurrencia del resto del código.

Es importante limitar el ámbito de datos, ya que dos procesos que modifican el mismo campo pueden interferir entre ellos. Una solución consiste en usar la palabra clave *synchronized* para proteger una sección del código, aunque conviene limitar el número de estas secciones. Recomendación, encapsule los datos y limite el acceso a los datos compartidos.

Una forma de evitar datos compartidos es no compartirlos. En algunos casos se pueden copiar objetos y procesarlos como solo lectura, o copiar, recopilar los resultados y combinarlos en un resultado en mismo proceso. Si existe una forma sencilla de evitar los objetos compartidos, el código resultante tendrá menos problemas.

Los procesos deben ser independientes, intente dividir los datos en subconjuntos independendientes que se puedan procesar en procesos independientes, posiblemente en distintos procesadores.

En Java, Doug Lea desarrolló varias colecciones compatibles con procesos que pasaron a formar parte del JDK en el paquete *java.util.concurrent.* De hecho la implementación *ConcurrentHashMap*, tiene mejor rendimiento que *HashMap* en la mayoría de los casos. Como recomendación, revise las clases de las que disponga. En el caso de Java, debe familiarizarse con *concurrent, concurrent.atomic* y *concurrent.locks*

Es importante conocer los distintos modelos de ejecución, entre ellos tenemos recursos vinculados, exclusión mutua, inanición, bloqueo, bloqueo activo. Una vez conozcamos todos estos modelos, podemos describir los distintos modelos empleados en la programación concurrente.

Productor-Consumidor:
Uno o varios procesos productores crean trabajo y lo añaden a un búfer o a una cola. Uno o varios procesos consumidores adquieren dicho trabajo de la cola y lo completan. La cola sería un recurso vinculado, la coordinación entre productores y consumidores a través de la cola hace que unos emitan señales a otros, cuando uno u otro a completado su trabajo.. Ambos esperan notificaciones para poder continuar.

Lectores-Escritores:
Cuando un recurso compartido actúa básicamente como fuente de información para lectores pero ocasionalmente se actualiza por parte de escritores, la producción es un problema. La coordinación de los lectores para que no lean algo que un escritor está

actualizando y viceversa es complicada, los escritores tienden a bloquear lectores durante tiempo prolongado, lo que genera problemas.
El desafío consiste en equilibrar las necesidades de ambos para satisfacer un funcionamiento correcto.

La cena de los filósofos:
Imagine varios filósofos, sentados en una mesa y espaguetis, mientras no tienen hambre piensan, y para comer necesitan 2 tenedores, si el de la derecha o izquierda ya tiene un tenedor tendrá que esperar, ahora cambie los filósofos por procesos y los tenedores por recursos y tendrá un problema habitual en muchas aplicaciones en las que los procesos compiten por recursos, a menos que se diseñen correctamente, estos sistemas sufren problemas de bloqueo.

La mayoría de problemas de concurrencia son una variante de estos 3 que se han mencionado. Como recomendación, aprenda estos algoritmos básicos y comprenda sus soluciones.

No deben existir dependencias entre métodos sincronizados, pueden generar sutiles errores en el código, si hay más de un método sincronizado en la misma clase compartida puede que su sistema sea incorrecto.

En ocasiones tendrá que usar más de un método en un objeto compartido. En ese caso hay tres formas:

Bloqueo basado en clientes:
El cliente debe bloquear al servidor antes de invocar el primer método y asegurar de que el alcance incluye la invocación al último método.

Bloqueo basado en servidores:
Debe crear un método en el servidor que bloquee el servidor, invoque todos sus métodos y después anule el bloqueo.

Servidor adaptado:
Cree un intermediario que realice el bloque.

Es recomendable reducir el tamaño de las secciones sincronizadas, la palabra clave *synchronized* presenta un bloqueo, los bloqueos son costosos ya que generan retrasos y añaden sobrecarga, por ello reduzca al máximo el tamaño de dichas secciones.

Crear código de cierre correcto es complicado, crear un sistema y que se ejecute indefinidamente es distinto a crear algo que funcione de forma temporal y después se cierre. Pueden existir en estos casos bloqueos, con procesos que esperan una señal para continuar que nunca se produce.

Imagine un sistema que crea procesos y finaliza cuando todos han finalizado, si uno de los subprocesos no finaliza, el principal nunca finalizará.

Ahora imagine un sistema similar, que se le indica que finalice y 2 procesos que funcionan como productor/consumidor, y el producto se cierra de pronto. El consumidor puede quedarse esperando indefinidamente por un mensaje que jamás llegará y quedarse bloqueado y no recibir la señal del principal. Recomendación, planifique con antelación el proceso de cierre y pruébelo hasta que funcione.

A la hora de probar código con procesos, cree pruebas que puedan detectar problemas y ejecútelas periódicamente, con distintas configuraciones de programación y del sistema, y cargas.
Hay muchos factores a tener en cuenta, entre los más importantes:

Considerar los fallos como posibles problemas de los procesos.
El código con procesos hace que fallen elementos que no deberían fallar. Los problemas del código con procesos pueden mostrar sus fallos una vez cada mil o un millón de ejecuciones. Los intentos por repetir el fallo suelen fallar, lo que suele provocar que los programadores lo consideren como un caso aislado. Recomendación, no ignore los fallos del sistema como algo aislado.

Conseguir que primero funciones el código sin procesos.
Los procesos ejecutan código fuera de sus entornos, no intente identificar fallos de procesos y que no sean de procesos al mismo tiempo. Asegúrese de que su código funciona fuera de los procesos.

El código con procesos se debe poder conectar a otros elementos.
El código con procesos debe poder conectar a otros elementos y ejecutar en distintas configuraciones.

El código con procesos debe ser modificable.
La obtención del equilibrio adecuado de procesos suele requerir operaciones de ensayo y error. Permita que se puedan modificar los distintos procesos y también durante la ejecución del sistema.

Ejecutar con más procesos que procesadores.
Cuando el sistema cambia de tarea se producen reacciones, para ello realice la ejecución con más procesos que procesadores o núcleos, así comprobará si el código carece de sección crítica o se producen bloqueos.

Ejecutar en diferentes plataformas.
En 2007 diseñamos un curso de programación concurrente para OS X, la clase la hicimos en WIndows XP con máquina virtual, en todos los casos de prueba el código era incorrecto, cada sistema operativo tiene una política de procesos diferentes que afecta a la ejecución del sistema. Ejecute el código con procesos en todas las plataformas de destino con frecuencia y en las fases iniciales.

Diseñar el código para probar y forzar fallos.

Es habitual que los fallos del código concurrente se oculten. Para ello es conveniente intentar forzar estos fallos, haciendo que el código se ejecute de mil formas diferentes, para ello puede usar métodos como *wait(), sleep(), yield()* o *priority()*. Hay dos opciones de instrumentación de código:

Manual
Puede añadir invocaciones de los métodos anteriormente mencionados manualmente a su código.

Automática
Puede usar herramientas como la estructura orientada a aspectos, CGLIB o ASM para instrumentar su código mediante programación.

FIN.

Hasta aquí el resumen del libro, si te ha gustado el resumen el libro te encantará, en mi opinión son las bases de todo buen desarrollador, además de que a partir del capítulo 14 empiezan los casos prácticos, que seguro te gustarán mucho más.