**Hágase el código.**

El código nunca desaparecerá, ya que representa los detalles de los requisitos. En algún nivel, dichos detalles no se pueden ignorar ni abstraer; deben especificarse, y para especificar requisitos de forma que un equipo pueda ejecutarlos se necesita la programación. Dicha especificación es el código. El que piense que el código va a desaparecer es como el matemático que espera que un día las matemáticas no sean formales. Esperan descubrir una forma de crear maquinas que hagan lo que queramos en lugar de lo que digamos. Esas máquinas tendrían que entendernos de tal forma que puedan traducir necesidades ambiguas en programas perfectamente ejecutados que satisfagan dichas necesidades a la perfección. Esto nunca sucederá. Ni siquiera los humanos, con toda su intuición y creatividad, han sido capaces de crear sistemas satisfactorios a partir de las sensaciones de sus clientes. En realidad, si la disciplina de la especificación de requisitos nos ha enseñado algo es que los requisitos bien especificados son tan formales como el código y que pueden actuar como pruebas ejecutables de dicho código. Recuerde que el código es básicamente el lenguaje en que expresamos los requisitos en ultima instancia, Podemos crear lenguajes que se asemejan a dichos requisitos. Podemos crear herramientas que nos permitan analizar y combinar dichos requisitos en estructuras formales, pero nunca eliminaremos la precisión necesaria; por ello, siempre habrá código.

**Código incorrecto.**

Una de las premisas mas robustas, admitidas e importantes de nuestro sector es la relevancia el código correcto. Una popular empresa que, a finales de la década de 1980, creo una magnifica aplicación. Pero los ciclos de publicación empezaron a distanciarse. No se corrigieron los errores entre una versión y la siguiente. Crecieron los tiempos de carga y aumentaron los fallos. Habían comercializado el producto antes de tiempo con graves fallos en el código. Al añadir nuevas funciones, el código empeoro hasta que ya no pudieron controlarlo. El código incorrecto fue el motivo del fin de la empresa.

Sin duda el código incorrecto le ha supuesto un obstáculo. Entonces, ¿Por qué lo escribió?

¿Tenia prisa? ¿plazos de entrega? Seguramente. Puede que pensara que no tenia tiempo para hace run buen trabajo; que su jefe se enfadaría si necesitaba tiempo para limpiar su código. O puede que estuviera cansado de trabajar en ese programa y quisiera acabar cuanto antes. Todos hemos visto el lio en el que estábamos y hemos optado por dejarlo para otro día. Todos hemos sentido el alivio de ver como un programa incorrecto funcionaba y hemos decidido que un mal programa que funciona es mejor que nada. Todos hemos dicho que lo solucionaríamos después. Evidentemente, por aquel entonces, no conocíamos la ley de Leblanc: Después es igual a nunca.

**El coste total de un desastre.**

Si eres programador, probablemente hayas sufrido los desastres cometidos por otros en el código, en mayor o menos manera. El grado de sufrimiento puede ser significativo. En un equipo que avanza rápidamente en el inicio de un proyecto puede acabar a paso de tortuga. Para ampliar o modificar el sistema es necesario comprender todos los detalles, efectos y consecuencias, para ese modo poder añadir nuevos detalles, efectos y consecuencias. Con el tiempo, el desastre aumenta de tal modo que no se puede remediar. Al aumentar este desastre, la productividad del equipo disminuye y acaba por desaparecer. Al reducirse la productividad, el directos hace lo único que puede: ampliar la plantilla del proyecto con la esperanza de aumentar la productividad. Pero esa nueva plantilla no conoce el diseño del sistema. Por tanto, todos se encuentran sometidos a una gran presión para aumentar la productividad. Por ello, cometen más errores, aumenta el desastre y la productividad se acerva a cero cada vez más.

**El gran cambio de diseño.**

En un caso extremo, el equipo informa al director que no pueden seguir trabajando con ese código y exigen un cambio de diseño. La dirección no quiere invertir en eso, pero no puede ignorar el bajo nivel de productividad. Termina cediendo a las exigencias de los programadores y autoriza el cambio de diseño. Forma un nuevo equipo, y se empieza de cero con la esperanza de un código de buena calidad, pero solo los mejores serán elegidos para el nuevo equipo. Los demás deben continuar con el mantenimiento del sistema actual.

Ahora existe una competencia entre los dos equipos. El nuevo debe crear un sistema que haga lo que el antiguo no puede. Además, deben asumir los cambios que continuamente se aplican al sistema antiguo. La dirección no sustituirá el sistema antiguo hasta que el nuevo sea capaz de hacer todo lo que hace el antiguo. Esta competición puede durar mucho tiempo.

**Actitud**

¿Alguna vez ha tenido que superar un desastre tan grave que ha tardado semanas en lo que normalmente hubiera tardado horas? ¿Ha visto un cambio que debería haberse realizado en una línea, aplicado en cientos de módulos distintos? Son síntomas demasiado habituales.

¿Qué sucede en el código? ¿Por qué el código de calidad se transforma tan rápidamente en código incorrecto? Hay muchas explicaciones. Los requisitos cambian de forma que comprometen el diseño original, los plazos de entrega son demasiado exigentes para hacer las cosas bien. Culpamos a los directores incompetentes, a los usuarios intolerantes y a comerciales sin sentido. Pero la culpa, es nuestra. No somos profesionales.

Los directores y los comerciales nos exigen la información que necesitan para realizar sus promesas y compromisos, e incluso cuando no recurren a nosotros, no debemos tener miedo de decirle lo que pensamos. Los usuarios acuden a nosotros para validar la forma de encajar los requisitos en el sistema. Los directores de proyectos acuden a nosotros para determinar los objetivos. Somo cómplices en la programación del proyecto y compartimos gran parte de la responsabilidad de los fallos, en especial si tiene que ver con el código incorrecto.

No es profesional que los programadores cedan a la voluntad de los jefes que no entienden los riesgos de un posible desastre.

**El enigma**

Los programadores se enfrentan a un enigma de valores básicos. Saben que un desastre ralentiza su trabajo, y aun así todos sientes la presión de cometer errores para poder cumplir los plazos de entrega. No se toma el tiempo necesario para avanzar. Los verdaderos profesionales saben que la segunda parte del enigma no es cierta. No se cumple un plazo de entrega cometiendo un error. De hecho, el error nos ralentiza de forma inmediata y hace que no lleguemos al plazo de entrega. La única forma de cumplirlo, la única forma de avanzar, es intentar que el código siempre sea limpio.

**¿El arte del código limpio?**

Imagine que cree que el código incorrecto es un obstáculo significativo. Imagine que acepta que la única forma de avanzar es mantener el código limpio. Entonces se preguntará como crear código limpio. No tiene sentido intentar crearlo si no sabe lo que es. La mala noticia es que crear código limpio es como pintar un cuadro. Muchos sabemos si un cuadro se ha pintado bien o no, pero poder reconocer la calidad de una obra no significa que sepamos pintar. Por ello, reconocer código limpio no significa que sepamos como crearlo.

Para ello se requiere el uso disciplinado de miles de técnicas aplicadas mediante un detallado sentido de la “corrección”. Este sentido del código es la clave. Algunos nacen con este sentido, otros han de luchar para conseguirlo. No solo permite distinguir entre código correcto e incorrecto, sino que también muestra la estrategia para aplicar nuestra disciplina y transformar código incorrecto en código correcto. Un programador sin este sentido puede reconocer el desastre cometido en un módulo, pero no saber como solucionarlo. Un programador con este sentido vera las posibles opciones y elegirá la variante optime para definir una secuencia de cambios. En definitiva, un programador que cree código limpio es un artista que puede transformar un lienzo en blanco en un sistema de código elegante.

**Concepto de código limpio**

Existen tantas definiciones como programadores.

Bjarne Stroustrup, inventor de C++ y autor de *The C++ Programming Language.*

*Me gusta que mi código sea elegante y eficaz. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser mínimas para facilitar el mantenimiento, el procesamiento de errores completo y sujeto a una estrategia articulada, y el rendimiento debe ser optimo para que los usuarios no tiendan a estropear el código con optimizaciones sin sentido. El código limpio hace bien una cosa.*

El usa la palabra “elegante”, que significa “dotado de gracia, nobleza y sencillez”. El piensa que el código limpio es un placer a la hora de leerlo.

El código incorrecto tiende a aumentar el desastre. Cuando otros cambian código incorrecto, tienden a empeorarlo.

También menciona que el procesamiento de errores debe ser completo, lo que se relaciona con la disciplina de prestar atención a los detalles. El procesamiento de errores abreviado es una forma de ignorar los detalles. Otras son las fugas de memoria, las condiciones de carrera o el uso incoherente de los nombres. En definitiva, el código limpio muestra gran atención al detalle.

Grady Booch, autor de *Object Oriented Alanysis and Design with Applications*

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

Como una buena novela, el código limpio debe mostrar de forma clara el suspense del problema que hay que resolver. Debe llevar ese suspense hasta un punto álgido para después demostrar al lector que los problemas y el suspense se han solucionado de forma evidente. La frase “nítida abstracción” de Grady es un oxímoron fascinante. Nítido es casi un sinónimo de concreto, con un potente mensaje. El código debe ser especifico y no especulativo. Solo debe incluir lo necesario. Nuestros lectores deben percibir que hemos tomado decisiones.

“Big” Dave Thomas, fundador de OTI, el padrino de la estrategia Eclipse

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

Big Dave comparte el deseo de Grady de la legibilidad, pero con una importante variante. Dave afirma que el código limpio facilita las labores de mejora de otros.

Dave vincula la limpieza a las pruebas. Hace 10 años esto hubiera provocado cierta controversia. Pero la disciplina del Desarrollo controlado por pruebas ha tenido un gran impacto en nuestro sector y se ha convertido en uno de sus pilares. Dave tiene razón. El código, sin pruebas, no es limpio. Independientemente de su elegancia, legibilidad y accesibilidad, si no tiene pruebas, no será limpio. Dave usa dos veces la palabra mínimo. Valora el código reducido.

También afirma que el código debe ser culto, una referencia indirecta a la programación de Knuth y que en definitiva indica que el código debe redactarse de una forma legible para los humanos.

Michael Feathers, autor de *Working Effectively with Legacy Code.*

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

Una palabra: dar importancia. Es el verdadero tema de este libro.

Michael ha acertado de pleno. El código limpio es aquel al que se le ha dado importancia. Alguien ha dedicado su tiempo para que sea sencillo y ha prestado atención a los detalles. Se ha preocupado.

Ron Jeffries, autor de *Extreme Programming Installed y Extreme Programming Adventures in C#*

*En los últimos años, comencé y prácticamente terminé con las reglas de código simple de Beck. En orden de prioridad, el código simple:*

* *Ejecuta todas las pruebas.*
* *No contiene duplicados.*
* *No expresa todos los conceptos de diseño del sistema*
* *Minimiza el número de entidades como clases, métodos, funciones y similares.*

*De todos ellos, me quedo con la duplicación. Cuando algo se repite una y otra vez, es una señal de que tenemos una idea que no acabamos de representar correctamente en el código. Intento determinar cual es y, después, expresar esa idea con mayor claridad. Para mí, la expresividad debe incluir nombres con sentido y estoy dispuesto a cambiar los nombres de las cosas varias veces, Con las modernas herramientas de creación de código como Eclipse, el cambio de nombres es muy sencillo, por lo que no me supone problema alguno.*

*La expresividad va más allá de los nombres. También me fijo si un objeto o un método hacen mas de una cosa. Si se trata de un objeto, probablemente tenga que dividirse en dos o más. Si se trata de un método, siempre recurro a la refactorización de extracción de métodos para generar un método que exprese con mayor claridad su cometido y varios métodos secundarios que expliquen como lo hace.*

*La duplicación y la expresividad son dos factores que permiten mejorar considerablemente código que no sea limpio. Sin embargo, existe otra cosa que también hago conscientemente, aunque sea más difícil de explicar.*

*Tras años en este trabajo, creo que todos los programas están formados de elementos muy similares. Un ejemplo es la búsqueda de elementos en una colección. Independientemente de que sea una base de datos de registros de empleados o un mapa de claves y valores, o una matriz de elementos, por lo general tenemos que buscar un elemento concreto de esa colección. Cuando esto sucede, suelo incluir esa implementación concreta en un método o una clase mas abstractos. De ese modo disfruto de una serie de interesantes ventajas.*

*Puedo implementar la funcionalidad con algo sencillo, como un mapa hash, por ejemplo, pero como ahora todas las referencias a la búsqueda se ocultan en mi pequeña abstracción, puedo modificar la implementación siempre que desee. Puedo avanzar rápidamente al tiempo que conservo la posibilidad de realizar cambios posteriores.*

*Además, la abstracción de la colección suele captar mi atención en lo que realmente sucede e impide que implemente comportamientos de colecciones arbitrarias si lo que realmente necesito es una forma sencilla de localizar un elemento.*

*Reducir los duplicados, maximizar la expresividad y diseñar sencillas abstracciones en las fases iniciales. Para mí, es lo que hace que el código sea limpio.*

Ward Cunningham, inventor de Wiki, Fit, y uno de los inventores de la programación eXtreme. Uno de los impulsores de los patrones de diseño. Una de las mentes tras Smalltalk y la programación orientada a objetos. El padrino de todos a los que les importa el código.

*Sabemos que estamos trabajando con código limpio cuando cada rutina que leemos resulta ser lo que esperábamos. Se puede denominar código atractivo cuando el código hace que parezca que el lenguaje se ha creado para el problema en cuestión.*

Este tipo de afirmaciones son características de Ward. Es tan razonable y evidente que apenas parece profundo.

Ward espera que al leer código limpio no le sorprenda. De hecho, ni siquiera tendrá que esforzarse. Lo leerá y será prácticamente lo que esperaba. Sera evidente, sencillo y atractivo. Cada modulo prepara el camino del siguiente. Cada uno indica como se escribirá el siguiente. Los programas limpios están tan bien escritos que ni siquiera se dará cuenta. El diseñador consigue simplificarlo todo enormemente, como sucede con todos diseños excepcionales.

Afirma que el código atractivo hace que el lenguaje parezca creado para el problema. Por tanto, somos responsables de que el lenguaje parezca sencillo. No es el lenguaje el que hace que los programas parezcan sencillos, sino el programador que consigue que el lenguaje lo parezca.

**Escuelas de pensamiento**

Los especialistas de las artes marciales no se ponen de acuerdo sobre cual es la mejor de todas, ni siquiera sobre cual es la mejor técnica de un arte marcial. Es habitual que los maestros de las artes marciales creen sus propias escuelas de pensamiento y los alumnos aprendan de ello. Los alumnos de estas diciplinas se sumergen en las enseñanzas del fundador. Se dedican a aprender lo que su maestro les enseña y suelen excluir enseñanzas de otros maestros. Después, cuando han mejorado su arte, pueden convertirse en alumnos de otro maestro diferente para ampliar sus conocimientos y su experiencia. Algunos seguirán mejorando sus habilidades, descubriendo nuevas técnicas y fundando sus propias escuelas. Ninguna de estas escuelas tiene la razón absoluta, pero dentro de cada una actuamos como si las enseñanzas y las técnicas fueran correctas. Después de todo, existe una forma correcta de practicar Hakkoryu Jiu Jitsu o Jeet Kune Do, pero esta corrección dentro de una escuela determinada no aula las enseñanzas de otras diferente.

Imagine que este libro es una descripción de la Escuela de mentores del código limpio. Las técnicas y enseñanzas impartidas son la forma en la que practicamos nuestro arte. Podemos afirmar que, si sigue nuestras enseñanzas, disfrutará de lo que hemos disfrutado nosotros, y aprenderá a crear código limpio y profesional. Pero no cometa el error de pensar que somos los únicos que tenemos razón. Existen otras escuelas y otros maestros tan profesionales como nosotros, y su labor es aprender de ellos también.

**Somos autores**

El campo @author de un Javadoc indica quienes somos. Somos autores. Y los autores tiene lectores. De hecho, los autores son responsables de comunicarse correctamente con sus lectores.

La próxima vez que escriba una línea de código, recuerde que es un autor y que escribe para que sus lectores juzguen su esfuerzo.

**La regla del Boy Scout**

No basta con escribir codigo correctamente. El codigo debe limpiarse con el tiempo. Todos hemos visto que el codigo se corrompe con el tiempo, de modo que debemos adoptar un papel activo para evitarlo.

Los Boy Scout norteamericanos tiene una sencilla regla que podemos aplicar a nuestra profesión:

Dejar el campamento más limpio de lo que se ha encontrado.

Si todos entregamos el código mas limpio de lo hemos recibido, no se corromperá. No hace falta que la limpieza sea masiva. Cambie el nombre de una variable, divida una función demasiado extensa, elimine elementos duplicados, simplifique una instrucción if compuesta.

**Conclusión**

Los libros sobre arte no lo prometen que se convertirá en artista. Solamente pueden mostrarle herramientas, técnicas y procesos de pensamiento que ottos artistas hayan utilizado. Del mismo modo, este libro no puede prometer que se convierta en un buen programador, que tenga sentido del código. Solo puede mostrarle los procesos de pensamiento de buenos programadores y los trucos, técnicas y herramientas que emplean.

Al igual que un libro sobre arte, este libro esta repleto de detalles. Encontrará mucho código. Vera código correcto y código incorrecto. Vera código incorrecto transformado en código correcto. Vera listas heurística, disciplinas y técnicas. Vera un ejemplo tras otro. Y después de todo, será responsabilidad suya.

¿Recuerda el chiste sobre el violinista que se pierde camino a un concierto? Se cruza con un anciano y le pregunta como llegar al Teatro Real. El anciano mira al violinista y al violín que lleva bajo el brazo y le responde: “Practique joven, practique”.