Los Principios del Programador

Los Principios del Programador es una traducción hecha por Juan A. Romero del proyecto original *Principled Programming* cuyo autor es *Daniel Read*. La página del proyecto original es http://www.developerdotstar.com.

1. Principio #1: El Principio del Carácter Personal

Lo sepamos o no, nos guste o no, nuestro carácter está reflejado en cada línea de código que escribimos, en cada informe que diseñamos, en cada interfaz de usuario que construimos, en cada diagrama que hacemos. Cuando otra persona mira nuestro código --o lo que es tan importante, la salida que produce nuestro código-- esa persona, conscientemente o no, se hace un juicio de nosotros. Piensa en el código que has escrito hasta ahora... ¿cuál sería ese juicio? ¿Estarías orgulloso, te pondrías en pie y dirías que ese código lo has escrito tu? ¿o solo admitirías tímidamente que es tuyo, y lanzarías excusas de por qué no es lo bueno que debería ser?. Si tu código es malo, alguien que lo lea probablemente asuma que no es el único código que has hecho mal, que no es lo único que haces mal. La buena noticia es esta: tenemos control absoluto sobre la calidad de nuestro código.

La cuestión no es si uno es capaz de escribir el mejor código posible, sino si se preocupará por intentarlo. Si un programador carece de ciertos conocimientos o de cierta experiencia, pero hace lo posible por escribir el código de forma clara, entendible, bien comentada, de forma que muestre que al menos ha invertido tiempo en aprender algunos fundamentos básicos, entonces habrá actuado como debía con diligencia-- y eso será obvio para un observador que lea con atención. No es razonable que alguien juzgue mal a un programador porque le falte algo de experiencia o porque desconozca aún algunas técnicas. Sin embargo, es absolutamente razonable encontrar una conexión entre la calidad global del código de un programador y la calidad del carácter de dicho programador.

El Principio del Carácter Personal establece: *Escribe tu código de forma que refleje, y saque a relucir, solo lo mejor de tu carácter personal.*

2. Principio #2: El Principio de la Estética

Un aspecto de la programación que con frecuencia descuidamos es la estética. La estética trata sobre la belleza y la elegancia, y de el valor de estas cualidades. Mucha gente, sin embargo, cree que la estética es solo importante cuando se habla de arte y literatura. Pocas personas se dan cuenta de la importancia de la belleza y la elegancia en las cosas cotidianas,

y muy pocos programadores se dan cuanta de la importancia de éstas cuando se escribe código. La estética es especialmente importante en el desarrollo de software, un terreno en el que siempre estamos tratando con niveles de abstracción. Los aspectos estéticos de nuestras abstracciones están directamente relacionados con su entendibilidad y, por lo tanto, con su utilidad.

Un programador debe esforzarse en conseguir la belleza, sin importar la herramienta o el lenguaje de programación que esté utilizando. La belleza puede conseguirse a muchos niveles, desde el alto nivel de la elegancia en el diseño del sistema hasta el más bajo nivel de la apariencia visual del código en la pantalla. Ser ordenado y claro cuenta. El mejor código no solo funciona de forma correcta y eficiente, y está bien formado desde el punto de vista del compilador; el mejor código es también agradable de ver por el ojo humano-- y por lo tanto más fácil de absorber y de comprender para el cerebro humano.

Steve McConnell escribe en su libro *Code Complete*, "El disfrute visual e intelectual de un código bien formateado es un placer que pocos no-programadores pueden apreciar. Pero los programadores que se sienten orgullosos de su trabajo experimentan una gran satisfacción artística puliendo la estructura visual de su código." (Página 399)

El Principio de la Estética establece: *Esfuérzate por conseguir la belleza y la elegancia en cada aspecto de tu trabajo.*

3. Principio #3: El Principio de la Claridad

La claridad en el código es un estado que debemos buscar activamente. Uno de los mayores delitos que como desarrolladores podemos cometer es olvidar que nuestro código tiene una vida más allá de los pocos momentos que nos lleva escribirlo. Las probabilidad de que alguien, posiblemente nosotros mismos, maneje nuestro código en el futuro son muy altas. Incluso aunque escribamos un código que funciona perfectamente y nunca causa problemas al usuario, estaremos perjudicando a otros compañeros desarrolladores (sin mencionar a nuestra empresa) si no escribimos nuestro código de la forma más clara posible.

Hay una diferencia entre claro y correcto, y muchas veces se confunden. La corrección es siempre el principal interés del desarrollador, como debe ser. La corrección lleva a que la sintaxis del código sea correcta a los ojos del compilador, que el diseño de la interfaz cubra las necesidades del usuario, y que los algoritmos que se implementan cumplan con sus requerimientos. Pero si no se dedica una atención igual a la claridad, la comprensibilidad y la mantenibilidad del código sufrirán mucho. Para que nuestro código sea lo más claro posible, debemos deliberadamente usar técnicas como la utilización de identificadores descriptivos, la modularidad, la indentación (el sangrado), los espacios en blanco, la cohesión del código, el acoplamiento débil del código, propiciar la fácil realización de las pruebas y la documentación, y comentar adecuadamente.

La falta de claridad en nuestro código causa problemas innecesariamente, y también situaciones profesionalmente embarazosas. A nuestros colegas desarrolladores que tienen que tratar con nuestro código en años (o incluso décadas) sucesivos. Y a nuestra empresa, con la que no jugamos limpio devaluándola; incluso en algunos casos nuestro código puede convertirse en un problema de responsabilidad legal para nuestra empresa. Dejar a nuestros colegas en esa situación es como mínimo descortés, pero dejar a nuestra empresa en esa situación es mucho peor: nosotros llegamos al acuerdo de producir código de calidad por un precio. La empresa ha cumplido pagándonos ese precio pero, ¿en qué situación sale ella de este acuerdo con nosotros?

El Principio de la Claridad establece: *Dale el mismo valor a la claridad que a la corrección. Utiliza activamente técnicas que mejoran la claridad del código. La corrección vendrá casi por sí sola.*

4. Principio #4: El Principio de la Distribución

Este principio se refiere a la distribución visual del código. El Principio de la Distribución es un corolario a los dos principios anteriores: El Principio de la Estética y El Principio de la Claridad. El Principio de la Estética nos dice que, además del disfrute intelectual que supone la lectura de código bello y elegante, la propia belleza y la elegancia juegan un papel crucial para conseguir dicho buen código. Por otro lado, El Principio de la Claridad nos dice que hagamos nuestro código lo más claro posible a un lector humano, y que la claridad va de la mano con la corrección. El Principio de la Distribución pone estos dos principios duales en práctica.

Es difícil discutir la importancia de la distribución visual del código sin referirnos al principio fundamental de *Steve McConnell* (*Fundamental Theorem of Formating*) que establece: "Una buena distribución visual muestra la estructura lógica de un programa." (página 403, *Code Complete*) Esto significa que la distribución visual no solo sirve para hacer el código visualmente más atractivo, sino que también actúa a nivel del subconsciente haciendo el código más entendible al lector. El objetivo es reducir la cantidad de trabajo que un lector necesita para entender el código. La apariencia visual debe ser nuestra primera herramienta para comunicarnos con claridad con un lector humano que esté leyendo nuestro código.

Hay varias técnicas que aseguran que la distribución visual del código ayude a entender su estructura lógica. Estas técnicas deben formar parte de los conocimientos fundamentales de todo programador: uso adecuado de los espacios en blanco (y líneas en blanco), indentación, agrupamiento de líneas relacionadas, uso de paréntesis extra para que las expresiones lógicas y las fórmulas matemáticas sean más claras, márgenes para alinear el código relacionado de distintas líneas, emplazamiento adecuado de delimitadores de bloque, etc. Como *Steve*

McConnell sugiere en su teorema, la idea es utilizar la distribución visual de nuestro código para comunicar a nivel de subconsciente con el lector. Recuerda que cuando estamos trabajando en el terreno del desarrollo de software, estamos siempre trabajando con abstracciones, y una abstracción es más útil cuanto mas entendible es.

El Principio de la Distribución establece: *Usa la distribución visual de tu código para comunicar la estructura de tu código a un lector humano.*

5. Principio #5: El Principio de lo Explícito

Seguir el Principio de lo Explícito nos ahorrará, a nosotros y a nuestros sucesores, innumerables problemas. El Principio de lo Explícito es un corolario de El Principio de la Claridad. Pero el Principio de la Claridad nos dice que hagamos nuestro código de forma clara y entendible para el lector humano. Y el Principio de lo Explícito también se aplica a esa misma entendibilidad, y a lo que es más importante, a hacer nuestro código más tolerante a cambios.

Veamos un ejemplo simple: muchos lenguajes de programación ofrecen el concepto de "fichero". Normalmente el fichero no lo proporciona el lenguaje de programación directamente, sino que se utiliza a través de una librería. Cuando escribimos el código necesario para abrir el fichero, normalmente existen propiedades para determinar el estado del cursor del fichero. Cuando nuestro código llama a la función que abre el fichero, la función probablemente tenga un estado del cursor por defecto, lo cual significa que no tenemos que especificar explícitamente el estado del cursor-- simplemente podemos aceptar el estado por defecto.

Si un desarrollador acepta ese estado por defecto, entonces el desarrollador ha introducido una *suposición implícita* en su código, lo cual, para empezar, reduce la claridad del mismo. Y lo que es mucho peor, ha creado una oportunidad gigantesca para que se den futuros errores. El estado del cursor afecta directamente el comportamiento del fichero. Nuestro código obviamente depende de ese comportamiento. ¿Qué ocurriría con nuestro código seis meses después cuando se actualice la librería de manejo de ficheros?¿Qué ocurre si la nueva versión cambia el estado por defecto del cursor del fichero?¿Sabes lo que va a ocurrir con el código que asumía el estado por defecto del fichero? Simplemente va a fallar por completo. O peor, va a cambiar su comportamiento en algo leve: alguna condición de un *IF* va a ser *Falsa* en vez de *Verdadera*, y el bloque *ELSE* se va a ejecutar en vez del bloque *IF* que debería ejecutarse; y entonces, meses después, alguien se dará cuenta de que los datos corruptos y los daños se han extendido por todo el sistema.

Por no haber sido explícito, el desarrollador del código del ejemplo aceptó implícitamente el estado del cursor por defecto, haciendo su código menos tolerante a futuros cambios. Esto es inaceptable. Debemos aprender a detectar donde pueden darse este tipo de problemas (el

ejemplo del fichero es uno de muchos, y cada lenguaje de programación y plataforma tiene otros muchos), y debemos usar nuestros conocimientos para que no nos afecten en el futuro. Como profesionales, debemos saber que el cambio es algo constante en nuestra profesión, y debemos hacer lo posible para asegurar que nuestro código soportará fácilmente esos cambios sin producir errores o resultados incorrectos. Incluso cuando nuestro código falla, debemos asegurarnos que lo hace correctamente, es decir, que a pesar de fallar, nos proporciona la mayor cantidad de información específica acerca de dónde y por qué ha fallado.

El truco para que estos problemas no ocurran es ser explícitos. Los problemas merodean por lo implícito-- en las suposiciones indocumentadas y no probadas, en las técnicas crípticas, en las secciones indocumentadas explícitamente, en las cuales nos perdemos como un barco hundiéndose silenciosamente en el vasto y opaco océano.

El Principio de lo Explicito establece: *Intenta siempre favorecer lo explícito sobre lo implícito.*

6. Principio #6: El Principio del Código Auto-Documentado

El código auto-documentado no se consigue accidentalmente.

Como desarrolladores, debemos esforzarnos en el desarrollo de un estilo de codificación sólido, lo cual es la clave del código auto-documentado. Debemos estar constantemente mejorando y perfeccionando nuestro estilo, de forma que cada programa que escribimos sea mejor que el anterior. Un estilo bien desarrollado se consigue incorporando técnicas probadas, como el uso de identificadores informativos y consistentes; la modularización bien cohesionada y acoplada; evitar el uso de técnicas difíciles de comprender; hacer una buena distribución visual del código; dar nombres adecuados a las constantes; probando y documentando las suposiciones; y muchas otras.

Para aprender estas técnicas debemos leer los textos realizados por otros que nos precedieron, y que abrieron camino. Buscar en la bibliografía clásica. Aprender de los maestros. Suscribirse a revistas. Unirse a listas de discusión de Internet. Leer código y código de otros programadores. Cuando veas código que decepciona estas expectativas, analízalo y trata de ver por qué. Cuando veas código que alcanza el alto nivel que buscamos, trata de identificar las técnicas que el desarrollador usó para alcanzar ese gran nivel. Hay un dicho que dice: cualquier tonto puede aprender de sus propios errores; una persona sabia también aprende de los errores ajenos.

¿Y qué ocurre con los comentarios? ¿Los necesitamos aunque nuestro código esté auto-documentado? Auto-documentar el código, como la perfección, es un objetivo difícil de conseguir, probablemente imposible. Sin embargo, eso no debe detenernos en nuestro

empeño. Allí donde esa perfección no se pueda conseguir, debemos suplementar nuestro esfuerzo añadiendo buenos comentarios. El código bien escrito no debe necesitar demasiados comentarios; debe explicarse por si mismo. De todas formas, añadir ciertos comentarios es necesario--solo que deben ser del tipo adecuado (ver El Principio de los Comentarios).

Visto de cerca, el código verdaderamente auto-documentado es un placer de contemplar, y a su observador le queda claro que esa maravilla solo pudo ocurrir mediante el esfuerzo de un ingeniero de software concienzudo y diligente.

El Principio de Código Auto-Documentado establece: *La documentación más fiable para el software es el propio código. En muchos casos, el propio código es la única documentación. Por lo tanto, esfuérzate en hacer que tu código sea auto-documentado, y allí donde no sea posible, añade comentarios.*

7. Principio #7: El Principio de los Comentarios

Los comentarios son armas de doble filo. Usados correctamente, pueden mejorar infinitamente la entendibilidad y el mantenimiento del código. Usados de forma indebida, pueden hacerlo confuso y menos legible. Comentar indebidamente es en el mejor de los casos de poca ayuda, y en el peor de los casos un enorme desastre.

El Principio de los Comentarios tiene tres apartados:

Primero, comenta mediante frases completas. Esta simple técnica incrementa en gran medida la comprensión por parte de un lector tanto de los propios comentarios como del código que comentan. Los fragmentos de frases tienden a ser crípticos. Escribir cada comentario mediante una frase completa también hace que lo entiendan más personas, lo cual es muy importante en el entorno multi-cultural en el que nos encontramos hoy día. Los comentarios de alta calidad escritos mediante frases completas también son una ayuda para instruir a desarrolladores con menos experiencia.

Segundo, usa los comentarios para resumir código. El "comentario resumen" describe un bloque de código con la idea de ahorrar a una persona el tiempo que llevaría leer todo el código que el comentario describe. El comentario resumen aparece normalmente unas líneas por encima de un bloque de código. Un buen resumen no repite de nuevo el código, sino que explica diversas líneas de código en dos o tres frases.

Tercero, intenta siempre comentar "a nivel de la intención". Lo que esto quiere decir es que se debe comentar a nivel del problema, más que a nivel de su solución. El código es la solución al problema. Idealmente, el código debería hablar por si mismo (ver El Principio del Código Auto-Documentado). Una persona que lea el código, si el código es bueno, debería entender fácilmente qué hace el código y cómo lo hace. Sin embargo, la información que no permanece en el código es precisamente *lo que había en la mente del desarrollador que lo*

escribió: su intención. En general, *esto* es lo que necesita estar comentado. ¿Cuál era la *intención* del desarrollador?¿Con qué *intención* se ha usado ese código? ¿Cómo *intenta* este código solucionar el problema en cuestión? ¿Cuál es la idea que hay detrás de ese código? ¿Cómo este código explícitamente se relaciona con las otras partes de código?. Uno de los mayores pecados que un desarrollador puede cometer es abandonar su código sin que su intención quede clara a futuros lectores del código.

El Principio de los Comentarios establece: *Comenta mediante frases completas para resumir y comunicar la intención.*

8. Principio #8: El Principio de las Suposiciones

El Principio de las suposiciones es un corolario al Principio de lo Explícito. Hacer comprobaciones y documentar bien las suposiciones hechas en el código tiene varios beneficios: uno, incrementa su comprensión; dos, hace el código más predecible; tres, hace el código más fácil de mantener; cuatro, reduce la necesidad de comentarios; cinco, hace el código más fiable; seis, hace el código más comunicativo cuando algo va mal; siete, la detección temprana mediante comprobaciones protege los datos de que se corrompan ; ocho, obliga que prestemos atención a las suposiciones y comprobaciones que hace una rutina, y por tanto a su relación con otras rutinas y datos compartidos, lo cual reduce la incidencia de errores.

Las comprobaciones son la piedra angular de la programación defensiva. En realidad en cada pieza de código hacemos suposiciones. Lo que ocurre es que algunas son obvias y no es necesario que se comprueben ni se documenten. Sin embargo, podemos y debemos comprobar (y documentar) las muchas otras suposiciones que son menos evidentes.

El tipo de suposiciones más común que deben ser comprobadas (y documentadas), son los pre-requisitos en los que una rutina se basa. Estas comprobaciones normalmente son una serie de sentencias `If...Then` al principio de la rutina. Si alguna de las comprobaciones falla, entonces el código debe ejecutar una acción que corrija la situación, o bien devolver un mensaje de error explicando que una de las suposiciones falló. (Otra forma de hacer comprobaciones de las suposiciones, a veces pasada por alto, es el uso de asertos, que son expresiones `Verdadero/Falso` que normalmente sólo se compilan en versiones de "depuración" del programa para su prueba.)

El Principio de las Suposiciones establece: *Da los pasos que sean razonables para comprobar, documentar y prestar atención a las suposiciones hechas en cada módulo y rutina.*

9. Principio #9: El Principio de la Interfaz con el Usuario

La formulación de esta regla está tomada prestada de *About Face: The essentials of User Interface Design*, escrito por el gurú del diseño de interfaces de usuario: Alan Cooper. Cooper abunda en esta idea en su posterior libro, *The Inmates are Running the Asylum* : "La mayoría del software se usa en entornos de trabajo, donde las víctimas de los malos interfaces de usuario son los trabajadores. Sus trabajos les obligan a usar el software, ellos no pueden elegir *no* usarlo--deben tolerarlo lo mejor que puedan. Están obligados a aguantarse su frustración y a ignorar la vergüenza que sienten porque el software les hace sentirse estúpidos".(Página 34) Esta es una afirmación que debe hacer que nos paremos a considerar el impacto real, bueno o malo, que nuestro software tiene en la gente.

¿Cómo llega un usuario a sentirse estúpido?. Alan Cooper ha dedicado dos libros enteros a responder a esta pregunta, y otros autores se han enfrentado también a este tema (incluyendo a Donald Norman en su excelente libro, *The Design of Everyday Things.)* Obviamente, aquí no podemos extendernos tanto. Pero de todas formas, aquí va un ejemplo sencillo: un usuario hace *click* en un botón e inmediatamente le sale un mensaje que que dice: "No puede usar esa función en este momento". *¿Entonces por qué estaba el botón habilitado para ser pulsado?* El desarrollador, por no tomarse la molestia tan sencilla de deshabilitar o bien ocultar el botón, ha creado una situación en la que él o ella es como un bromista que señala una falsa mancha en la camisa del pobre usuario y le da en la cara cuando éste mira hacia abajo. Muy divertido.

El mal diseño de la interfaz de usuario es un enorme problema que afecta a toda la industria del hardware y del software. Es triste comprobar como desarrolladores tanto de hardware como de software regularmente diseñan soluciones que terminan haciendo que los usuarios se sientan estúpidos. Es incluso más triste que muchos desarrolladores sean felices en su ignorancia de desconocer el estrés que causan a la gente. Sin embargo, como desarrolladores estamos en la posición de ayudar a resolver este problema.

Como desarrolladores software, nuestra principal responsabilidad en cuanto a la interacción con el usuario descansa en el diseño de la interfaz de usuario. Afrontémoslo: en la mayoría de los casos no hay un diseño de la interfaz de usuario preparado con antelación. La mayoría de las decisiones de diseño de la interfaz de usuario (y esto incluye el diseño de los informes que genera el software) son tomadas *por el desarrollador mientras se construye*. Por lo tanto, en la mayoría de los casos, es *solo responsabilidad del desarrollador* tomar los pasos necesario para no hacer que el usuario se sienta estúpido. El desarrollador es el que sitúa los botones y los campos. El desarrollador, por tanto, tiene casi control total de la experiencia que vive el usuario delante del software.

El Principio de la Interfaz con el Usuario establece: *Nunca hagas que el usuario se sienta estúpido.*

10. Principio #10: El Principio de Volver Atrás

Todos hemos sido culpables alguna vez de esto: "No tengo tiempo de hacer eso ahora. Ya volveré y lo haré luego". Este tipo de decisiones suelen aplicarse a tareas como comentar el código, su adecuada distribución visual, el control de errores, la modularización adecuada, la correcta implementación, etc. Quizás tu seas esa persona, rara, que siempre vuelve más tarde a ese código y hace todo ese trabajo *tedioso*, pero la mayoría de nosotros los mortales nunca lo hacemos.

El momento de hacer todas esas tediosas tareas asociadas a la codificación es *el preciso momento en el que se está escribiendo el código*. La principal razón por la que nadie vuelve más tarde a "limpiar" su código es que una tarea que es tediosa de realizar mientras escribes el código, es monumentalmente más tediosa cuando tienes que volver atrás y hacerla después. ¿Alguien cree de verdad que volver y codificar buenas comprobaciones de errores en cientos de rutinas a posteriori es *menos* tedioso que crear esas rutinas con sus comprobaciones de errores desde el principio? No solo odiarás cada minuto de esa tarea, sino que casi con toda probabilidad introducirás errores que no estaban allí antes.

En el caso de los comentarios, los comentarios que añadas después nunca serán tan buenos como los comentarios que podrías haber escrito justo en el momento en que escribiste el código. ¿Y qué ocurre si tienes que dejar un trabajo antes de tener la oportunidad de volver atrás y hacer todas esas tareas? Habrás dejado esa tediosa y mucho más monumental tarea a otro, lo cual es muy poco profesional.

El Principio de Volver Atrás establece: *El momento de escribir buen código es justamente el preciso momento en el que lo estás escribiendo.*

11. Principio #11: El Principio de El Tiempo y El Dinero de Otros

El Principio de El Tiempo y El Dinero de Otros se aplica a todo el trabajo que realiza un desarrollador: código, informes, interfaces de usuario, modelos, diagramas, pruebas, y documentación. Esta regla no solo se aplica al código, sino a la profesionalidad. Recuerda la regla con la que empezamos: El Principio del Carácter Personal establece: *Escribe tu código de forma que refleje, y saque a relucir, solo lo mejor de tu carácter personal.* El Principio de El Tiempo y El Dinero de Otros es una forma menos cordial de decir lo mismo.

Enorgullécete de tu trabajo, porque tu trabajo eres tú, y te juzgarán por el trabajo que has hecho--y a veces sólo por eso. Incluso si no te preocupa que te juzguen, ¿Te preocupa hacer las cosas bien hechas? ¿Te sientes bien habiendo aceptado dinero por ese código? ¿Te preocupa como tú y tu trabajo se refleja en la profesión de ingeniería del software en general? Escribir código para alguien no es un tan solo un juego divertido, la calidad del trabajo de cada desarrollador se refleja en todos los demás desarrolladores.

El Principio de El Tiempo y El Dinero de Otros establece: *Un verdadero profesional no gasta el tiempo ni el dinero de otras personas produciendo software que no esté razonablemente libre de fallos; que no esté mínimamente probado; que no cumpla con los requisitos establecidos; que esté falsamente adornado con características innecesarias; o que tenga un aspecto deplorable.*