Las 10 meteduras de pata más comunes entre los programadores

Reutilización
Mantenimiento
Usabilidad
Variables
Comparaciones
Seguridad
Complejidad
Optimización
Validación de datos
Excepciones

CONTENIDOS

CUANDO LOS PROGRAMADORES METEMOS LA PATA....

Esto es un recopilatorio de "Gambadas" orientada a dar consejos generales para programadores sobre cómo escribir un mejor código. Para ello revisaremos algunos de **los errores más comunes que todos los programadores cometemos.** Son errores que van más allá de un lenguaje o entorno concreto por lo que cualquiera, independientemente del entorno en el que trabaje, le va a poder sacar partido. Esperamos que te sea útil**.**

Error #1:

Escribir código difícil de leer

Muchas veces, por desgracia, los programadores escribimos más para el compilador que para las personas. Hay muchos ejemplos:

- Esa expresión súper-optimizada que el compilador entenderá sin problemas pero que una persona tendrá dificultades para comprender.
- Esa estructura en la que, por ahorrar espacio (ya sabemos que el GB disco duro va carísimo), omites los paréntesis y llaves, y además escribes en una sola línea varias instrucciones ya que, de todos modos, se van a interpretar bien.
- Divides el código en varias partes, cada una en una clase diferente, entrelazándolos mediante eventos o llamadas encadenadas. Sí, seguro, es *cool* pero, aunque creas lo contrario, la Separación de Responsabilidades no es eso y además lo que haces es liar tu código tremendamente. ¡Cualquiera le sigue la pista!
- Usas nombres de variables tan clarificadores como 's', 'res', 'i' o 'n'. Estupendo, has ahorrado unos bytes, pero ¿no sería mejor que las variables tuvieran un nombre adecuado a la función que cumplen y siguiendo algún tipo de notación?
- No usas comentarios, porque al fin y al cabo el código es muy fácil de entender y siempre puedes ejecutarlo paso a paso. Claro que, ahora acuérdate de por qué hacías precisamente eso y no otra cosa. O explícale a otro programador aquella optimización que hiciste por aquel motivo de rendimiento tan concreto.

En todo esto lo que subyace es la siguiente pregunta: Si alguien tiene que leer tu código ¿lo entenderá? ¿Podrá seguirle la pista enseguida? O lo que es más importante: **¿lo entenderás tú si lo vuelves a leer dentro de tres meses?** El compilador traga con todo y lo entiende todo, pero las personas no.

Aunque muchos no lo crean, lo más importante de un buen código no es lo optimizado que esté (salvo en muy pocas excepciones), **sino lo fácil de mantener que sea en el futuro.** Y esto depende de la persona que lo escribe y de lo fácil que lo haga para otras personas.

Así que escribe **código más amable y fácil de seguir.** Piensa en las personas (incluso en ti mismo) que más adelante tendrán que leerlo y entenderlo para darle soporte, y piensa menos en el lenguaje o el compilador.

¡Ah!, y usa comentarios, pero comentarios que tengan sentido. No tengas miedo de extenderte. ¿Sabes aquel viejo chiste que dice que si se desvelara el código completo del ADN del hombre, el 90% serían comentarios? Pues aplícate el cuento ;-)

Error #2:

Usar variables globales

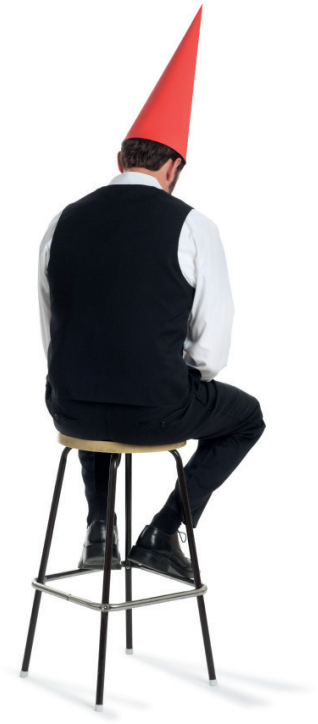
Los programadores que ya tienen unos cuantos años de práctica a sus espaldas recordarán los tiempos de Visual Basic 5.0 o de ASP 3.0 clásico. Estos lenguajes tenían el problema de contar con variables globales, es decir, aquellas que son accesibles desde cualquier parte de tu código.

La ventaja de las variables globales es que te permiten acceder a su contenido en cualquier momento y desde cualquier línea de tu código, por lo que vienen genial para compartir información. La parte mala de las variables globales es exactamente la misma: al poder acceder a ellas desde cualquier parte nunca puedes estar seguro de su valor.

Lo que ocurre es que muchas de las veces que vas a usar una variable global, asumes que ésta con-

tiene un determinado valor o está en determinado estado según el orden esperado de ejecución.

Nunca sabes desde qué parte de tu código o qué evento puede haberlas modificado, así que son un accidente esperando a ocurrir.



Si te crees que eso está ya muy superado y que en lenguajes modernos como C# ya no tienes ese problema, piénsalo de nuevo. ¿Qué pasa con las variables estáticas?, ¿y los objetos de tipo Singleton?, y en las aplicaciones Web ¿qué me dices de las cachés y de las variables de aplicación? Todos estos mecanismos de compartición de información pueden causar los mismos problemas.

Incluso hoy en día, lenguajes tan utilizados como JavaScript, adolecen del mismo problema ya que disponen de variables globales y un programador poco experimentado abusará de su uso, resultando en páginas con errores en el momento en que uno menos se lo espera.

El consejo: procura no compartir información globalmente salvo que sea estrictamente necesario y si lo haces verifica siempre antes de utilizarlo que el almacén de esa información (sea una variable global, una estática o lo que sea) contiene lo que se espera que debe contener y que está inicializado. Parecen inofensivas pero tienen un gran peligro, incluso a la hora de hacer testing de apliaciones. Así que ¡cuidado!.

Error #3:

Escribir rutinas demasiado largas

Por fortuna hace muchos años desde que terminó la época de la programación tipo Top-Down, en la que todo el sistema era un único programa. Por aquel entonces tu programa era una gran rutina que controlaba absolutamente todo. Los famosos listados de código podían ocupar cientos de folios... Hoy en día no tenemos disculpa para crear enormes listados monolíticos con decenas (o incluso cientos) de líneas de código para realizar una sola tarea.

La regla general es que **el código de una rutina no debería ocupar más de un *"scroll"* de pantalla,** es decir, que no debería ser necesario darle a la tecla "Av. Pág" más de una vez para poder leer el código completo. El motivo es que debe ser fácil de leer y entender para cualquiera, lo cual contribuirá a su mantenimiento. Si escribes rutinas largas, ni siquiera tú serás capaz de entenderlas bien al cabo de

unos meses. Otro factor adicional es el poder hacer el testing adecuadamente. Si tenemos un código muy largo y por lo tanto con muchos condicionales, bucles, etc... se hace difícil testearlo bien y llegar a una buena cobertura de código. De hecho existe una medida formal de la complejidad del código: la Complejidad Ciclomática. Aunque su uso formal es complicado y la fórmula compleja, existe una **regla general** muy interesante para calcular la CC de manera aproximada y **saber si tu código es demasiado largo:** cuenta el número de condicionales y bucles que tiene tu rutina y súmale 1. Si tienes más de 9 o 10 ya sabes: refactoriza, extrae métodos, simplifica... Tu código y tú mismo lo agradeceréis en el futuro.

Error #4:

No validar los campos de entrada de datos

Un problema muy común en los programadores es que son incapaces de imaginarse situaciones diferentes a las que ya han considerado en su código. Y lo cierto es que a veces les falta imaginación. Así, ocurren cosas como que un campo que sólo puede aceptar números enteros se valida -en efecto- para comprobar que no tiene decimales, pero no se nos ocurre que los usuarios quizá tengan la brillante idea de introducir números enteros... ¡pero negativos! Resultado: la aplicación rompe con un feo error.

Nunca, jamás, en la vida, te fíes de nada que llegue introducido por un usuario.

Existen infinidad de ejemplos como este y son bastante comunes. Ten por seguro que si existe la posibilidad de meter la pata o hacer algo ilógico, algún usuario lo hará... y nuestra aplicación romperá miserablemente.

Aparte de tratar de prever todos los casos a la hora de validar datos, hay que tener en cuenta la seguridad. La regla de oro de la seguridad informática es: **Nunca, jamás, en la vida, te fíes de nada que llegue introducido por un usuario.** Lo más suave es que pueda romper tu aplicación, pero, lo más probable, es que al fiarte estés facilitando la vulneración de la seguridad, siendo pasto de ataques por inyección de SQL, Cross-Site Scripting, CSRF, etc...

Y hoy en día esto es más importante que nunca ya que rara es la aplicación que trabaja individualmente, sin estar en una red a la que acceden varios usuarios o, en muchos casos, directamente conectada a Internet para que cualquiera pueda intentar *hackearte*.

Así que no te olvides: por elegancia y robustez de la aplicación pero sobre todo por seguridad: **valida absolutamente todo lo que te llegue** desde los usuarios, y **piensa en todos los posibles casos** dependiendo del tipo de datos que vayas a recibir. Incluso piensa en qué ocurriría si te mandan la información usando un tipo de datos diferente al que esperabas inicialmente. Tu aplicación, tus datos y tus usuarios te lo agradecerán.

Error #5:

No utilizar gestión estructurada de excepciones

Tradicionalmente se han utilizado diversas técnicas para notificar que algo ha fallado dentro de un método o función.

Por regla general se le notificaban los posibles errores a otras rutinas de ámbito superior utilizando un valor devuelto, con un significado diferente según el error/excepción. Así, por ejemplo, si la rutina devolvía un 0 es que todo había ido bien, pero si algo fallaba o había cualquier problema se devolvían otros números diferentes, cada uno con un significado. El que haya usado la API tradicional de Windows sabe perfectamente a qué nos referimos.

Otra variante de esto, pero similar en esencia, la podíamos encontrar en lenguajes "viejunos" como Visual Basic clásico -hasta la versión 6.0- que incluso disponía de una constante *(vbobjectError,* ¡qué tiempos aquellos!) específicamente pensada para devolver errores propios. En este caso no se devolvía un valor, sino que se generaba un error que había que capturar y mirar el numerito para ver de qué situación se trataba.

Por fortuna ya hace muchos años que se ha superado todo esto y disponemos de la **gestión estructurada de excepciones,** implementado en los diversos lenguajes a través de las palabras clave **try, catch, finally y throw** o sus variantes.

Lo interesante de este modo de gestionar excepciones, es que **éstas se transmiten a través de la pila** de llamadas hasta que alguien las intercepte, por lo que eliminan de un plumazo multitud de condicionales de comprobación que había que hacer en cada llamada a una función, pudiendo gestionarlos en el nivel que más nos convenga.

OJO, usar gestión estructurada de excepciones está bien, **pero no debes hacerlo de cualquier manera.**

Por ejemplo, algo muy típico que constituye una costumbre muy mala es **capturar siempre todas las excepciones de manera genérica.** ¿Por qué lo haces? Lo bonito de estas técnicas es que nos permiten ser muy específicos con el tipo de excepción que capturamos en el "catch", pudiendo así hilar fino en la gestión o delegarla al nivel de la pila de llamadas que más nos interese.

Otra mala praxis típica es la de **capturar los errores de manera genérica para, a continuación, dejar el "catch" en blanco.** ¡No lo hagas! Si pasa algo que no te esperabas (y siempre pasa algo que no te esperas) no te enterarás. Sí, tu aplicación no romperá, pero si deja de funcionar correctamente no tendrás manera de saber por qué ocurre.

Si eres de los todavía devuelven constantes para marcar que han ocurrido errores, olvídate de esa técnica desfasada y adopta la gestión estructurada :-)

Error #6:

Optimizar antes de tiempo

La expresión más común para este error es "**Optimización prematura**" y fue acuñada por el legendario informático Donald Knuth en su libro clásico de 1974 **"Structured Programming with go to Statements"**. A pesar de haber pasado casi 40 años el principio sigue siendo perfectamente válido.

Esta expresión se refiere a la situación común en la que un programador tiene a un pequeño demonio sentado en su hombro diciéndole: "Seguro que si tocas ese código puedes hacer que vaya mucho más rápido. ¡Hazlo ahora!". Y lo hace :-)

No podemos dejar que, s**in un análisis global de rendimiento,** pequeñas consideraciones sobre la eficiencia o rendimiento de un código afecten a nuestras decisiones de diseño. En concreto Donald decía:

> *"Debemos olvidarnos de las pequeñas eficiencias, digamos el 97% del tiempo: la optimización prematura es la raíz de todo mal."*

A lo que añadía:

> *"De todos modos no debemos pasar por alto las oportunidades de ese otro 3%"*

Lo que quería decir es que, **sin tener unas mediciones precisas y globales** de la aplicación, **no deberíamos optimizar** por el mero hecho de que pensemos que podemos mejorar el rendimiento. Hay optimizaciones triviales (ese 3%) que deberíamos hacer (como no concatenar cadenas de texto en un bucle largo), pero todo lo que no sea así de obvio es mejor dejarlo de lado hasta que realmente lo podamos medir con contexto global.

Debemos olvidarnos de las pequeñas eficiencias, digamos el 97% del tiempo: la optimización prematura es la raíz de todo mal.

¿Por qué? Existen múltiples razones, pero las más importantes son que sin una adecuada instrumentación (un profiler) no podremos **saber realmente el impacto** que esa optimización tiene en el resultado final, y a cambio **complicaremos el código** haciéndolo más difícil de entender y de mantener, podremos **provocar bugs** fácilmente y **perderemos el tiempo.**

Un procedimiento mucho mejor es **crear el código según el diseño inicial** (con las optimizaciones triviales) y más tarde **hacer profiling** de la aplicación completa para ver qué partes realmente necesitan ser optimizadas. Un diseño simple siempre es fácil de optimizar a posteriori, y el análisis de rendimiento nos mostrará lugares inesperados en los que realmente se necesita la optimización y que probablemente no se nos habían ocurrido en una optimización prematura.

Error #7:

No considerar en serio la seguridad

Muchos programadores **se ciñen en exceso a las especificaciones técnicas del proyecto,** pero no van más allá en otras cuestiones que están implícitas y que es imposible que un cliente o alguien no técnico les vaya a detallar. Una de estas cuestiones es **la seguridad.**

Obviamente la seguridad es algo deseable y que se da por hecho, pero lo cierto es que **la mayoría de los sistemas son inseguros.** Esto es especialmente cierto para aquellos que están conectados a Internet o a una red, que en la actualidad son la mayoría.

Muchos programadores tienden a pensar que **la seguridad es más bien cosa de la gente de sistemas:** para eso, esos tíos feos se dedican a colocar cortafuegos, sistemas de detección de intrusos, defensas ante ataques DDOS y demás ingenios para controlar las comunicaciones.

Lo cierto **es que en la mayor parte de los sistemas que se asaltan** usando medidas técnicas (y no ingeniería social), **se utilizan debilidades en la forma en la que está escrito el código.**

Ya hablábamos en el **"Error #4"** de esta serie sobre la importancia de **validar los datos.** En aquella ocasión nos referíamos más bien a la evitar posibles errores por conversiones o tipos de datos inválidos. Pero realmente, esta validación debe hacerse también para evitar problemas de seguridad. Ataques como los de **Inyección de SQL, Cross-Site-Scripting** (XSS), **Cross-Site Request Forgery** (CSRF) son muy comunes y en gran parte se pueden mitigar validando los datos de entrada desde el punto de vista de seguridad.

Otra cuestión que generalmente se hace mal, es **todo lo que tiene que ver con el tratamiento de información confidencial:** desde la gestión de credenciales y claves de acceso, hasta el almacenamiento seguro de datos de manera que sólo puedan ser accedidos por ciertos usuarios. Se suele almacenar más información confidencial de la realmente necesaria, muchas veces se guardan contraseñas en claro en las bases de datos (escapa como de la peste de los sitios que te ofrecen recuperar tu clave por email, y no generar una nueva: ¡mala señal!), en otras ocasiones los algoritmos de cifrado son muy débiles o no se utilizan adecuadamente....

Y eso nos lleva a **la cuestión de la criptografía:** se suele utilizar muy mal. Por ejemplo, no se hace un uso adecuado de los "salt" para cifrado o cálculo de hashes (recomendamos encarecidamente que te leas este artículo de Troy Hunt), nos inventamos nuestros propios algoritmos de cifrado o seguridad (¡la madre de todos los errores!), se utilizan claves de cifrado débiles, no se entiende bien cómo funcionan los sistemas de clave pública....

Existen infinidad de otros puntos a considerar. Podríamos escribir un libro solo sobre este tema (de hecho los hay y muy buenos). La idea con la que debemos quedarnos es que en **cada línea de código que escribas tienes que pensar en la seguridad.** No puede ser un pensamiento a posteriori, cuando ya está todo terminado.

Para hacerlo bien deberíamos tener **formación en seguridad** (para saber por dónde nos pueden venir los

problemas), **planificar la seguridad a alto nivel** junto con la arquitectura de la aplicación, y luego pensar en **cualquier dato que manejemos** y en cómo podría verse comprometido. Es laborioso pero no hay otra manera...

Error #8:

Utilizar comparaciones innecesarias en condicionales

En esta caen hasta los programadores más experimentados.

¿Cuántas veces has escrito condicionales como este?:

```
if (miBooleano == true){
    ....
}
```

en C#, JavaScript, Java, C++...o bien, por ejemplo, en Visual Basic:

```
If miBooleano = True Then
    ....
End If
```

Aunque en realidad no es código erróneo y funcionará perfectamente, utilizar los condicionales de esta manera es una prueba de que, o no se entiende bien cómo funciona un condicional, o de que el código que escribimos es poco elegante.

Dentro de la cláusula de prueba de un condicional debe haber siempre un booleano. Cuando trabajamos con valores que no son booleanos, no nos queda más remedio que hacer algún tipo de comparación, como por ejemplo:

```
if (miValorEntero < 50){
    ....
}
```

de modo que obtenemos un verdadero o falso en función del resultado de la comparación.

Sin embargo, cuando lo que tenemos en nuestra variables es ya un valor verdadero o falso ¿por qué

hacer entonces la comparación? No tiene sentido hacerlo y sobra el comparador, que generalmente ponemos por mimetismo con el caso general. Sería más lógico y elegante simplemente escribir:

```
if (miBooleano){
    ....
}
```

¿Verdad?

Quizá la única excepción a esta regla no escrita de elegancia en el código puede darse en el lenguaje JavaScript. Como se trata de un lenguaje poco tipado se suele utilizar el comparador "===" para asegurar que una variable no sólo tiene un valor determinado, sino que además ambos valores (el de la variable y el del valor con el que se compara) son exactamente del mismo tipo y no se convierte implícitamente durante la comparación:

```
if (miBooleano === true){
    ....
}
```

No siempre es necesario o tiene sentido, pero en algunas ocasiones puede ser útil si debemos asegurar igualdad en valor y en tipo. Esta sería la excepción.

Quizá te parezca que nos hemos puesto excesivamente puntillosos con este asunto, pero realmente en estos pequeños detalles se marcan las diferencias entre los buenos y los grandes programadores ;-)

Error #9:

No pensar como el usuario

A muchos programadores les cuesta horriblemente pensar cómo lo haría uno de sus usuarios. El mayor problema suele ser que, como el programador conoce muy bien cómo está hecho el programa por debajo y cuál es la manera "correcta" de trabajar con él, tiende a adaptar el paradigma de funcionamiento a este conocimiento. Pero eso hace que muchas veces nos olvidemos por completo de cómo piensan los usuarios, que sólo quieren solucionar sus problemas y sacar adelante sus tareas.

Para ver un ejemplo en la práctica consideremos cómo han funcionado históricamente los procesadores de texto. El programador sabe cómo funciona un ordenador y sabe que lo que el usuario escribe en la pantalla se va almacenando en algún lugar de la memoria. Estos datos se perderán si

se cierra el programa sin guardarlos o si se va la luz de repente. Para almacenarlo permanentemente es necesario persistirlo a disco, y por eso los programadores le ponen a los programas un botón de "Guardar". A estas alturas a casi todo el mundo le parece la forma natural de trabajar y a nadie le sorprende. Sin embargo esto no es natural en absoluto.

A un usuario le da igual cómo trabaje un programa por debajo mientras funcione. Si lo pensamos, obligar a un usuario a guardar en disco un texto que está escribiendo es una manera de forzarles a trabajar de manera poco natural. Implica que éstos deben saber que el ordenador almacena temporalmente en memoria la información y que deben guardarla a disco si quieren conservarla. Estamos introduciendo conocimiento del sistema (memoria volátil, sistema de archivos...) en un ámbito que no tiene nada que ver ("¡quiero escribir una maldita carta!"). Es un ejemplo prototípico de la mentalidad técnica de un programador forzando a los usuarios a adaptarse.

Pues como este, podemos encontrar cientos de ejemplos en programas que usamos diariamente si nos paramos a analizarlos. Por ejemplo, en la mayoría de los programas de gestión, la estructura interna de la base de datos (es decir, cómo está almacenada la información) impone formas de trabajar o limitaciones a los usuarios que jamás deberían asomar a la superficie y que, desde luego, si ellos utilizasen lápiz y papel (o sea, no se les impusieran restricciones técnicas de manera artificial) jamás harían de ese modo.

Lo que el usuario ve al usar un programa debe responder a sus expectativas de uso, no a cómo está programado un sistema por debajo. Es el programador el que debe adaptarse a la forma de pensar de los usuarios, no a la inversa. Para que los usuarios estén contentos con tu programa deben sentir que simplemente funciona, no que deben hacer un esfuerzo para adaptarse a ellos.

Los programas más exitosos son siempre los que siguen esa norma tan simple y tan compleja al mismo tiempo. Dos ejemplos:

- **Google Docs:** sus editores no tienen un botón de guardar, y de hecho no existe esa acción como tal. A pesar de ser un sistema basado en web, este hecho es transparente al usuario que lo puede utilizar como cualquier programa de escritorio, con la enorme complejidad que ello conlleva, pueden colaborar en tiempo real como si estuviesen en la misma red, etc...
- **DropBox:** es transparente por completo para el usuario que tiene la sensación de que simplemente funciona. No tiene que saber nada de sistemas de archivo, sistemas operativos, qué información está en local, cuál en la misma red y cuál en remoto, dónde se guardan los datos, si hay conflictos, si se guardan versiones... Es un producto muy complejo que no podría ser más fácil de manejar y que se adapta a lo que los usuarios esperan.

Al final lo que más importa es lo que piensen los usuarios sobre tu aplicación. Aunque la hayas optimizado a tope, uses la última tecnología o tenga la arquitectura mejor diseñada del mundo, una mala interfaz que no piense en los usuarios la echará a perder por completo.

Así que la moraleja sería: **no dejes para el final la interfaz**, y **trata de pensar siempre como un usuario y no como un técnico**. Imagina que no sabes nada sobre cómo funciona tu aplicación ni sobre cómo está organizada, y piensa en cambio en cómo es el esquema mental de tus usuarios. Conseguirás software mucho más exitoso y que la gente quiera usar :-)

Error #10:

Reinventar la rueda... y todo lo contrario

Por mucho que algunos quieran negarlo, esta profesión tiene mucho de ego personal, con una alta dosis de obsesión por el control y un pellizco de dificultades para delegar.

El síndrome del **"No Inventado Aquí"** (más conocido como NIH por sus siglas en inglés), se refiere a desechar la opción de utilizar algún componente externo de software que no haya sido creado por uno mismo o dentro de la empresa. Es lo que se llama también "Reinventar la rueda".

Los motivos para caer en el NIH pueden ser múltiples, y se parecen a una lista de pecados:

- **Orgullo:** El código no me gusta porque no está escrito siguiendo mi estilo, así que no me quiero molestar en comprenderlo.
- **Obsesión por el control:** Si uso algo que no he hecho yo, no hará exactamente lo que yo quiero y como yo lo quiero.
- **Desconfianza:** Tengo la certeza de que no va a ser tan seguro o estable como algo que haga yo mismo o mi equipo.
- **Egoísmo:** aunque al cliente le urge, prefiero hacerlo a mi manera que acelerar la entrega con algo en lo que no confío.
- **Avaricia mal entendida:** si lo construimos nosotros nos ahorraremos el dinero de las licencias. Ya. Pero ¿cuál será el verdadero coste de desarrollar algo remotamente similar?.
- **Duda:** ¿qué pasará en el futuro si dejan de mantenerlo o desaparece?.
- **Narcisismo:** si lo hago yo mismo ganaré muchos puntos ante los jefes por la proeza.
- **Soberbia:** si no resuelvo yo mismo las dificultades técnicas, entonces este trabajo carece de sentido. Yo, y solo yo, debo ocuparme de esto y llevarlo a buen puerto, así que rechazo cualquier ayuda externa.

Lo cierto es que para casi cualquier cosa que necesitemos ya existirá en el mercado algo que se parezca mucho a lo que buscamos, o que será exactamente lo que nos hace falta. Entonces **¿por qué reinventar la rueda y hacerlo nosotros?.**

Existen algunos motivos (aunque pocos) por los que es mejor **ocuparnos nosotros mismos y de verdad ponernos a reinventar la rueda** en lugar de comprar o descargar un componente externo:

- **La licencia** de lo que hay en el mercado no nos permite distribuirlo dentro de un paquete comercial.
- Si buscamos algún componente o biblioteca del que hay **poca variedad**, y hay indicios auténticos de que los pocos p**roveedores existentes no merecen confianza** (vemos código inestable, empresas que en cualquier momento desaparece, proyectos Open Source que hace años que no se actualizan...). En este caso quizá es mejor idea implementar algo nosotros mismos. Si algún componente es Open Source podemos partir de ese código para hacerlo, pero ojo con la licencia y lo que nos permite hacer.
- Lo que necesitamos es, en realidad, **el núcleo de funcionalidad de nuestro producto.** En este caso no hay duda: si estás seguro de que el producto tiene sentido lo lógico es construirlo. Si te limitas a comprarlo, cualquier otro puede venir después a competir con vosotros. Claro que esto daría para un gran debate sobre si lo que importa es el producto o es el marketing y la forma de llegar al mercado... Pero eso quizá sea tema de otro artículo algún día :-)

En resumen: por regla general lo mejor es no tener miedo a utilizar código ajeno y debemos tratar de no reinventar la rueda constantemente. **El trabajo de programador es entregar a tiempo y con calidad soluciones que funcionen,** no demostrar lo buenos que somos retrasando el proyecto y aumentando los riesgos por el mero hecho de querer hacerlo todo uno mismo. Ahora bien, si lo que necesitamos es parte de la funcionalidad esencial del producto, entonces sí que merece la pena invertir en construirlo.