Calidad del Software

Las 10 reglas de oro para escribir código de calidad

Errores y lecciones aprendidas sobre crear buen código, a base de caer de precipicios y levantarse, y de trabajar con los mejores desarrolladores del mundo

Iñaki Ayucar

Sobre el autor

Iñaki Ayucar

Iñaki es ingeniero informático. Es fundador de la empresa de simulación híper-realista de vehículos **Simax Virt**, Director Técnico en **Engineea** y profesor en la **Universidad Pública de Navarra**.

Hasta 2016, fue Director Técnico Asociado en **Electronic Arts** durante el desarrollo de juegos como Battlefield 1, FIFA 17, Star Wars Battlefront o Need For Speed, entre muchos otros. Anteriormente, fue Ingeniero Senior en **Microsoft Game Studios – Rare**, durante el lanzamiento de Xbox One, trabajando en el desarrollo del juego Kinect Sports Rivals.

Microsoft Corp lo premió como Microsoft MVP (*Most Valuable Professional*), en la categoría de DirectX, entre 2009-2013

Puedes seguirlo en Twitter: @iayucar

Foto de la portada: desmoronamiento del puente Queen Isabella, en Cameron County, Texas, en 2001. Dominio Público.

¿Qué vas a encontrar?

Decálogo para escribir código de calidad

Todos sabemos que escribir buen código es muchísimo más que conocer un lenguaje. Del mismo modo que saber hablar inglés no te convierte en Shakespeare, conocer C# o Java no te convierte en programador (y mucho menos en un Ingeniero de Software).

Suelo preguntar a mis alumnos qué criterios utilizarían para decidir si un software está acabado o no. Hay respuestas que aparecen pronto, como:

- "comprobar que resuelve el problema"
- "ver que no gasta recursos innecesarios"
- "asegurar que no tiene bugs"
- ...

Otras respuestas, sin embargo, son mucho más esquivas.

> Que un programa resuelva un problema, que lo haga sin fallos y sin gastar más recursos de lo necesario, es el mínimo exigible. El "aprobado raspado", podría decirse.

Si se pretende **alcanzar cotas de calidad más altas**, el software debe tener muchas otras propiedades, más difíciles de ver a simple vista.

Un software **excelente** debe:

- Ser fácilmente **entendible**
- Ser **extensible** y **reusable**
- Estar **bien estructurado y testado**
- Facilitar el **compartir conocimiento**
- Estar bien **preparado para cambios** futuros que no podemos prever.

¿Cómo se consigue todo eso?

A continuación, te proporciono unas pautas que creo que te pueden ayudar a caminar en la dirección de esta excelencia del software.

Foto de Sara Pflug, Dominio Público

Mi decálogo de consejos para la excelencia en el software

Antes de nada, te ofrecemos una **tabla de resumen** del decálogo, y posteriormente se desarrollan un poco más cada uno de los puntos.

Esta tabla **no está estructurada siguiendo un orden concreto de prioridad**. Es decir, el que un consejo vaya antes que otro no significa que le dé más importancia. Todos son importantes, y cada uno de ellos puede ser más relevante en un contexto y menos en otro. Pero, en general, si los sigues todos deberías tender a conseguir software de calidad, que es lo que se busca.

> Se trata de un **resumen de errores y lecciones** aprendidas a base de caer por el mismo precipicio una y otra vez, o gracias a trabajar con gente de un talento increíble, de la que aprendes cada día. Son reglas que yo, personalmente, considero importantes. Por supuesto, todo es discutible y, por supuesto, existen muchas más. Cada uno tiene las suyas... pero éstas son las mías 😎

Tabla resumen

1	**No duplicar**. Nunca.	Código duplicado == bugs
2	**Comentar**, comentar y comentar. Y luego documentar	No comments == código ilegible
3	Priorizar la **herencia** de clases sobre los interfaces/servicios	Interfaces == código no extensible
4	**Code reviews** antes de subir cambios a servidor, sin excepción.	No reviews == islas de conocimiento
5	No escribir métodos que hagan más de una cosa	Métodos **multipropósito** == código no reusable
6	No admitir código que no se entienda rápidamente	Código **ilegible** == bugs
7	Dedicar tiempo a pensar el **lugar apropiado** para el código	Programadores impetuosos == estructura incorrecta
8	Si te cuesta escoger el nombre de algo, probablemente esté mal diseñado	**Nombres** largos o vagos == código mal estructurado
9	**Reutilizar**, reutilizar y reutilizar	No reutilizar == trabajo extra y bugs
10	No acumular más de un cambio por *commit*	***Commits*** largos == revisiones de código lentas

Regla #1: No Duplicar

Si estás duplicando código, algo estás haciendo mal. Es así de sencillo. No hay ni un solo contexto que justifique duplicar código, decisiones, contenidos o *assets*.

Un ejemplo: si existe un método que calcula el número de elementos de una tabla de tu base de datos, debes reutilizarlo en todos y cada uno de los lugares en los que necesites ese dato. Y si no puedes, probablemente eso te esté indicando que tu solución debería estar diseñada de otra forma, quizás con un paquete o biblioteca más *core*, que contenga ese tipo de utilidades o clases y que pueda ser referenciada por todos los componentes de tu solución. Duplicar cosas es una solución rápida que te ahorra el problema, pero que sin embargo te impide ver todo eso, e introduce *bugs* potenciales (¡a montones!).

Por ejemplo, cuando hay métodos duplicados y algo cambia, hay que acordarse de aplicar el cambio en todos ellos. Y claro, lo típico es que el ingeniero encargado se acuerde de cambiar 10 de los 11 sitios repetidos, lo cual es equivalente a un nuevo *bug*.

O peor aún, es demasiado frecuente encontrarse con cosas que se duplicaron en su día y que han seguido su propia evolución, por lo que ahora ya no son exactamente iguales. Todo parece indicar que el objetivo de ambos métodos es (o era) el mismo, pero la implementación varía en uno u otro sitio. ¿Cuál debe prevalecer ahora?

He visto este tipo de errores en toda clase de entornos: desde videojuegos AAA hasta aplicaciones corporativas. Desde empresas pequeñitas y StartUps, hasta la propia Microsoft. Y sinceramente, evitar las duplicidades probablemente sea el consejo más importante que puedo dar a nadie.

Regla #2: Comentar, comentar y comentar. Y luego documentar

Es también demasiado frecuente encontrarte con aquello de: "si el código está bien escrito, no hacen falta comentarios". Y es cierto, un código bien estructurado, con nombres bien escogidos, que sigue una estructura lógica y con sentido común, requiere muchos menos comentarios. Pero eso no debe llevarnos a pensar que los comentarios no son necesarios.

Comentar es esencial, no solo para hacer nuestro código más legible, sino porque muchas, muchísimas veces, al hacer el esfuerzo de explicar nuestro propio código, nos damos cuenta de que es mejorable.

> Si te cuesta explicar tu propio código, probablemente debas revisarlo y estructurarlo de otra forma.

Los comentarios deben escribirse **pensando en los demás**, no en uno mismo. Se trata de ponérselo fácil a todo el que llegue a tu código y necesite entenderlo rápidamente. El objetivo no es explicar lo evidente, con comentarios del estilo: "este es el constructor de la clase", sino describir (y recordar) **la lógica** que hay detrás de lo que escribes: por qué se tomaron ciertas decisiones y por qué se decidió que el código se comportara de esa forma. Cosas como: ¿por qué se cambia el valor de una variable en un momento determinado? ¿por qué se le da la vuelta a esa lista? ¿por qué ese `for` empieza en `1`, y no en `0`?

Da igual si estás desarrollando el algoritmo más difícil del videojuego más complejo del mundo. Tu código, su estructura básica y su objetivo, deberían ser fáciles de entender para cualquier programador en cuestión de segundos o pocos minutos.

Regla #3: Priorizar la herencia de clases sobre la composición

Si preguntas a cualquier ingeniero de software si domina la Programación Orientada a Objetos, el 100% contestará que sí. Lamentablemente, muchos la aplican sólo parcialmente (o no la aplican en absoluto). A veces por desconocimiento, a veces guiados por prácticas un tanto *misleading*. Especialmente en entornos de desarrollo corporativo que implican bases de datos, donde se prioriza solucionar la persistencia por encima de la propia lógica de negocio.

La persistencia en bases de datos relacionales ha sido un problema típico en el desarrollo de software. De ahí que existan los ORMs y patrones de diseño centrados en resolver este problema, porque es comprensible que en entornos donde hay miles de tipos de entidades distintos, el problema de la persistencia adquiera un gran protagonismo.

Sin embargo, muy a menudo esto no es así. Gestionamos un volumen de tipos de entidades manejable y el grueso de nuestros problemas no está en la propia persistencia, sino en la lógica de negocio, que sí puede llegar a complicarse mucho.

En estos casos, es un error guiarse por los consejos tradicionales que se centran en facilitar la persistencia, a costa de complicar todo lo demás.

> El hecho de tener que persistir tu modelo de dominio en una base de datos relacional, no debería condicionar el diseño de tu solución.

No son pocos los *gurús* que directamente aconsejan olvidarse de la herencia, y centrarse exclusivamente en la *composición/agregación*. Craso error.

Porque lo cierto es que tu negocio crecerá (y si no, malo), y nunca vas a poder predecir completamente qué necesitarás en el futuro. Llegarán nuevos proyectos, nuevos clientes y nuevas necesidades. Y tu modelo de dominio necesitará adaptarse y especializarse, por lo que una de las claves para sobrevivir con éxito a esa fase, es que tu solución sea extensible y reusable. ¿Puede conseguirse una solución extensible y reusable, utilizando únicamente la composición/agregación? Sí, pero si no se hacen las cosas muy muy bien, tu solución terminará incumpliendo gran parte de las normas explicadas en este artículo (sobre todo, la de "no duplicar"). Y en el mejor de los casos, aunque tu

solución sea un prodigio de la composición y logres no caer en esos problemas, seguirá siendo una maraña de interdependencias entre clases que no es nada evidente de seguir (porque no hay aparente relación jerárquica entre ellas) y terminará siendo una pequeña pesadilla.

Un código que crece en base a la herencia, crece de forma vertical, estructurada, **extensible y reusable**. Así que la composición/agregación y las interfaces, solo cuando son necesarias.

Regla #4: Revisiones de código antes de subir cambios al repositorio, sin excepción

Nombre terrible el de las *Code Reviews*, porque ni son de código (exclusivamente), ni son revisiones. El proceso de hacer una *review* es el de compartir y **explicar tus cambios** a otro miembro del equipo, como paso **obligatorio** antes de subirlos al control de código fuente. Cambios de cualquier tipo, aunque los más frecuentes sean de código.

Es habitual que los programadores perciban las *Code Reviews* negativamente, como un proceso de control, o de evaluación de su trabajo. Lo cierto es que, quizá a excepción de los miembros más *junior* del equipo (donde sí es más necesario ese control), ese es el **último** de los objetivos de una revisión de código.

Es cierto, a veces se detectan fallos en estos procesos de revisión. Y cuando eso sucede, bienvenido sea. Pero no va de eso, porque se asume que un ingeniero sénior no comete fallos graves.

> El auténtico objetivo de las CodeReviews es la **compartición del conocimiento y la unificación de criterios**.

Porque uno de los problemas más graves a los que se enfrentan las empresas, son las *islas de conocimiento*. Que un equipo entero dependa de una sola persona, es una de las peores situaciones en las que se puede encontrar. Y una de las mejores formas de evitarlo son las *Code Reviews*.

Para que sean eficientes y efectivas, han de ser rápidas y concisas (máximo 5-10 minutos). Salvo excepciones, si duran más, probablemente sea porque has acumulado demasiados cambios en tu *commit* o porque tu código es difícil de explicar. Ambas situaciones las deberías corregir en el futuro.

También es fundamental que el proceso de selección de *reviewer* sea aleatorio. Si no, todos tendemos a pedir las *reviews* siempre a las mismas personas. Normalmente, a aquellos miembros del equipo con los que nos llevamos mejor o que sabemos que las hacen más rápido (y sin demasiadas preguntas). Eso, además de injusto (porque unas personas terminan dedicando mucho más tiempo que otras a hacer revisiones), anula completamente el sentido real de las mismas (de compartir conocimiento). En Microsoft, apuntábamos en una pizarra el número de *reviews* que hacía cada miembro del equipo, y la norma era pedir la siguiente siempre al que menos había hecho. De esta forma se aseguraba un proceso semialeatorio y bien balanceado.

Cuando un sistema de *reviews* funciona bien, terminas viendo situaciones curiosas, como que el más *junior* del equipo (que acaba de salir de la Universidad y aún no sabe gran cosa) haga una *review* al gurú más "crack" de toda la empresa. Eso es sano y muy positivo. Así debe ser.

¿Y qué debe mirarse en una *CodeReview*? Pues casi cualquier cosa, empezando por posibles fallos evidentes y terminando por el estilo de código. Si tu empresa utiliza una guía de estilo, la *Code Review* es el momento oportuno para asegurarnos de que todo el mundo la sigue.

Por último, es especialmente importante asegurarse de que el código a revisar sea fácil de entender. Si no lo es, el resultado de la *Code Review* debe ser negativo, y se deben **rechazar los cambios** (luego más sobre esto).

Regla #5: No escribir métodos que hagan más de una cosa

Uno de los fallos que más a menudo me encuentro, son los métodos que resuelven varios problemas a la vez.

> Un método que consulta la cartelera del cine, que calcula el tiempo que va a hacer mañana y además te prepara una tostada con mantequilla, no se puede reutilizar. Tres métodos por separado, sí.

Cuando un método resuelve 3 problemas es un método totalmente específico para el caso de uso concreto que motivó su creación, y por tanto es prácticamente imposible que sirva para nada más. Si separamos esas tres tareas en métodos distintos, es más que probable que sean de utilidad en otros sitios, y podamos reutilizar el mismo código. Además, la solución será mucho más **fácil de entender** y seguir.

Escribir habitualmente métodos extralargos, que pretenden solucionar varias cosas a la vez, suele ser síntoma de una carencia clara a la hora de estructurar bien el código.

Regla #6: No admitir código que no se entienda rápidamente

Como decíamos antes...

> Ningún algoritmo es tan complejo como para que un desarrollador deba invertir más de 5 minutos en entender su propósito y su estructura básica.

Todo el tiempo que exceda de eso, es tiempo perdido.

Además, el código críptico y difícil de seguir suele ser síntoma de una carencia más profunda, relacionada con la capacidad de dividir las tareas en partes más pequeñas y de seguir una línea argumental lógica y con sentido común.

Si permitimos que mucho código así termine en nuestro repositorio, estaremos complicando el mantenimiento de nuestra solución a futuro e incrementando el riesgo de forma innecesaria.

Regla #7: Dedicar tiempo a pensar el lugar apropiado para el código

Otro problema muy frecuente es el de los programadores impetuosos que ante un problema se lanzan a escribir líneas de código demasiado rápido.

Porque con mucha frecuencia se nos ocurren soluciones fáciles y evidentes, que sin duda resuelven el problema, pero que no son la mejor opción si tenemos en cuenta el decálogo que se menciona en este artículo: ¿nuestra solución implica duplicar algo? ¿Complica la estructura del software de forma innecesaria? ¿Establece dependencias que en realidad no necesitamos? ¿Es reutilizable? ¿Es extensible? ¿Es fácilmente entendible?

Antes de lanzarnos a escribir código, merece la pena pararse a pensar todo eso, y dónde debe estar. **Agrupa las cosas de forma conceptual, no funcional**.

> Las operaciones sobre tomates, deben estar con los tomates. Para que luego cuando vengan los tomates Raf y los tomates Cherry, ahorres trabajo y problemas.

Y si poner las cosas en su lugar correcto te genera problemas, probablemente sea un indicativo de que tu software está mal estructurado.

Regla #8: Si te cuesta escoger el nombre de algo, probablemente esté mal diseñado

Este es muy breve...

A menudo me encuentro con nombres de métodos del estilo `processItems`, o *`updateLogic`*. O todo lo contrario:

```
obtenerElementosActualizarSuEstadoHacerUnRecuentoYSalvarEnBaseDe
DatosSiNoExisteYa()
```

Cuando alguien pone un **nombre demasiado vago, o genérico**, suele ser porque intentó poner un nombre mejor y no fue capaz. Porque su método hace demasiadas cosas (ver punto anterior al respecto).

Regla #9: Reutilizar, reutilizar y reutilizar

Otra costumbre de los programadores impetuosos es que se lanzan a escribir cosas que ya están hechas (o parcialmente hechas). Lo cual, además de una pérdida de tiempo, termina creando duplicidades.

Es imperativo **dedicar tiempo a pensar si ya existe algo parecido** a lo que vamos a hacer, y no abrir camino nuevo a no ser que sea estrictamente necesario.

El caso fácil es cuando encontramos algo exactamente igual a lo que necesitamos. Lamentablemente esto no suele ser así, y lo que nos encontramos es **algo parecido o que soluciona parte** de lo que necesitamos. En esos casos, es tentador descartarlo y empezar a programar algo nuevo que se adapte perfectamente a lo que queremos, pero es más que recomendable hacer el esfuerzo de **extender** y adaptar lo que ya hay, lo cual será mucho más sencillo si cumplimos otros consejos incluidos en este decálogo (como el **uso de la herencia**).

Regla #10: No acumular más de un cambio por *commit*

Las revisiones de código, como hemos visto, son eficientes si son sencillas y rápidas. En caso contrario no lo son. Y pocas cosas odia más todo el mundo que una *Code Review* lenta y exasperante... 😡

Además, lo habitual es que todo equipo tenga un *manager* que controle y gestione la evolución del repositorio. Le pedirán planificaciones y estimaciones, y para eso necesita controlar los cambios y analizar el progreso. También tendremos *builds* que se rompen por culpa de un *commit*, y *logs* interminables de herramientas como Jenkins o TeamCity, en los que tendremos que bucear para buscar la causa.

Y para eso, **los *commits* gigantescos** que solucionan 5 *bugs* y completan 7 tareas **son un problema**.

Un momento...

¿10 consejos o reglas sobre programación, y **ni una sola mención a los Test**, a patrones de diseño o a **Martin Fowler**? Exacto.

Después de casi 20 años trabajando, me ha tocado colaborar con ingenieros expertos en prácticamente todos los entornos imaginables: desde desarrollo de videojuegos en C++ hasta aplicaciones corporativas en C#. Desde desarrollo *full-stack* con Java, hasta bases de datos relacionales y no relacionales. Desde software embebido en C puro, hasta desarrollos para la nube...

...Y mucho más a menudo de lo que hubiera imaginado me encuentro con ingenieros muy preocupados por patrones de diseño complejísimos que, sin embargo, ignoran por completo estos principios tan básicos. Se centran en cosas mucho más avanzadas, y se saltan cosas tan fundamentales como: escribe código fácil de entender. Y es especialmente dañino, porque a menudo tutorizan a gente más joven e inexperta, y hacen que su lista de prioridades relacionada con la calidad de software esté completamente desordenada.

Es especialmente frecuente encontrarse con ingenieros que prácticamente delegan toda su calidad en los *test*. Se formaron en los años en los que los *test* eran lo más *trendy*, y su respuesta para prácticamente todo es: "¡Hay que hacer más test!".

No me entiendas mal. Uno de mis objetivos fundamentales en mis últimos años en Electronic Arts fue precisamente trabajar por mejorar el alcance de las pruebas automáticas. Y uno de los temas más recurrentes en los que me tocó trabajar en Microsoft, fue el testeo automático de *pipelines*. Así que considero que **los Test son extremadamente importantes**. Pero no sirven de nada si tu código es una maraña indescifrable, ininteligible, imposible de reutilizar o extender.

Aplicar *test*, y determinados patrones, es lo que viene después. Una vez cumplamos (más o menos) estas pautas, nuestro código partirá de una calidad aceptable. Luego hay que asegurarse de todo lo demás y de que no se rompa al menor cambio 😋

¡Pero eso es materia para otro decálogo!