

# PATH TO SENIOR DEVELOPER

Mastering Code, Leadership & Impact



Matteo Baccan

## Prefacio

El material de este libro surge de la experiencia adquirida en el campo, recopilando una serie de artículos que he seleccionado y desarrollado cuidadosamente a lo largo de varios años de trabajo profesional en el mundo de la programación.

Los artículos, originalmente publicados en Codemotion Magazine, una de las plataformas de información técnica más autorizadas para desarrolladores en Italia, representan un testimonio de mi trayectoria profesional y de las reflexiones acumuladas con el tiempo.

El objetivo es compartir reflexiones, estrategias y profundizaciones técnicas que puedan resultar valiosas para programadores senior, profesionales que ya han consolidado una experiencia significativa pero que siempre están en busca de nuevas ideas para crecer y mejorar.

Cada artículo representa una pieza de un camino de aprendizaje continuo, fruto de desafíos enfrentados, errores cometidos y lecciones aprendidas durante proyectos complejos y contextos laborales desafiantes.

A través de estas páginas, pretendo ofrecer no solo conocimientos técnicos, sino sobre todo una perspectiva más amplia sobre el oficio del programador, compartiendo enfoques metodológicos, mejores prácticas y reflexiones que van más allá del simple código.

Espero que estos aportes, ya apreciados por los lectores de Codemotion Magazine, puedan ser un apoyo concreto para todos los profesionales del sector que desean ampliar sus competencias, innovar constantemente su bagaje técnico y enfrentar con mayor conciencia y maestría los desafíos cada vez más complejos del mundo del desarrollo de software.

Y si al leer algunos capítulos encuentras ideas que contrastan con lo que piensas o si deseas añadir algo, siéntete libre de escribirme. Este libro se actualizará y enriquecerá constantemente gracias a tus comentarios.

DRAFT DRAFT DRAFT

## **Agradecimientos**

Gracias a mi familia, cuyo amor y apoyo incondicional me han permitido realizar este proyecto. Sin su paciencia, comprensión y aliento, este logro no habría sido posible.

Un agradecimiento especial también va para los amigos que me han apoyado a lo largo del camino, ofreciendo consejos valiosos, críticas constructivas y momentos de compartir que han enriquecido mi trabajo.

Finalmente, quiero expresar mi gratitud a todos aquellos que, directa o indirectamente, han contribuido a la realización de este libro. Cada palabra, cada página, es fruto de un trabajo colectivo y de una pasión compartida.

*"La gratitud no es solo la memoria del corazón, sino también la luz que ilumina el camino futuro."* – Anónimo

Gracias de todo corazón.

## Introducción

En el vertiginoso universo de la tecnología, donde el código es el nuevo lenguaje universal, convertirse en un programador senior no es simplemente una cuestión de años de experiencia, sino de crecimiento continuo, pasión y estrategia. Este libro nace con el objetivo de transformar a programadores entusiastas en profesionales expertos, capaces de enfrentar los desafíos más complejos del mundo del desarrollo de software.

Cada capítulo ha sido cuidadosamente diseñado como una etapa de un viaje de transformación profesional. No encontrarás solo conceptos técnicos, sino una hoja de ruta completa que abarca habilidades técnicas, habilidades blandas, metodologías de trabajo y estrategias de aprendizaje. Desde mejorar tus capacidades de programación hasta adquirir una mentalidad de desarrollador senior, este libro te guiará a través de un recorrido de crecimiento profesional único.

Ya seas un joven programador con pocos años de experiencia o un profesional que busca alcanzar el siguiente nivel de excelencia, estas páginas están pensadas para ti. Son el resultado de años de experiencia en el campo, de éxitos, fracasos y aprendizajes continuos en el mundo del desarrollo de software.

Prepárate para un viaje que irá más allá de simplemente escribir código: aprenderás a pensar como un verdadero profesional, a resolver problemas complejos y a construir tu carrera con conciencia y estrategia.

**El mundo de la programación ha cambiado y con él la manera  
de convertirse en programador**



Aún recuerdo el momento en que decidí convertirme en programador: era 1985, las radios sonaban "The Wild Boys" de Duran Duran y tenía que tomar una decisión importante: "¿Computadora o ciclomotor?".

A los 14 años es una elección difícil: por un lado podrías realizar lo que en tu cabeza es la evolución natural de un chico al que le gustan las matemáticas, es decir entender qué es la informática y hacer lo que películas como War Games pintaban como algo al alcance de cualquier niño; por otro lado, comprar un ciclomotor, acelerar libre por las calles, poder ir donde quieras y mostrarte a las chicas fuera del estereotipo del nerd.

Una elección difícil, agonizante, pensada durante meses, pero enfrentada como todas las mejores decisiones de mi vida: instintivamente, de manera completamente aleatoria.

Ganó la computadora.

Después de meses de espera, cuando logré convencer también a mis padres de que era una elección imprescindible, arrastré a mi padre a comprarla.

Mi padre conocía a alguien que tenía una "tienda Olivetti", lo que en los años 80 significaba estar en el centro del mundo informático, y la primera propuesta fue un "Olivetti Prodest".

Era joven, completamente ignorante sobre lo que significaba programar, pero estaba seguro de una cosa: el Prodest era la elección equivocada. Todos tenían el Commodore64, ¿era posible que yo debiera tener un Prodest?

También aquí, varios días para convencer a todos de que el Prodest no tenía futuro. "El Commodore sí lo tenía, era la computadora del futuro".

Con la perspectiva del tiempo entiendo cuán infundada era esta afirmación, no

tenía ningún dato que la respaldara, excepto la publicidad martillante de ese período.

No recuerdo el día de la compra, solo recuerdo mi alegría al armarlo: se trataba de conectar la corriente y un monitor, que en realidad era una vieja TV en blanco y negro.

El primer día estaba devastado: me habían regalado un juego y había pasado toda la tarde jugando.

Pensé que dormiría bien, pero esa noche no pude cerrar los ojos, una mezcla de alegría, euforia y decepción. A esa edad siempre tienes más dudas que certezas: ¿por qué había perdido la tarde jugando, cuando en cambio quería ser "el programador"? Después de todo, ni siquiera sabía qué significaba, pero estaba seguro de que no se trataba de jugar, sino de algo diferente.

En los días siguientes leí varias veces el manual: un cuaderno de anillas, en formato A5, con la cubierta a rayas y el logo de Commodore. Creo que aún recuerdo el olor del papel, pero tal vez sea solo una ilusión de la memoria.

Desde allí comenzaron años de errores, intentos, presuntos éxitos y fracasos.

De lenguaje a lenguaje, de computadora a computadora, de sistema operativo a sistema operativo, de librería a librería, de framework a framework.

Un viaje que duró décadas, donde cada día aprendí algo nuevo, donde cada día entendí que nunca se deja de aprender.

Pero ese es el pasado, un pasado romántico que nunca volverá: ahora programar es algo diferente.

## **Convertirse en programador hoy**

Convertirse en programador hoy ha perdido completamente el encanto que tenía en los años 80 y 90.

Una vez aprender un lenguaje de programación significaba tener trabajo, hoy solo significa tener una herramienta que te ayuda a resolver parte de un problema.

Un lenguaje de programación ya no es un punto de llegada, sino que representa solo el primer escalón de un camino que no tiene fin.

Este aspecto desalienta a muchos jóvenes a emprender la carrera de programador y también los expone a una serie de desafíos que no existían en el pasado, los pone en un mundo donde cada vez más a menudo uno se siente inadecuado, en un mundo donde se tiene miedo de no estar a la altura.

## **La paradoja de la elección y la ansiedad por el rendimiento**

Hoy, un aspirante a programador ya no tiene que elegir entre una computadora y un ciclomotor, sino entre docenas de lenguajes, cientos de frameworks e innumerables trayectorias profesionales: frontend, backend, móvil, ciencia de datos, IA, DevOps, y cada día el mercado inventa una sigla nueva donde todos se sienten desprevenidos: DevSecOps, FinOps y así sucesivamente.

Como observa Barry Schwartz en su libro "The Paradox of Choice" (2004), un exceso de opciones puede paradójicamente llevar a la parálisis decisional y la insatisfacción. En el mundo de la programación, este fenómeno es particularmente evidente: según el Stack Overflow Developer Survey, existen más de 80 lenguajes de programación utilizados activamente, con nuevos frameworks emergiendo constantemente.

Esta abundancia ya no es una riqueza, no significa que podamos hacer cualquier cosa, más bien es una manera de hacer que el programador se sienta cada vez más desprevenido: cualquier camino que elija, el riesgo de terminar en un nicho

de poco valor es cada vez más probable.

Por eso es conveniente despegarse cada vez más de la parte técnica, especializándose en la actitud hacia la resolución de problemas, que es una práctica que tiene un grado de aprendizaje transversal y debería ayudar a resolver la paradoja de elegir el mejor lenguaje.

Para quienes tenían un tupido flequillo en la frente en los años 80, la pregunta que más se hacía era "¿cómo hago mover este sprite?", luego pasó a "¿qué tecnología me garantizará trabajo por los próximos 5 años?" y ahora comienza a transformarse en "¿cómo puedo mantenerme actual en un mundo en constante evolución?".

Si esto no basta, pensemos en la presión creada por las comunidades online. Si en los años 80 la comparación se limitaba a pocos amigos o a las revistas, hoy plataformas como GitHub, LinkedIn o Twitter exponen constantemente el trabajo de miles de otros desarrolladores, alimentando el síndrome del impostor.

La mayoría de los profesionales de TI experimenta el síndrome del impostor al menos una vez en su carrera, con los programadores particularmente expuestos a este fenómeno debido a la naturaleza colaborativa y pública de su trabajo.

Se tiene la sensación de estar siempre un paso atrás, no se tiene tiempo de leer algo y pensar en probarlo, cuando desde los sótanos de un suburbio de Beijing un equipo de desarrolladores ha creado una herramienta que promete revolucionar el mundo del desarrollo y tú, pobre programador que aún es acosado por sus colegas de trabajo, te sientes cada vez más obsoleto.

## **El impacto de la Inteligencia Artificial**

Como si no fuera suficiente, el advenimiento de herramientas basadas en IA como GitHub Copilot, ChatGPT, Gemini o Claude están reescribiendo las reglas.

Si por un lado pueden acelerar el desarrollo y ayudar a superar obstáculos, por otro lado corren el riesgo de crear una generación de programadores que sabe "preguntar" pero no "hacer". El riesgo es perder la comprensión profunda de los mecanismos, delegando el pensamiento crítico a la máquina. Se aprende a usar una herramienta, no a resolver un problema desde los fundamentos.

Como se destaca en el informe "State of AI", casi la totalidad de los desarrolladores de Estados Unidos utiliza herramientas de codificación basadas en IA, pero solo una subparte declara comprender plenamente cómo estas herramientas generan sus soluciones.

Si observamos las estadísticas de OpenRouter sobre el uso de IA a lo largo del tiempo, no podemos dejar de notar un aumento vertiginoso en la adopción de estas herramientas, con un crecimiento exponencial de consultas e interacciones.

De hecho, las IA se han convertido en nuestro compañero de trabajo diario, una herramienta imprescindible en la vida de cualquier programador.

Pero cuanto más pasa el tiempo y su evolución, se hace cada vez más evidente que el programador está destinado a perderse en el mar de código que tendrá que revisar, aumentando de manera incontrolada la sensación de inadecuación que tenemos por naturaleza.

A pesar de esto, Bill Gates afirma:

**Programming will remain a 100% human profession, even a century from now**

Una señal de que probablemente podemos dormir tranquilos.

## **¿Qué queda (y qué se necesita) para no perderse?**

Sin embargo, a pesar de todo, el corazón de la programación no ha cambiado. No

es el lenguaje, no es el framework, no es la IA.

El corazón sigue siendo la **curiosidad**, las ganas de entender, de inventar siempre algo nuevo, sin limitarse a repetir infinitamente lo que hemos aprendido.

Tomemos el ejemplo de BitChat, una aplicación de mensajería descentralizada. Su creación requirió una comprensión profunda de los protocolos de red y la criptografía. La curiosidad de explorar nuevas tecnologías y la determinación de enfrentar nuevos desafíos llevó a su realización: pero ¿cómo se escribió su núcleo? A través de IA, en un fin de semana de vibecoding, pero sin una mente humana detrás del proyecto, ninguna IA (hasta ahora) habría sido capaz de concebirla.

La **lógica** capaz de descomponer un problema en partes diminutas sigue siendo un arte que las IA deben aprender, y es fruto de años de evolución, que estamos tratando de emular día tras día.

Otro aspecto es la **tenacidad** frente a un bug que no se encuentra. Un programador ataca con fuerza bruta, luego reflexiona, estudia y reenfrenta el problema desde nuevos puntos hasta que encuentra una solución aceptable. Actualmente las IA solo tienen una fuerza bruta inmensa que tratan de usar hasta el agotamiento, pero tal vez no sea el mejor enfoque para problemas comprimidos o en los que se necesita una visión holística del producto.

Tal vez, el verdadero desafío para el programador de hoy no es aprender la última tecnología, sino aprender a gestionar el ruido. Aprender a elegir un camino y seguirlo con dedicación, sin dejarse distraer por la frenesí de "lo nuevo a toda costa".

Las nuevas herramientas son una ayuda, no un atajo para no pensar, pero deben ser absorbidas para poder ser explotadas al máximo.

Para convertirse en programadores hoy, se debe primero que todo aprender a

ser como ese niño de 1985: concentrado en un objetivo, curioso de descubrir qué hay dentro de la "caja" y listo para leer el manual, aunque hoy ese manual sea tan grande como toda la web.

Y repensando en Bill Gates, esperemos que sus palabras no se conviertan en un meme como la frase:

640K ought to be enough for anybody!

**"Conviértete en programador después de los 40 años"**



Quien, como yo, pasa muchas horas en línea, no puede no haber visto nunca este título asociado a la venta de cursos de programación. Se trata de un título al borde del clickbait, que intenta convencer a las personas de que, incluso después de los cuarenta, se puede fácilmente convertirse en programadores. El propósito es claro: dado que la carrera de un programador normalmente comienza muy temprano, se quiere elevar el listón tomando un grupo de personas que ya creían que no podían hacerlo o que quizás están en la condición de no tener un trabajo que los satisfaga y quieren hacer algo más: buscan la clásica reubicación profesional.

## **La realidad**

La razón de esta tendencia reside en la creciente demanda de profesionales en el sector de la programación, donde actualmente es difícil encontrar individuos con las habilidades necesarias para satisfacer la demanda del mercado.

Aunque es loable que se intente enseñar programación también a personas que no tienen conocimientos previos, no comparto la manera en que se quiere transmitir este concepto.

Que quede claro, siempre estoy a favor de los cursos de programación y veo con buenos ojos a quien quiere esforzarse y cambiar una situación que no le satisface, pero estoy en contra de los mensajes que inducen a pensar que algo que requiere tiempo y dedicación de repente se vuelva accesible para todos.

Esa frase en mi cabeza suena como un "Conviértete en cirujano después de los 40 años" o "Conviértete en abogado después de los 40 años". No es que no sea posible, pero es difícil, mucho más difícil que a los veinte, especialmente si en la vida se ha hecho todo lo contrario.

**Creo que cada individuo tiene el derecho de elegir su propio camino en la vida: uno puede despertarse un día y decidir que la programación es su futuro, así como un programador puede decidir convertirse en un criador de vacas.**

Intentemos, sin embargo, traer un poco de realidad a esta frase e imaginar qué sucederá al final del curso que te convertirá en programador, independientemente de la calidad de la enseñanza y de las capacidades personales.

Inevitablemente, será necesario buscar un empleo ya que intentar la suerte como freelance sin contactos y experiencia podría ser una perspectiva bastante frustrante.

Una vez encontrada una empresa que esté dispuesta a invertir en ti, podría haber una serie de sesgos que podrían cambiar la manera en que los potenciales empleadores se acercan a ti. El primero de estos sesgos es sin duda la edad.

## **Sesgos de edad**

Intentemos considerar dos escenarios para simplificar el razonamiento:

Si tienes 20 años y te adentras en el mundo de la programación, después de haber hecho un curso o un camino universitario, las expectativas sobre ti serán indulgentes. La falta de experiencia se perdonará normalmente y se espera un gran deseo de aprender. Si la empresa cree realmente en la formación, no es inusual tener también un mentor dispuesto a ayudarte en tu camino de crecimiento profesional.

Por otro lado, si tienes 40 años, las expectativas hacia ti son mayores y podría haber muchos sesgos que se anteponen a tu preparación. Sesgos de edad, sesgos relacionados con tu situación familiar, sesgos sobre tus límites de aprendizaje en comparación con los chicos de veinte años: rápidos para programar y con mucho más tiempo disponible.

Estas discrepancias de expectativas representan un aspecto crucial a considerar y para el cual es necesario prepararse mentalmente.

Es necesario estar muy motivado si se quiere seguir este camino que, aunque no es imposible, al principio puede ser realmente duro y podría desarrollar una profunda frustración. Es importante tener en cuenta estos aspectos y no desanimarse por las dificultades iniciales, porque solo con el tiempo se lograrán demostrar las propias capacidades y hacer valer la propia experiencia.

En definitiva, no existe una regla fija, pero es fundamental ser consciente de estas dinámicas para evitar subestimar los desafíos que podrían presentarse después de completar el curso y durante la búsqueda de empleo en el sector de la programación.

Sin duda, uno de los factores mitigantes, que puede derribar o al menos mitigar cualquier preconcepto, es demostrar que se está muy motivado, no solo en palabras, sino también con acciones tangibles. Ser activo en plataformas como GitHub, participar en eventos del sector y colaborar en proyectos de código abierto son todos elementos que demuestran un compromiso concreto y una pasión por la programación.

Estos "pequeños fragmentos" no solo testimonian el deseo de aprender y de ponerse a prueba, sino que también pueden revelar un talento latente que comienza a explotar en ese momento, contribuyendo a reforzar la credibilidad y el atractivo a los ojos de los posibles empleadores.

## **La propuesta formativa**

Quisiera ahora detenerme también en otro aspecto que emerge leyendo con más atención algunas propuestas formativas, he encontrado algo que me ha dejado perplejo y que me ha impulsado a escribir este artículo.

Un comentario recurrente, que se encuentra en diversas formas bajo este tipo de trayectorias y que quisiera mencionar como ejemplo, es el siguiente:

"Nosotros enseñamos cómo convertirse en un arquitecto de software, de manera constante... para no tener que hacer la gaveta".

El inicio de estos cursos es a menudo enfocado en "programadores cuarentones" y, con igual frecuencia, intenta vender el concepto de que las personas se convertirán en "arquitectos de software sin gaveta".

Puedo comprender el deseo de ofrecer cursos a personas que aspiran a convertirse en programadores, y reconozco la existencia de un mercado en el que buscar potenciales talentos. Sin embargo, permanezco perplejo ante el mensaje que se quiere transmitir, según el cual convertirse en arquitectos de software después de un curso no requeriría "gaveta".

Es importante subrayar que la experiencia, o "gaveta", es fundamental para desarrollar una comprensión precisa del diseño de software funcional, capaz de satisfacer las necesidades de los clientes y de los usuarios.

La "gaveta" implica cometer errores, aprender de ellos y enfrentar desafíos prácticos en diferentes proyectos. Significa tomar decisiones difíciles, a veces en contraste con las mejores prácticas, porque el contexto o las necesidades del proyecto lo requieren.

Cortar camino en la "gaveta" corre el riesgo de producir proyectos que, aunque puedan parecer bien realizados, no logran traducirse en soluciones viables debido a problemas como costos, rendimiento, tiempos o simplemente falta de experiencia.

Si después de completar un curso y obtener el título de "arquitecto de software", te preguntas por qué no puedes encontrar trabajo, la respuesta es simplemente la falta de "gaveta". Ningún curso puede reemplazar la experiencia práctica y el aprendizaje que conlleva: la mera participación en un curso no te hace un programador profesional o un arquitecto de software.

Como lo ha dicho alguien que ha tecleado más veces que yo: "No hay un atajo para convertirse en un programador profesional. Es necesario enfrentar desafíos reales y aprender de los errores en el camino." ("The Pragmatic Programmer: Your Journey to Mastery"). Esta frase captura de manera excelente el concepto de que la programación es una habilidad que puede ser enseñada, pero es solo a través de la práctica que se puede aspirar a convertirse en verdaderos profesionales.

Para elevar el nivel, cuando hablamos de la figura del arquitecto de software, nos enfrentamos a un discurso diferente.

Ser un arquitecto de software no es algo que se improvise; requiere años de experiencia y práctica. Se trata de un trabajo que involucra la estructura completa de una empresa y puede decretar su fracaso o éxito según cómo se aborde el trabajo. Aunque un curso de formación puede ser útil y profundo, no puede reemplazar la experiencia adquirida con el tiempo y siempre proporcionará una visión parcial del trabajo de un arquitecto de software.

Otro aspecto importante es no confundir las figuras de "programador" y "arquitecto de software": son dos roles distintos, cada uno con sus propias competencias y experiencias. Ser un buen programador no implica necesariamente ser un buen arquitecto de software y viceversa. Sin embargo, es innegable que tener un trasfondo como programador puede ser ventajoso para aspirar a convertirse en un buen arquitecto de software.

Podemos hacer un paralelo con el mundo del fútbol: muchos entrenadores exitosos han tenido una carrera como futbolistas y han vivido en primera persona las dinámicas del juego. Al mismo tiempo, hay casos de jugadores que, impulsados por la prisa de convertirse en entrenadores, han quemado etapas sin tener la experiencia adecuada, poniéndose al frente de equipos importantes y comprometiendo su credibilidad y carrera.

Mientras que la práctica es fundamental para convertirse en buenos

programadores, convertirse en un arquitecto de software requiere una profunda experiencia y comprensión del sector que solo el tiempo puede proporcionar. Por lo tanto, es necesario ser consciente de los caminos y los tiempos necesarios para alcanzar los objetivos profesionales deseados.

## **Conclusiones**

Cualquiera puede convertirse en programador, a cualquier edad, pero es importante ser consciente de los desafíos y expectativas que surgirán en el camino.

Cuanto más tarde comiences, más difícil será, pero no imposible. La pasión y la dedicación pueden ayudar a superar las dificultades iniciales y demostrar su valor en el sector de la programación.

Después de algunos años de experiencia profesional, comprenderás cuán fundamental es enfrentar la fase de aprendizaje práctico y cuán ilusorio es creer que un curso puede reemplazar la importancia del "aprendizaje" en el sector de la programación.

La práctica y la experiencia en el campo son pilares indispensables para una carrera exitosa en programación. Aunque los cursos pueden proporcionar una base teórica sólida, es solo a través del trabajo práctico y la investigación continua que realmente se puede adquirir la competencia y el dominio necesarios. Por lo tanto, es esencial reconocer la importancia del "aprendizaje" y mantener expectativas realistas sobre los caminos de formación y desarrollo profesional en el sector de la programación.

"He hecho el curso, pero si no lo practico, lo olvido": se convertirá en una frase que escucharás a menudo y que te hará reflexionar sobre las verdaderas competencias adquiridas.

## **Artículos relacionados**

The Pragmatic Programmer: Your Journey to Mastery  
Se puede aprender a programar después de los 40 años?

DRAFT DRAFT DRAFT

**Los programadores que completan todas las tareas no han terminado su jornada laboral**



Está surgiendo entre los programadores y los equipos de desarrollo un enfoque demasiado individualista y perjudicial en la forma en que se gestionan los proyectos de software. El objetivo de las personas parece haberse convertido únicamente en cerrar sus propias tareas lo más rápido posible, sin preocuparse por cómo esto puede afectar el trabajo de los demás miembros del equipo y el hecho de que este enfoque solo aporta un valor efímero al proyecto.

## **Cerrar tareas no es el objetivo final**

Este enfoque también podría ser el primer indicio de una falta de gobernanza por parte de la empresa, se produce en proyectos altamente estructurados por tareas, donde a cada persona se le asigna una tarea que debe completarse lo antes posible y sobre la cual se evalúa el rendimiento de su trabajo.

Ese programador es una máquina de guerra, termina todo lo que se le asigna por la mañana y luego se pone a hacer otra cosa.

¿Cuántas veces has escuchado esta frase y la satisfacción generalizada por parte de colegas y gerentes? Pero, ¿es realmente tan positivo?

Para lograr cierto rendimiento, se tiende a encerrar a la persona en un entorno aislado, despojándola de cualquier preocupación, para impulsarla hacia el máximo resultado.

Desde cierto punto de vista, esto es positivo: se eliminan todas las distracciones, las reuniones constantes, las interminables llamadas telefónicas y las situaciones de multitarea para enfocarse en el objetivo. Desde otro punto de vista, se pierde de vista el panorama general y se corre el riesgo de crear un ambiente de trabajo tóxico, altamente individualista, en el que las personas no se ayudan mutuamente ni comparten sus conocimientos.

El hecho de que muchos programadores sean propensos al aislamiento cuando realizan su trabajo no ayuda a esta situación.

## **El síndrome del caballo de carreras**

Se entiende lo fácil que es, en un entorno como este, no darse cuenta de los descontentos o de la desconexión entre las diferentes figuras, si solo se mira el resultado a corto plazo. Al igual que un caballo de carreras lleva anteojeras para correr rápidamente hacia la meta sin mirar el mundo exterior, los programadores se lanzan a resolver sus propias tareas.

**Lo importante es terminar la tarea, no la economía del proyecto.**

## **Premiando el individualismo**

Se premia el individualismo, vinculado al resultado tangible.

Es una lástima que un proyecto también esté compuesto por una serie de resultados intangibles, como compartir conocimientos, colaboración y crecimiento personal y profesional de las personas involucradas.

Las métricas establecidas por la gerencia y muchas herramientas utilizadas en la empresa tienden a evaluar todo lo tangible: el número de tareas cerradas, las líneas de código escritas, el tiempo entre la apertura de un problema y su resolución: rara vez tienen en cuenta las tareas intangibles.

Piensa en todas las veces que has ayudado a un colega a superar un problema, corregido una línea de código que resuelve un problema conocido desde hace meses y nunca resuelto, las sesiones de programación en pareja o los consejos dados durante el café.

Por lo tanto, es necesario pensar en los proyectos de manera holística y tratar de alejarse de la lógica de la tarea como único objetivo. Un equipo que trabaja junto debe alcanzar un objetivo común. No ganan los individuos, sino que gana el grupo que es capaz de evolucionar y mejorar el producto de manera continua y

armoniosa.

## **El problema de "terminar las tareas"**

El verdadero desafío es convertir a un grupo de personas, insertas en un contexto donde se premia el rendimiento individual, en un equipo que piensa, actúa y trabaja hacia objetivos a largo plazo.

Trabajar en equipo no es solo una cuestión de habilidades blandas, sino también de cultura empresarial y de cómo se evalúa a las personas dentro de la empresa.

Crear compartimentos estancos no beneficia a nadie y debe equilibrarse con actividades que aporten valor incluso a las personas más lentas o aisladas, para poder impulsarlas hacia una mejora cultural y personal.

Cuando se tienen equipos compuestos por personas más o menos experimentadas, si se encuentra la manera de transferir conocimientos de una persona a otra, se convierte en un enriquecimiento tanto para el proyecto como para las personas individuales.

**Para aquellos que gestionan un proyecto de software, es fundamental utilizar el tiempo de las personas de la mejor manera posible para aumentar el valor del proyecto además de cerrar las tareas.**

Es esencial fomentar una cultura colaborativa dentro del equipo, donde las personas puedan ayudarse mutuamente y compartir conocimientos.

## **¿De quién es la responsabilidad?**

Sin duda, fomentar un enfoque colaborativo es responsabilidad de la gerencia, pero también los miembros individuales del equipo desempeñan un papel importante en este proceso. Es fundamental que los miembros del proyecto sean

conscientes de que su trabajo no se limita únicamente a cerrar sus propias tareas y regresar a casa lo antes posible.

En las reuniones periódicas, en las reuniones de pie, es responsabilidad de los miembros del equipo informar a los demás si hay bloqueos o si necesitan ayuda. De la misma manera, los miembros del equipo deben estar dispuestos a ayudar a sus colegas compartiendo sus conocimientos y habilidades.

Sin embargo, esto no significa que este aspecto surja de forma espontánea. Por eso es necesario que la gerencia promueva activamente la colaboración, identificando estas situaciones y alentando a las personas a trabajar de manera colaborativa.



**¿Qué estrategias podemos implementar para mitigar este problema?**

Existen varias formas de evitar que este enfoque comprometa la economía del proyecto.

Si hablamos de habilidades blandas, definitivamente debemos incluir en el equipo de trabajo personas que tengan una predisposición natural para el trabajo en equipo, que sepan trabajar en grupo y que sean capaces de compartir sus conocimientos.

Si una persona termina sus tareas y no tiene nada más que hacer, se le puede animar a realizar revisiones de código del trabajo de otros miembros del equipo, a realizar refactorizaciones del código recién agregado o a brindar mentoría y capacitación a las personas menos experimentadas, y el rendimiento individual debería evaluarse en función de este tipo de actividades.

## Gestión de "SuperStar"

En un mundo perfecto, todos pueden hacer su trabajo, almorzar en casa con la familia y jugar al tenis por la tarde.

En un mundo real, nos enfrentamos a problemas y no todos se pueden resolver por sí solos. A veces es necesario pedir ayuda a los llamados "programadores SuperStar".

En este sentido, me viene a la mente un discurso de Ottavio Bianchi a los jugadores del Napoli cuando Maradona se unió al equipo:

Él es Maradona y nosotros somos nosotros. El problema surge si alguno de nosotros quiere ser Maradona. Si aceptamos esto, es decir, que él es él y nosotros somos otra cosa, podemos ganar, pero si nosotros no lo aceptamos o alguno de nosotros no lo acepta, las cosas no funcionan.

A veces, la SuperStar es necesaria para cerrar algunas tareas: imagina todas las

situaciones en las que te has encontrado con un problema que una persona muy competente en el tema habría resuelto en minutos, mientras que tú has tardado días.

En estos casos, es fácil que la persona individual trabaje en tareas bien definidas y que no necesite, o tenga poca necesidad de ayuda de los demás.

La obsesión por completar la tarea por parte de la SuperStar no es un problema, pero es fundamental que la gerencia sea capaz de gestionar estas situaciones y garantizar que el trabajo de la "SuperStar" sea comprensible y mantenible para todos los miembros del equipo.

Por eso es fundamental que el equipo analice posteriormente el código escrito por la "SuperStar", lo revise, lo documente y realice refactorizaciones para que sea comprensible y mantenible para todos.

### **El efecto de la "cadena de montaje"**

Por lo tanto, debemos evitar pensar en la programación como una cadena de montaje, en la que cada persona es responsable de una sola tarea y no tiene ninguna responsabilidad hacia los demás.

¿Por qué has puesto esta línea de código que ralentiza el programa?  
Porque de esta manera resolví una advertencia de seguridad.  
¿Pero no te diste cuenta de que podrías haberlo hecho de esta manera?  
No, no me di cuenta, no era mi tarea.

Sin embargo, ten cuidado de no abusar del enfoque opuesto, donde tan pronto como una persona termina sus actividades, se le asignan continuamente nuevas tareas o se cierran los trabajos de otras personas.

Esto podría llevar a situaciones en las que las personas perciben un aumento del

estrés y una carga de trabajo excesiva, lo que les impide gestionar de manera óptima sus propias tareas.

El "burnout" es un problema real y tangible, y una situación mal gestionada podría fomentar la insatisfacción laboral y el alejamiento de las personas más talentosas, que se sienten abrumadas por la cantidad de tareas que deben completar.

Del mismo modo, puede haber personas que deliberadamente ralenticen su trabajo para evitar tener que hacer el trabajo de los demás. En estos casos, es fundamental que la gerencia sea capaz de gestionar estas situaciones, garantizar que la carga de trabajo se distribuya equitativamente entre los miembros del equipo y poner en práctica una serie de estrategias para mejorar la colaboración dentro del equipo.

## **Conclusiones**

Un buen programador no es aquel que completa todas las tareas asignadas, sino aquel que se le permite trabajar de manera colaborativa y compartir sus conocimientos con los demás miembros del equipo.

Es necesario hacer crecer al grupo de trabajo en el que uno está inserto, y es importante que la gerencia no evalúe sus resultados solo en función de datos numéricos, sino que sea capaz de gestionar a las personas, escucharlas, motivarlas y comprender dónde están las diferencias para nivelarlas.

El valor que puede aportar un senior va más allá de la resolución de tareas, debe extenderse a actividades de coaching y mentoría, para hacer crecer a aquellos con menos experiencia.

Esta es la mejor manera de garantizar que el proyecto de software sea exitoso y que el equipo sea capaz de crecer y mejorar con el tiempo.

## **La fuerza de admitir que no se sabe**



El mundo de la programación es un universo en constante evolución, donde las habilidades técnicas representan solo el punto de partida.

Todos miran con temor el mundo de la inteligencia artificial, en teoría software capaz de saber más que cualquier programador y capaz de generar código con la facilidad de beber un sorbo de agua, pero la diferencia entre un buen programador y un gran programador no reside solo en la capacidad de escribir líneas de código, sino también en la habilidad de interpretar correctamente las especificaciones, de ver más allá de lo que se pide, de reconocer cuándo un camino es incorrecto y sería conveniente tomar otro.

Todas estas habilidades se adquieren con la experiencia y, por el momento, no existen inteligencias artificiales con tales características.

Si te defines como programador, pero aún así tienes miedo: es normal.

El miedo de no saber lo suficiente, de ser superado por un software es una sombra que acompaña constantemente el camino profesional de cada programador.

Si acabas de entender la diferencia entre un `for` y un `while` y te cae una gota de sudor cuando ves una terminal: es perfectamente normal. Eres un junior y te asustas por cosas que un día aprenderás a hacer con los ojos cerrados.

Pero este miedo también pesa sobre los hombros de quienes llevan años trabajando, que han escrito más ciclos for que su propio apellido, que identifican los errores solo con mirar el código y que poseen una visión holística de los proyectos.

## **El miedo a no saber**

El miedo a no saber es una constante de este trabajo que debe ser pronto aceptada y superada, ya que nos bloquea, impidiéndonos crecer y mejorar.

Para ser programador se necesitan habilidades, y cuanto más se adquieran, más se da uno cuenta de que siempre hay nuevas cosas por aprender. Al mismo tiempo, no es posible detenerse, limitarse a un nicho o especializarse en un solo ámbito, porque siempre habrá algún desarrollador que, tal vez trabajando desde su cabaña en la cima de un árbol, creará código destinado a convertirse en fundamental para tu trabajo.

No podemos conocer plenamente todos los productos que utilizamos ni todo lo que se lanza cada día: o dedicamos tiempo a entender qué sucede o pasamos los días escribiendo código. Pero incluso si fuéramos capaces de hacer ambas cosas, no seríamos capaces de conocer todo.

¿Quién lo sabe todo? ¿Las inteligencias artificiales? No, ni siquiera ellas. Aprenden lo que pueden asimilar de la red, pero si el dato no está presente o se interpreta mal, el resultado será erróneo: existe un mundo de código en el ámbito legacy, sin documentación que las inteligencias artificiales no conocen. Están todos los productos de código cerrado que no pueden ser analizados, todos aquellos que aún deben ser lanzados y así sucesivamente.

Así que pongámonos en paz: no podemos saberlo todo y, incluso con las herramientas más modernas, nunca seremos capaces de conocer cada cosa. Siempre habrá grados de ignorancia o de error en todo el código que escribamos.

De la misma manera, todo lo que escribimos hoy podría resultar incorrecto: porque no conocemos plenamente un producto y porque la tecnología evoluciona y cambia, y lo que hoy se considera correcto, mañana podría ser considerado erróneo.

## **Era ignorante y no lo sabía**

Últimamente he trabajado en la optimización de un producto Java al que se le pedía ejecutar una serie de operaciones dentro del umbral de 10 milisegundos.

El código parecía correcto, pero ocasionalmente el programa superaba los 100 milisegundos. En un entorno donde incluso los milisegundos tenían un peso, esta oscilación no era tolerada porque superaba el SLA del servicio que debíamos proporcionar.

¿Cuál era el problema? La instanciación de un objeto llamaba al cargador de clases para buscar un cierto tipo de clase, en lugar de instanciar directamente una clase. Esta operación, que en el 99.999% de los casos se resolvía en menos de 1 milisecondo, en algunos casos llegaba a 100 milisegundos.

¿La solución? Evitamos la búsqueda de la clase de manera dinámica, pasando a una instanciación estática. Peso de la modificación: 1 línea de código, pero el retraso ocasional desapareció.

No conocíamos tan profundamente la clase que usábamos, y en condiciones normales el problema no existía: aparecía solo en momentos de estrés aplicativo en fuerte concurrencia.

¿Cómo lo entendimos? Hablándonos entre nosotros, confrontándonos, juntando nuestras habilidades y experiencias.

## **La ignorancia es un recurso**

En este mar de conocimiento – o de ignorancia, si se quiere ver el vaso medio vacío – lo único que podemos hacer es potenciar aquellas que son las habilidades blandas, en su momento subestimadas y hoy más que nunca esenciales.

Que quede claro: las habilidades técnicas son fundamentales, así como es esencial conocer y usar bien las nuevas herramientas que se nos ponen a disposición cada día, pero para quienes han pasado años trabajando con el código, hay habilidades que marcan la diferencia cuando se colabora en equipo.

Dado que libros, manuales e inteligencias artificiales nunca tendrán todas las

respuestas, lo que más aprecio es el **coraje de preguntar**.

Hay habilidades y experiencias que residen solo dentro de las personas, y a menudo es más fácil preguntar a quien sabe en lugar de tratar de entender por uno mismo.

Por miedo a ser humillados o ridiculizados, no nos atrevemos a hacer una pregunta a un colega, no decimos "no entendí", no tenemos el coraje de pedir aclaraciones. Pasamos horas frente a piezas de código incomprensibles, pedimos ayuda de forma anónima en foros, Reddit, Stack Overflow o a varias inteligencias artificiales, pero en lugar de dirigir una simple pregunta al colega sentado junto a nosotros, preferiríamos casi prendernos fuego. No hacemos grupo y preferimos rompernos la cabeza solos frente a un problema.

El orgullo se transforma en una cadena invisible que nos limita, nos paraliza, nos hace desperdiciar tiempo valioso y nos induce a cometer errores evitables.

Tememos ser juzgados como débiles, pero en realidad, admitir que no se sabe es una gran virtud.

Una vez un antiguo filósofo dijo:

**Tu mantra para la vida debe ser "dignidad cero"**

En realidad no era un filósofo, sino Mauro Repetto, fundador de los 883, pero el mensaje que llega es el mismo: **seguir tus sueños sin preocuparte por las críticas o el juicio ajeno**.

Muchas veces es de hecho el miedo al juicio lo que nos detiene y no nos permite crecer. Pero si no preguntamos, ¿cómo podemos aprender? Si no admitimos que no sabemos, ¿cómo podemos mejorar?

## **Gestión de la ignorancia**

Todos somos ignorantes, pero en temas diferentes. Admitir que no se sabe y manifestar este aspecto en la empresa a veces puede mover recursos.

Hace años me encontraba en una empresa donde habíamos decidido apostar por C++ para el nuevo producto que queríamos realizar: los motivos eran la velocidad de ejecución y la posibilidad de escribir código más cercano al hardware. Con el tiempo la elección se reveló equivocada, no por los supuestos iniciales, sino por el hecho de que se dirigía a un grupo de programadores que nunca habían trabajado con ese lenguaje y provenían de lenguajes más simples.

En ese momento la dirección se dio cuenta de una laguna: el equipo en el que estaba y que debía desarrollar en C++ no tenía, o tenía solo en parte, las competencias adecuadas para poder realizar el producto que nos habíamos propuesto. Por este motivo se organizó un curso con un experto del lenguaje, para desbloquearnos en toda una serie de aspectos que no habíamos considerado y no conocíamos.

Manifestar nuestra ignorancia nos hizo dar un salto adelante, nos permitió entender dónde estábamos fallando y corregir el rumbo, pero sobre todo nos hizo comprender que no estamos solos y que pedir ayuda es normal.

Si esto vale para un equipo, donde quizás es incluso más fácil preguntar, vale mucho también para los individuos.

## **El coraje de preguntar**

Hace años trabajaba para una empresa que había contratado a varios chicos recién graduados. Sus habilidades eran variadas: algunos ya demostraban cierto talento, otros menos. Sin embargo, uno en particular se destacaba de los demás no por sus capacidades técnicas – de hecho, tenía notables lagunas – sino por su extraordinaria predisposición a pedir aclaraciones, incluso repetidamente durante el día. Esta actitud, aparentemente simple pero en realidad revolucionaria, con el

paso del tiempo lo llevó a superar a todos sus compañeros.

Su ChatGPT eran los colegas, las personas más experimentadas que él, que lo guiaban, le explicaban qué debía hacer y no hacer y dónde debía mejorar su estudio.

El miedo a hacer el ridículo es a menudo una jaula que nos mantiene atrapados en nuestra zona de confort. Tenemos una elección: permanecer prisioneros del miedo o liberarnos con el coraje de preguntar. El camino para convertirse en mejores programadores comienza aquí.

## **El mito del sabelotodo**

El mundo tech abunda en personas que se erigen como sabelotodos, convencidas de saberlo todo y de no necesitar ninguna ayuda externa. Esta peligrosa ilusión nace de una imagen distorsionada de la perfección. Consideren a los superhéroes como metáfora: se representan como seres perfectos, infalibles, autosuficientes e intrépidos. Pero no somos superhéroes con superpoderes: somos seres humanos, con todos nuestros límites y nuestras fragilidades.

Intenten mirar también a todas las personas que siguen en las redes sociales. A menudo parten de temas que conocen, son muy hábiles al hacerlo y gracias a esto atraen seguidores. Con el tiempo, de manera más o menos inevitable, la vena creativa disminuye y la búsqueda desesperada de temas interesantes lleva a hablar de cosas que no se conocen bien, pero que se piensa que pueden interesar.

En este punto la calidad se deteriora, pero dado que el público sigue apreciando y los seguidores aumentan, se persevera en este camino, hasta que uno se encuentra hablando de temas que no se conocen en absoluto.

Este es el momento en que uno se convierte en sabelotodo, se cree que se puede hablar de todo y saberlo todo, pero en realidad es solo un engaño.

Esta situación le ocurre más o menos a todos con el tiempo, pero es importante reconocerlo y no caer en la trampa del sabelotodo.

Si tu audiencia te ve como una guía, como un punto de referencia, es importante en algún momento detenerse, entender el propio límite, admitir que no se sabe y hacer intervenir a quien es experto en un cierto tema.

La diferencia entre un sabelotodo y un experto está aquí: si somos expertos y queremos lo mejor para un cierto tema, iremos a preguntar a quien, sobre un tema particular, es más competente que nosotros y sabrá mejor guiarnos, y no siempre se trata de una inteligencia artificial, sino de una persona física, con nombre y apellido.

## **Preguntar solo no basta**

¿Hemos encontrado el coraje de preguntar? ¿De hacer intervenir al experto? En este punto podría surgir un segundo problema: fingir entender.

Ya preguntar es un gran paso, pero admitir que no se ha entendido una respuesta es un segundo paso que a menudo no se da. Se prefiere fingir haber comprendido, aunque no sea así, por miedo a ser juzgados como tontos.

### **Repetir y reformular**

Una manera simple de verificar la comprensión de un concepto es repetirlo y reformularlo con nuestras propias palabras. Si no se logra hacerlo, significa que no se ha entendido bien y es necesario pedir más aclaraciones.

## **La diferencia entre jamón cocido y jamón crudo**

Yo no logro distinguir el jamón cocido del jamón crudo, es algo que llevo arrastrando desde hace muchos años. Mi cerebro se niega a asociar la palabra al

producto. Sé lo que quiero, pero no logro asociar la palabra correcta.

**Pásame el jamón cocido**

es una de las preguntas más embarazosas que me pueden hacer, porque sé que la respuesta será:

**Eso es jamón crudo**

Con el tiempo entendí el problema: asociaba al color oscuro la cocción y al color claro la no cocción del alimento. Mi cerebro asociaba al contrario las palabras, e invertía el resultado, sabiendo que había algo que no iba bien, pero sin lograr entender qué.

A fuerza de equivocarme empecé a preguntar, y después de muchas explicaciones, muchos modos de recordar, entendí cómo distinguir el jamón cocido del jamón crudo: pero si no hubiera empezado a preguntar, quizás nunca habría tenido un mecanismo en la cabeza capaz de hacerme entender la diferencia.

## **El miedo al juicio**

Ser juzgados, ser vistos como el que no sabe, es un miedo que nos bloquea y nos impide crecer. Pero si no preguntamos, ¿cómo podemos aprender?

De la misma manera, si superamos este miedo en la pregunta, debemos superarlo también en la respuesta: no basta con preguntar, es necesario también entender la respuesta y admitir que no se comprende. Preguntar sin entender equivale a no preguntar, pero también admitir que no se ha comprendido la respuesta es un arte que debe ser aprendido con el tiempo.

## **Cultura de avanzar a través del fracaso**

Otro concepto que me fascina es el de la cultura de avanzar a través del fracaso, es decir, la cultura del fracaso. Fallar es normal, es humano, es un paso necesario para crecer y mejorar.

Admitir que no se sabe, equivocarse y usar el fracaso como trampolín para mejorar es una de las claves para convertirse en un mejor programador.

Desafortunadamente, no saber, fallar, mostrarse vulnerables todavía se percibe como un defecto: en realidad representa una poderosa herramienta para crecer, aprender y mejorar, una lección que debería ser transmitida desde la infancia.

No te detengas por miedo a equivocarte, sino equivócate para aprender

## Conclusiones

La próxima vez que dudes en hacer una pregunta, pregúntate: ¿prefiero parecer momentáneamente inseguro o quedarme estancado para siempre?

Y si alguien se burla de ti por tus preguntas aparentemente ingenuas, recuerda que es precisamente él, al no hacerlas, quien se limita y construye lentamente una jaula invisible de la que será cada vez más difícil salir.

Nadie nace sabiendo todo, pero solo quien tiene el coraje de admitir que no sabe puede realmente aprender y crecer.

## **Más allá del Full Stack: Caminos de Crecimiento en el Desarrollo de Software**

El viaje para convertirse en un desarrollador senior es un proceso complejo que va mucho más allá de la simple adquisición de habilidades técnicas. "Más allá del Full Stack: Caminos de Crecimiento en el Desarrollo de Software" explora cuatro pilares fundamentales de esta transformación profesional, comenzando por la deconstrucción del mito del Desarrollador Full Stack - descrito como una figura que "no es experto en backend, ni en frontend, ni en DevOps; conoce nociones de UX, UI, SEO, marketing, sabe un poco de todo, pero nada en profundidad" - hasta la importancia de las revisiones de código y el mantenimiento del software.

Estos capítulos nos guían a través de una metamorfosis profesional esencial: desde superar la ilusión de tener que dominar todos los aspectos del desarrollo de software hasta la conciencia de las consecuencias de cada modificación del código ("cada modificación, incluso la más trivial, puede causar problemas de rendimiento, regresión, seguridad, mantenibilidad").

Se analiza la adopción responsable de nuevos frameworks, advirtiendo contra el riesgo de "desperdiciar tiempo y recursos" en tecnologías no sostenibles a largo plazo, hasta llegar a la importancia crucial de las revisiones de código, donde "no se trata de imponer un estilo propio, sino de encontrar un compromiso que pueda ser aceptado por todos."

Descubrirás cómo la evolución de programador junior a senior se manifiesta en la capacidad de evaluar críticamente cada decisión técnica, desde el impacto de una sola condición en el código ("la pérdida de despreocupación y un conjunto de miedos regresivos") hasta la gestión de la deuda técnica y la eliminación de código obsoleto ("mantener código no utilizado es costoso y con el tiempo representa una deuda técnica innecesaria").

A través de ejemplos concretos y reflexiones profundas, este texto te

acompañará en el proceso de maduración profesional, destacando cómo cada decisión técnica debe ser evaluada no solo por su impacto inmediato, sino también por sus consecuencias futuras en el sistema y en las personas que lo mantendrán.

**¿Puedes cambiar esta condición en el código? ¿Qué se necesita?**



¿Qué diferencia a un programador senior de un programador junior? Muchos dirán que es la capacidad de evaluar las consecuencias de una modificación en el código, pero en realidad es la pérdida de despreocupación y un conjunto de miedos regresivos.

## **El miedo de los senior**

Antes solía ser mucho más rápido al cambiar un algoritmo, al modificar un parámetro de una función, al crear o destruir flujos, y así sucesivamente. El objetivo era, ante todo, hacer lo que pedía el cliente y solo en segundo lugar asegurarse de que el código fuera mantenible o de que la elección tomada fuera la mejor dentro de una serie de opciones más o menos ponderadas.

Luego envejecí y comencé a usar pruebas para entender si mi código seguía siendo bueno después de una modificación, empecé a escribir documentación para entender por qué había razonado de cierta manera y comencé a hacer revisiones de código para mejorar la comprensión del código y su mantenimiento.

Este nuevo enfoque "complicó" mi trabajo, o mejor dicho: alargó los procesos que, desde una solicitud, llevaban a la solución final. Una modificación que antes hacía en 10 minutos, ahora ocupa un día entero, porque debo evaluar todos los impactos que esa modificación puede tener, ejecutar todas las pruebas automatizadas, realizar algunas pruebas manuales donde no pude automatizar, escribir la documentación, hacer el merge con el código principal, esperar la compilación y las pruebas de integración, verificar que todo funcione y finalmente declarar que el problema está resuelto, esperando no haber olvidado algo.

Los años pasan y es inevitable que haya una evolución desde el niño despreocupado que hace un push sin pruebas, hasta el cariatide que ajusta incluso los espacios cuando debe agregar código a la rama principal (no, ya no se usa "master").

Lo que una vez fue un "sí" feliz y entusiasta, se convierte en un "sí, pero también hago esto", poco a poco se convierte en un "tal vez", hasta llegar a un "no" o peor aún un "depende".

## **Los senior no son malos**

No es por maldad que todo esto sucede, es por la conciencia de todo lo que rodea a una sola línea de código. La conciencia del error o de los impactos de una modificación y por un vasto retroceso que muchas veces ha minado la certeza de que una sola condición pueda cambiarse sin problemas. Esto no significa que las modificaciones no deban hacerse, solo significa que siempre se deben evaluar los impactos, para no crear problemas más grandes de los que se pretenden resolver y que a menudo, cuanto más se envejece, más se multiplican los impactos en la mente del programador.

Pensemos en el ciclo de vida de un software y en su diseño. Al principio, desarrollar un producto es relativamente fácil: vemos nuestro objetivo, tenemos una idea de cómo alcanzarlo, nos aseguramos de que los clientes y las partes interesadas estén satisfechos, y si tenemos suerte logramos crear un equipo cohesionado en el producto tanto desde el punto de vista técnico como humano. Esto no siempre es seguro porque a veces basta un solo elemento discordante capaz de desestabilizar a todo un grupo. Una vez alcanzado este objetivo, cuando se necesita hacer una modificación, todos los miembros están alineados y conscientes de lo que se necesita lograr.

Cuando se sale de esta fase de euforia colectiva y se abandona la fase creativa, el equipo de desarrollo comienza a cambiar en comparación con su forma original. Al mismo tiempo, el cliente reduce el presupuesto, porque se entra en una fase de mantenimiento y si no se presta la debida atención, se comienza a perder el dominio del producto.

No, ciertamente no son las pruebas las que pueden gobernar el producto,  
sino las mentes que lo han ideado.

El cambio de proyecto es un proceso inevitable, a menos que te encuentres en un iglú en medio del polo, sin conexión a internet, y los pobres programadores que han realizado el proyecto no puedan enviar un currículum si no es atándolo a una foca (aunque siempre existe la posibilidad de que el programador se dedique a la pesca, su gran pasión adormecida por el código).

Cuando se cambia un componente del equipo, se pierde una parte del conocimiento del producto y se comienza a perder el dominio del mismo. En cascada, cada intervención se vuelve cada vez más costosa, larga y arriesgada.

### **¿El mantenimiento de un proyecto cambia las reglas?**

En un ciclo de vida normal de un producto, sucede que algunas partes están sujetas a actualizaciones frecuentes, pero muchas otras partes, por cuestiones de madurez, falta de demanda o estabilidad, no necesitan modificaciones. Hacer menos modificaciones disminuye el conocimiento del código por parte del propio programador que lo escribió. Esto significa que la persona que escribió esas líneas de código hace 2 años es normal que no las recuerde y necesite revisarlas para recordar los procesos que llevaron a su creación.

Cuanto más vertical es la modificación, más difícil es recordar la razón: no recuerdas por qué dentro de una conexión a un recurso externo pusiste un parámetro en lugar de otro, por qué usas una suite de cifrado en lugar de otra o tal vez tienes una rama abierta desde hace meses, esperando una verificación con un cliente sobre una funcionalidad urgente, que de repente se vuelve de poca importancia, pero ahora necesitas hacer el merge en una línea base con 400 push más y te preguntas si tiene más sentido empezar de nuevo o aplicar los cambios a un rebase.

En medio de este tipo de intervenciones están los trabajos diarios, las modificaciones a las funcionalidades, las correcciones de errores, las solicitudes de nuevas funcionalidades, las correcciones de nuevos errores que has introducido al corregir un error, el código que ha introducido un colega con poco conocimiento del producto, los cambios de comportamiento introducidos por el uso de una actualización o un nuevo componente que obliga a un cambio de código e introduce una anomalía indirecta.

Tú que escribiste ese código no lo recuerdas, entiendes que ante una serie de cambios la modificación a realizar es más grande de lo que se proyectaba, pero debes entender cómo hacerlo, porque tienes un recuerdo de los racionales por los cuales se hizo el código anterior y entiendes que no es una modificación trivial. Imagínate quien tiene que arreglar el mismo código y es la primera vez que pone las manos en esa parte del programa.

Como decía Heráclito:

Ningún hombre puede cruzar el mismo río dos veces,  
porque ni el hombre ni el río son los mismos

Esta frase se puede aplicar fácilmente al código: cada modificación, incluso la más pequeña, cambia el código y el contexto en el que se encuentra, que es diferente al contexto en el que fue creado, y también la persona es diferente a la persona que escribió ese código.

## Caso de uso de un conocido banco italiano

Por un momento, imagina ser un programador senior de un "banco cualquiera", donde te han dicho que debías proceder con una actualización del sistema y su firmware, que no causaría ningún daño y que se procedería a realizar la actualización en producción. Al haber visto este tipo de intervenciones varias veces en tu vida, lo primero que se te ocurre hacer es realizar una prueba, pero no existe un entorno de prueba que cubra exactamente las dimensiones de

producción, ya sea por dimensionamiento o por casos de uso. Desaconsejas la actualización masiva, pero el fabricante de la solución asegura que no habrá problemas, el departamento de seguridad te avisa que es necesario implementar la actualización antes de una fecha específica porque de lo contrario no se cumplirían las normas de la empresa y que de todas formas, en caso de problemas, se puede retroceder.

A regañadientes aceptas, también impulsado por el hecho de que "trabajamos con la metodología ágil y debemos estar listos para el cambio" y que "no podemos ser siempre negativos".

Las interrupciones en los accesos a los servicios en línea (como la aplicación, la aplicación de inversión, la banca por Internet y Smart Business) ocurrieron después de la instalación de una actualización del sistema operativo y su firmware correspondiente que provocó una situación de inestabilidad.  
Queremos disculparnos nuevamente y consideramos fundamental agradecerte por la comprensión demostrada.

Una "simple" actualización del sistema operativo y el firmware provocó una situación de inestabilidad durante 5 días.

Claramente, en este caso hipotético, el problema no fue la actualización, sino la falta de pruebas, la falta de un entorno de prueba que cubriera la producción, la falta de un rollback inmediato y la falta de un plan de recuperación ante desastres o la subestimación del riesgo.

## **¿Cuál es la percepción de la modificación fuera del proyecto?**

A menudo, quienes están fuera del proyecto tienen una percepción completamente diferente del desarrollo de software y presionan para realizar modificaciones lo antes posible.

## Lo necesito para ayer

Esta frase es un clásico, pero a menudo no se entiende que una modificación, incluso banal, puede acarrear una serie de problemas que van mucho más allá de la modificación en sí misma.

**Solo pedí cambiar una condición, ¿por qué Mario me está causando todos estos problemas?**

**Ahora asigno la tarea a Bruno, quien lo resuelve en 2 minutos**

Hay situaciones dentro de los proyectos que permiten alterar el código muy rápidamente, y otras situaciones en las que una modificación, incluso banal, puede requerir mucha reflexión: incluso si se trata de unas pocas líneas de código o incluso una sola condición adicional. Cada programador aborda la modificación de manera diferente, según su experiencia, el conocimiento del producto y el conocimiento del código. Cuantos más programadores experimentados haya en el producto, es más probable que la modificación se evalúe cuidadosamente, porque se es consciente de lo que una modificación puede implicar. Cuanto menos conozca el programador el producto, es más probable que la modificación se realice rápidamente.

## Conclusiones

Cada modificación, incluso la más banal, puede acarrear problemas de rendimiento, regresión, seguridad, mantenibilidad, comprensión del código, documentación, pruebas, implementación.

De la misma manera, las alteraciones externas al proyecto pueden crear los mismos problemas: he actualizado el sistema operativo, he actualizado un controlador, he actualizado el firmware y ahora nada funciona.

La presión que se ejerce sobre un programador aumenta progresivamente con el tiempo porque paralelamente aumenta la conciencia de todo lo que puede salir mal y de lo difícil que es resolver un problema una vez que se ha producido.

Esto puede llevar a situaciones en las que un programador senior se niega a hacer una modificación, porque sabe que los riesgos son demasiado altos, o porque sabe que la modificación requeriría demasiado tiempo y recursos para hacerse correctamente.

La próxima vez que mires a un programador que ha estado trabajando en el mismo código durante muchos años y no quiere modificarlo y te advierte sobre cada alteración interna o externa al proyecto, no pienses en él como alguien incompetente, sino más bien como una persona consciente y utiliza su consejo para evitar que tu proyecto se derrumbe y para sopesar adecuadamente los riesgos y beneficios de cada modificación, porque al fin y al cabo, como dijo Isaac Asimov:

Ninguna decisión sensata puede ser tomada sin considerar no solo el mundo tal como es ahora, sino cómo será

## La adopción de nuevos frameworks podría hacer fracasar tu proyecto



En el mundo del desarrollo de software, la atracción por las novedades siempre es fuerte. Es tentador poner las manos en un nuevo producto, tal vez el primero en resolver una serie de problemas que otros frameworks no pueden abordar.

Estos nuevos proyectos a menudo cuentan con una comunidad pequeña pero energética, prometen acelerar los tiempos de desarrollo y presentan innovaciones interesantes en comparación con los competidores. Todo esto parece muy prometedor, pero ¿es realmente la elección correcta para un proyecto a largo plazo?

Donald Knuth, uno de los padres de la informática moderna, comentaba su propio código de esta manera:

Presten atención a los errores en el código anterior; solo he demostrado que es correcto, no lo he probado.

Si esta frase hubiera sido escrita por un programador junior, podría parecer una broma. Pero dicha por Knuth, además de ser humorística, podría también señalar que esa parte del código no era testeable, sino solo sintácticamente correcta.

A veces es mucho más sencillo demostrar un concepto, describir su funcionamiento y documentarlo, en comparación con su aplicación real, que podría involucrar una serie de aspectos no considerados en la demostración.

Este concepto se puede parafrasear así: "No es seguro que una idea brillante funcione siempre en la realidad". Pensando en esto, no puedo evitar considerar el mundo de las blockchain: en teoría, herramientas excelentes, pero en la práctica a menudo criticadas por su rendimiento y uso de recursos.

Saliendo del mundo del software, pensemos en productos como el Segway que, a pesar de ser una idea brillante, no tuvo el éxito esperado. O en Google Glass, que a pesar de ser un producto innovador, nunca despegó.

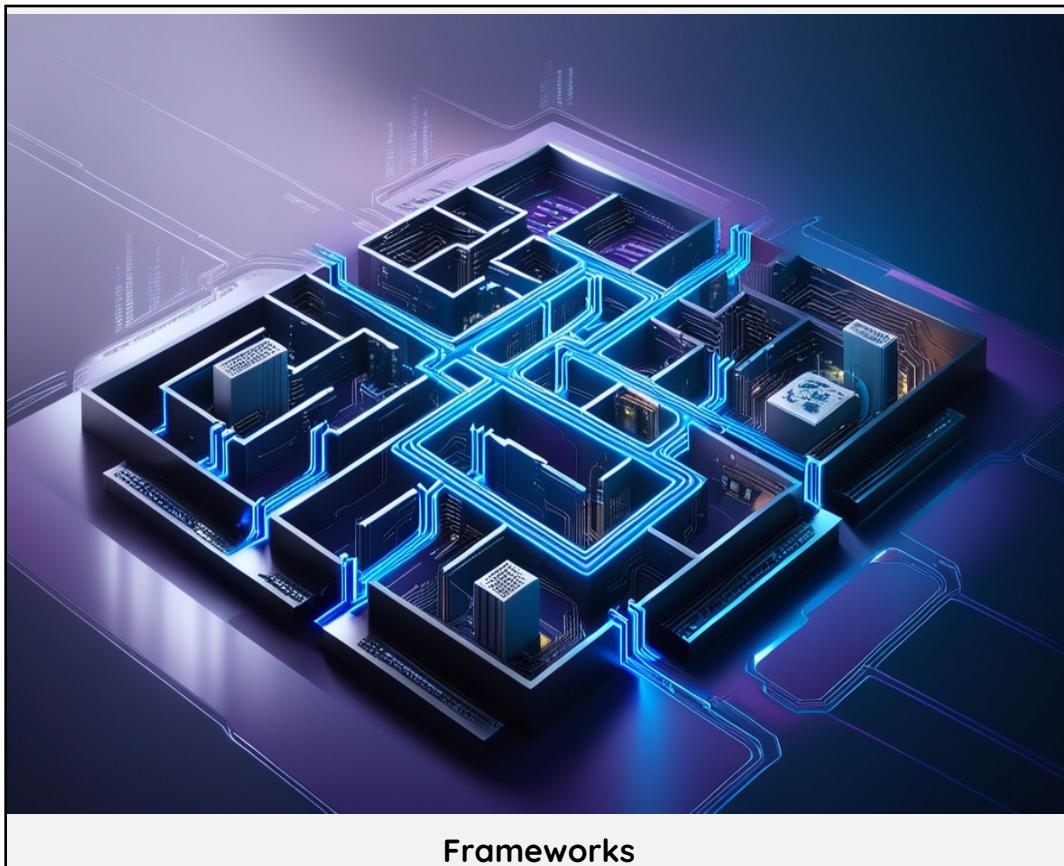
## ¿Y los frameworks?

Los frameworks y, más en general, las tecnologías de software, a menudo sufren de un hype inicial que entusiasma a los programadores, distrayéndolos de un análisis sólido de su eficacia a largo plazo. ¿Los conceptos utilizados en ese nuevo software son aplicables a todos los proyectos? ¿Son fácilmente integrables con otras tecnologías? ¿Hay empresas sólidas que invierten en esa tecnología? ¿La comunidad es activa y apoya el producto?

Los llamados "early adopters" a menudo se sienten atraídos por las novedades, pero frecuentemente no se dan cuenta de los riesgos que esta elección conlleva. Si todo va bien, han hecho una elección afortunada que ha llevado al éxito su software. Pero si el producto elegido no cumple con las promesas, se encuentran con un software que no funciona como debería y que requiere un trabajo de mantenimiento y refactorización mucho más pesado de lo previsto.

Para quienes desarrollan software desde hace más de 30 años, el nombre Visual Objects hará sonar algunas campanas: debía ser el nuevo Clipper y llevar a millones de programadores DOS al entorno Windows. Una idea fantástica, pero que no funcionó: demasiados problemas, demasiados errores, demasiados límites. En su momento, tenía detrás una empresa como Computer Associates, que podía permitirse invertir millones de dólares en un producto, pero a pesar de esto, nunca despegó.

Por eso, cuando se trata de elegir un framework o una tecnología para un proyecto, es importante considerar no solo sus características técnicas, sino también su sostenibilidad a largo plazo.



Frameworks

## El horizonte temporal: una dimensión crucial

Cada vez que evalúo un proyecto, trato de anticipar su ciclo de vida, o al menos lo que debería ser, basándome en la experiencia y en la información a mi disposición. La bola de cristal no existe y el futuro no es predecible, pero hay señales que permiten entender si un proyecto será de corta, media o larga duración.

Para comprender cuánto puede durar un proyecto, es importante considerar varios aspectos:

- **Objetivos a largo plazo:** ¿Cuáles son los objetivos del proyecto? ¿Es un producto que resuelve un solo problema o algo estructurado para durar años o solo unos pocos meses?

- **Estabilidad del mercado:** ¿El mercado en el que opera el proyecto es estable o está en rápida evolución? La efervescencia de un mercado puede llevar a cambios rápidos que requieren una mayor flexibilidad por parte del proyecto.
- **Competencia:** ¿Cuál es el nivel de competencia en el sector en el que opera el proyecto? Trabajar en una arena donde tal vez los otros actores provienen de empresas más grandes y estructuradas puede requerir una mayor atención a la calidad y la sostenibilidad del proyecto.
- **Tendencias del sector:** ¿Cuáles son las tendencias actuales en el sector en el que opera el proyecto? Algunos proyectos nacen ya viejos, porque se basan en tecnologías obsoletas o en modelos de negocio superados.
- **Recursos disponibles:** ¿Qué recursos están disponibles para el desarrollo y mantenimiento del proyecto? ¿Se trata de proyectos "one man show" o hay equipos estructurados detrás?

Estos son solo algunos de los indicadores posibles, pero basarse solo en el nombre y en lo que se promete a menudo es una subestimación del riesgo.

Cuando se utilizan frameworks o tecnologías jóvenes, es importante considerar que podrían no ser soportados a largo plazo o podrían no ser capaces de adaptarse a los cambios del mercado. Esto puede llevar a costos de mantenimiento elevados, problemas de compatibilidad y una mayor complejidad del código.

Por otro lado, también es necesario evaluar los proyectos que se pretenden realizar. Podrían ser proyectos nacidos y concluidos en el transcurso de un fin de semana: en este caso, la necesidad primaria es la resolución del problema. Otros podrían tener que durar unos pocos meses para satisfacer una necesidad específica. Finalmente, existen proyectos destinados a durar años, sobre los cuales una empresa pretende construir su base operativa: en estas situaciones, la elección del framework y la arquitectura del proyecto se vuelve crucial y no puede basarse solo en la moda del momento.

El horizonte temporal es una dimensión que todo arquitecto de software debería tener en cuenta, porque influye notablemente en las expectativas y las decisiones que se pueden tomar.

Para un proyecto "one-shot", se puede adoptar tranquilamente el framework CIRO, creado por un nómada de Tanzania, utilizado solo por su grupo de amigos, con un único lanzamiento hecho hace algunos años, si resuelve de manera asombrosa un problema crucial para el proyecto. Si en cambio debo realizar algo que debe durar más allá de la fecha de caducidad de un postre, tal vez debería moderar mi agresividad en la elección de los componentes de software a emplear y basarme en KPI diferentes.

## **El riesgo del abandono**

Me ha sucedido usar productos tanto "closed source" como "open source", cuyo creador desapareció en la nada, así como se disolvió la comunidad que formaba parte de ellos. Si esto ocurre con productos "closed", es ciertamente muy preocupante, pero no caigan en la ilusión de que todo es color de rosa si ocurre con productos "open". Una cosa es utilizar un producto, otra es desarrollarlo y conocerlo en cada uno de sus detalles.

La mente de quien diseña una solución suele estar atrapada entre las líneas de código del propio producto y está ciertamente más inmersa en sus lógicas internas que quien lo usa, que a menudo aprovecha solo una parte de sus potencialidades. No piensen entonces que "open" significa "hay un problema: lo resuelvo yo", porque no siempre es así.

Linus Torvalds, creador de Linux, tiene una opinión fuerte sobre este tema:

**El software es como el sexo: es mejor cuando es libre y gratuito.**

Aunque Torvalds utiliza esta cita para promover el software open source, es

importante recordar que "gratuito" no significa necesariamente "sin costos" y que la libertad de disponer de las fuentes de un proyecto no necesariamente significa poder mantenerlo y hacerlo evolucionar.

## **La importancia del análisis periódico**

Este es uno de los motivos por los que siempre deberíamos realizar un análisis de la vida de los productos que adoptamos. Un análisis a repetir periódicamente, que tenga en cuenta el contexto actual, la madurez del proyecto y sus perspectivas futuras.

Si en los proyectos cortos se puede ignorar este aspecto, en aquellos que superan los años de desarrollo, el problema se vuelve inminente: o se actualiza el producto y sus dependencias, o se encuentra uno en un callejón sin salida debido a los innumerables entrelazamientos de su propio código.

Un análisis periódico nos permite evaluar si el framework o la tecnología que estamos utilizando sigue siendo adecuada para nuestras necesidades, si está soportada activamente por la comunidad y si es capaz de adaptarse a los cambios del mercado.

## **El riesgo de los servicios no estándar**

Esto se aplica también a todos los problemas que pueden surgir de la adopción de un servicio sin estándar, que podría ser retirado del mercado o duplicar los precios de una temporada a otra.

Jeff Bezos tiene una visión interesante sobre cómo las empresas deberían abordar la tecnología:

**Hay que ser testarudos en la visión y flexibles en los detalles.**

Esta filosofía puede aplicarse también a la elección de frameworks, servicios y desarrollo de software: mientras los detalles técnicos pueden cambiar, los principios básicos como la escalabilidad, la mantenibilidad y la interoperabilidad deberían permanecer constantes.

Por este motivo, realizar un software fuertemente acoplado con un framework podría llevar a problemas de mantenimiento y escalabilidad en el futuro. Si el framework ya no es soportado o no es capaz de adaptarse a los cambios del mercado, podríamos encontrarnos en una situación en la que es necesario reescribir gran parte del código para adaptarlo a una nueva tecnología.

He visto productos tomar decisiones técnicas que, en teoría, parecían las mejores, y con el tiempo darse cuenta de que el acoplamiento de software que se había realizado era tan fuerte que cambiar una parte significaba tener que cambiar todo lo demás. Este problema también se puede evitar con un análisis correcto y una elección ponderada de los componentes utilizados, desacoplando uso de implementación y aceptando tener que cambiar una parte del software si es necesario.

## Conclusión

Cuando se trata de soluciones o frameworks jóvenes, el riesgo de desperdiciar tiempo y recursos es considerable. Si amas la incertidumbre y la duración del proyecto no es un factor que quieras considerar, eres libre de hacer cualquier elección y construir un producto lleno de preguntas sin respuesta.

En caso de errores, ten en cuenta que podría ser necesario empezar de cero, a menos que tu software sea lo suficientemente modular como para permitir la sustitución de un componente sin tener que reescribir todo lo demás.

La elección de un framework o una tecnología debería estar guiada no solo por el entusiasmo por la innovación, sino también por una evaluación ponderada de los riesgos y beneficios a largo plazo. La sostenibilidad, la mantenibilidad y la

escalabilidad deberían estar siempre en la cima de nuestras prioridades cuando tomamos decisiones arquitectónicas.

DRAFT DRAFT DRAFT

## El mito del desarrollador Full Stack: una realidad incómoda



El término "desarrollador Full Stack" ha experimentado una transformación radical que merece una reflexión cuidadosa.

Comencemos con la definición de lo que significa ser un "desarrollador Full Stack", tomando como referencia a ChatGPT:

Los "desarrolladores Full Stack" son desarrolladores de software que tienen habilidades tanto en el front-end como en el back-end del desarrollo web o de aplicaciones de software. En otras palabras, son capaces de trabajar en todas las partes de una aplicación o sitio web, desde el lado del cliente que se ejecuta en el navegador del usuario (front-end) hasta el lado del servidor (back-end) que maneja la lógica empresarial y el acceso a los datos.

## **Mi enfoque sobre el término desarrollador Full Stack**

En un momento de mi carrera, esta definición reflejaba perfectamente mi estado de ánimo, era lo que sentía que era y lo que deseaba que los demás percibieran al observarme. Orgulloso de este estatus, incluía esta calificación en mi currículum vitae, casi como cuando se obtiene una calificación excelente en la escuela y se desea compartirlo con la familia.

Con el tiempo, descubrí que el encanto de esta definición ha conquistado el corazón de muchas personas, principalmente debido a un pensamiento común: ser Full Stack es ser un programador completo y, para las empresas, contratar a un desarrollador Full Stack representa un valor agregado.

Si asigno un proyecto a este tipo de programador, él es capaz de manejar de manera autónoma todos los aspectos del desarrollo de software: ¡es como tener todo un equipo de desarrollo en una sola persona!

Con el tiempo, me di cuenta de que este enfoque podía tener sentido cuando los proyectos eran simples y las tecnologías eran pocas.

Hoy en día, el mundo ha cambiado, es mucho más complejo cubrir cada faceta de un proyecto. Ahora, ser un "desarrollador Full Stack" parece ser sinónimo de trabajar de manera mediocre en todos los aspectos de un proyecto: no se es un experto en el backend, ni en el frontend, ni en DevOps; se conocen conceptos de UX, UI, SEO, marketing, se sabe un poco de todo, pero nada en profundidad, o tal vez solo una parte.

Intentemos trasladar el concepto de "Full Stack" a su mecánico: su mecánico, cuando hay un problema en la carrocería, no la arregla a martillazos, sino que involucra a un chapista; cuando hay problemas de carburación, llama a un experto en carburadores; cuando hay un problema eléctrico, involucra a un electricista.

De la misma manera, si su médico sospecha de un problema en particular, lo envía a un especialista en esa área, quien realizará un examen exhaustivo para formular un diagnóstico y un tratamiento específicos.

Un programador "Full Stack" no: se le ha elegido para no tener que llamar a nadie, para resolver todo por sí mismo.

Este enfoque no es sostenible, o mejor dicho, no es abordable si se piensa que ser Full Stack es un punto de llegada y no un punto de partida. Muchas profesiones nos hacen comprender que es preferible tener un experto en un campo específico en lugar de un "todólogo" si queremos realizar un trabajo de calidad.

Los programadores experimentados saben lo importante que es especializarse y reconocen las horas perdidas buscando soluciones que un experto habría resuelto rápidamente.

En la cultura popular y a menudo cuando se habla con los clientes, hay una presunción de que un programador debe ser capaz de hacer todo a la perfección. Responder con frases como "no es mi campo, se necesitaría un experto en UI" se ve como una señal de debilidad, a pesar de ser una respuesta legítima,

profesional y honesta.

- "Soy un desarrollador Full Stack"
- "Ah, entonces eres un experto en todo?"
- "No, no exactamente"

## **El mercado laboral y el desarrollador Full Stack**

El mercado laboral ha comenzado a aprovechar la demanda de "desarrolladores Full Stack", respondiendo a esta necesidad de maneras a veces extrañas.

Están surgiendo figuras como los "desarrolladores Full Stack Junior", que se autodenominan "full stack" pero agregan el término "junior" para resaltar su falta de experiencia. A menudo, este título sigue cursos intensivos del tipo "conviértete en full stack en 6 meses".

No nos damos cuenta, pero nos enfrentamos a un problema serio. Por un lado, estamos inculcando en la mente de los jóvenes programadores la idea de que pueden cubrir de manera autónoma y fácil toda la pila tecnológica, para luego enfrentarlos a la cruda realidad: en el primer proyecto complejo que encuentran, se dan cuenta de que no son capaces de manejarlo solos.

Sin embargo, no todo es culpa de estos nuevos programadores. También existe un problema relacionado con algunas empresas que no tienen la capacidad de evaluar adecuadamente el nivel real de competencia de las personas y asignan roles de responsabilidad basándose en la poca información que tienen.

Responder brillantemente a una entrevista técnica es una cosa, pero como sabiamente sostiene Linus Torvalds:

**Hablar es barato. Muéstrame el código.**

## **La calidad del código y la evaluación del programador**

El código, el enfoque para resolver problemas y la forma en que se enfrentan las dificultades son los aspectos que distinguen a un programador de otro, incluso si ambos se autodenominan "desarrolladores Full Stack".

Desafortunadamente, evaluar cuánto una persona es realmente capaz de producir no es para nada sencillo. He visto programadores producir grandes cantidades de código, pero de baja calidad, y programadores que producían poco, pero de una calidad muy alta.

Para citar a Antoine de Saint-Exupéry:

**La perfección se alcanza no cuando no hay nada más que agregar, sino cuando no hay nada más que quitar.**

Evaluar el verdadero valor de un programador es una habilidad muy difícil de adquirir, pero es una competencia fundamental para un buen gerente. No es suficiente evaluar la cantidad de líneas de código producidas, ni el número de proyectos completados o las tecnologías conocidas, especialmente en una era en la que las tecnologías cambian rápidamente y nos ayudan a escribir cada vez más código en menos tiempo.

La cantidad de código no es un indicador de calidad, incluso si funciona perfectamente y está cubierto por numerosas pruebas funcionales. El código no debe evaluarse por kilo, sino por su calidad, mantenibilidad, escalabilidad y legibilidad.

¿Quién puede evaluar la calidad del código? No es fácil responder a esta pregunta.

La respuesta más sencilla podría ser: otro desarrollador Full Stack, pero esa no es la respuesta correcta.

No lo es porque un desarrollador Full Stack no es un experto en todo, sino que

debería ser experto principalmente en su área de aplicación específica. Por lo tanto, la evaluación del valor de una persona por parte de un desarrollador Full Stack puede estar influenciada por una serie de sesgos debido a sus experiencias personales, conocimientos previos y habilidades adquiridas.

Entonces, ¿puede ser un departamento de recursos humanos el encargado de evaluar la calidad del código? ¿O un gerente? ¿Un colega? ¿Un cliente? La respuesta correcta es: todos ellos pueden contribuir.

En este caso también, la evaluación se basará en la suma de las experiencias de las diversas personas involucradas, que pueden tener opiniones diferentes y decretar que el mismo desarrollador Full Stack es un genio en una pila tecnológica como LAMP, pero un incompetente en otra como MEAN.

## **¿Y si trabajáramos en la "actitud de resolución de problemas"?**

Hemos asimilado que "desarrollador Full Stack" es un término no calificativo, interpretable de manera diferente según el contexto y, sobre todo, muy costoso en términos de actualizaciones tecnológicas necesarias.

Entonces, intentemos trabajar en un concepto diferente: la "actitud de resolución de problemas".

Un programador que posee una sólida "actitud de resolución de problemas" es capaz de enfrentar cualquier problema, incluso si nunca ha enfrentado uno similar en el pasado. Es capaz de analizar el problema, descomponerlo en partes más pequeñas, encontrar una solución, implementarla y probarla con éxito.

Este tipo de programador no se desanima ante las dificultades, sino que las enfrenta con determinación, adaptándose y aprendiendo en el proceso. Esta mentalidad flexible y analítica es fundamental para enfrentar los desafíos siempre nuevos que se presentan en el desarrollo de software.

Un programador con una sólida "actitud de resolución de problemas" no tiene miedo de pedir ayuda, de admitir "no sé" o "no puedo hacerlo". De hecho, es capaz de trabajar de manera efectiva en equipo, compartiendo sus conocimientos, aprendiendo de aquellos que saben más y enseñando a aquellos que saben menos.

En este contexto, una gran cantidad de experiencias en diferentes contextos y tecnologías, en proyectos internacionales con tecnologías heterogéneas, es sin duda un valor agregado, mucho más que ser un desarrollador Full Stack. Este último, de hecho, no se especializa en la resolución de problemas, sino en el conocimiento profundo de su pila tecnológica específica, resultando fuera de contexto como un jugador de bolos en un juego de briscola.

Ser capaz de resolver problemas incluso en contextos no familiares es un valioso activo dentro de un equipo. Siempre habrá tiempo después para perfeccionar la solución al ponerla en manos de un experto en el producto, pero la capacidad de resolver problemas de manera autónoma es una habilidad invaluable.

Un programador con una mentalidad flexible, analítica y colaborativa, combinada con una amplia gama de experiencias diversas, será capaz de enfrentar con éxito los desafíos más complejos, adaptándose y creciendo constantemente.

## Conclusiones

Mi consejo es evitar autodenominarse "desarrollador Full Stack".

Este término está demasiado inflado y, cuando se escribe en el currículum vitae de un programador junior, no agrega ningún valor, excepto la hilaridad de los programadores más experimentados que lo leen.

Es mucho mejor identificar con precisión las tecnologías en las que uno es experto y aquellas en las que se tiene menos experiencia. Esto es válido para todos los programadores: puedes ser un mago de Angular, haber escrito API REST en Node,

pero si un experto en backend analiza tu código, probablemente encontrará deficiencias.

Al exponer claramente las habilidades y experiencias propias, será el interlocutor quien evalúe el valor real, un aspecto que vale mucho más que una etiqueta genérica de "desarrollador Full Stack".

Es necesario concentrarse en construir una sólida "actitud de resolución de problemas", adquirir experiencia en diferentes contextos tecnológicos, salir de la zona de confort y promover un enfoque colaborativo en el trabajo. Estas cualidades serán mucho más apreciadas que una autodeclaración como "Full Stack" que corre el riesgo de ser engañosa o excesivamente generalista.

## **"Veamos quién la tiene más grande: hagamos revisión de código"**



La revisión de código es uno de los momentos más delicados e importantes de un proyecto de software. Cualquier proyecto de software que hayamos desarrollado o desarrollemos en el futuro ha sido o será sometido a una revisión de código.

No es imaginable un código escrito por cualquiera que no sufra cambios con el tiempo. Aunque esto pueda ser cierto para partes individuales de un proyecto, si miramos la totalidad de un proyecto de software, la revisión de código se convierte en un paso significativo entre quedarse pequeño y jugar con muñecas, o salir a la luz y convertirse en adultos.

Este paso es, por lo tanto, fundamental para garantizar la calidad del código, para evitar que errores puedan comprometer el funcionamiento del software y para ampliar el conocimiento del código a todo el equipo. Sí, porque hacer evolucionar al equipo es un aspecto más importante que sublimar el propio ego al mirar unas pocas líneas que solo quien las escribió puede entender.

## **La revisión de código molesta**

Sin embargo, existe un problema de fondo: a los programadores la revisión de código les molesta, no les gusta, la consideran superflua o incorrecta si la hacen otros y creen que debe ser hecha y decidida solo por ellos.

Para algunos, es como perder un hijo:

Ese código estaba escrito de esa manera por una razón muy precisa: Mario no quería que nadie lo tocara, pero ahora que se ha ido, ya no se entiende nada.

Un buen momento para hacer revisión de código es cuando se corrige un error. Sin embargo, es necesario desmantelar el enfoque que a menudo adoptan los programadores, que tienden a ser quirúrgicos y a resolver la única notificación, sin aprovechar el pensamiento holístico.

**Me han informado que al hacer clic en un botón hay un error, me aseguro de que no lo haya.**

En realidad, el problema podría ser mucho más profundo y requerir una reflexión más amplia. Deberíamos plantearnos el problema de por qué se hizo clic en ese botón, si fue clicado por un usuario, si fue clicado por un usuario que no debería haberlo hecho, si fue clicado por un usuario que no debería haberlo hecho en ese momento, si fue clicado por un usuario que no debería haberlo hecho en ese momento y en ese contexto.

Si queremos ampliar el pensamiento, podríamos preguntarnos si el problema no se extiende a otros botones, si y cómo se informa, si tiene sentido tener ese botón o esos botones que realizan esa operación, si la operación que realiza ese botón es necesaria, si la operación que realiza ese botón es necesaria en ese contexto.

Ampliar el contexto, sin embargo, significa gastar tiempo, emplear recursos, cuestionar el propio trabajo y el de los demás, significa cuestionar el propio ego y el de los demás, pero al hacerlo se mejora el código, se mejora el sistema, se mejora el equipo.

Por lo tanto, es necesario no mirar la única notificación que se nos hace, sino tratar de entender el contexto en el que se hizo y si hay otras partes del código que podrían mejorarse.

El pensamiento holístico, incluso en el código, es importante: si no entendemos que una sola modificación tiene un impacto en todo el sistema, no podemos entender el sistema, si no entendemos el sistema, no podemos mejorarlo.

Como decía Platón:

**El refactoring es el arte de quitar el mal y añadir el bien al código.**

Quizás el autor no era Platón, pero el concepto es claro: el código debe mejorarse, debe hacerse más legible, más mantenible, más comprobable.

## **El código no utilizado**

Si has trabajado en el mismo proyecto durante más de un lustro, la frase:

Ese código lo escribimos para un "Cliente importante", que ya no es nuestro cliente, pero no lo quitamos porque "podría" servir

es casi un mantra que estoy seguro has escuchado al menos una vez.

Estratificar pensamientos de este tipo a lo largo del tiempo lleva a tener un código que no se entiende, que no se sabe qué hace, que no se sabe por qué lo hace y que no se sabe cómo lo hace: las especificaciones estaban entrelazadas con las necesidades del cliente, con las exigencias del equipo, con las decisiones técnicas del momento.

Mientras tanto, el código ha cambiado, el equipo ha cambiado, el cliente ha cambiado, las necesidades han cambiado, las decisiones técnicas han cambiado: es justo que el código cambie o sea eliminado del proyecto. Mantener código no utilizado es costoso y con el tiempo representa una deuda técnica innecesaria.

Hay empresas que han entendido lo costoso e inútil que es mantener proyectos que ya no se utilizan: el código es un costo, no un valor, si no se utiliza.

Google, por ejemplo, tiene una rica historia de proyectos que han sido abandonados. Puedes encontrar rastros de ellos en Wikipedia:

[https://en.wikipedia.org/wiki/Category:Discontinued\\_Google\\_services](https://en.wikipedia.org/wiki/Category:Discontinued_Google_services)

## **Yo uso TAB, quien usa espacios no es un buen programador**

En un episodio de "Silicon Valley", el protagonista Richard descubre que su novia Winnie utiliza espacios en lugar de TAB para indentar el código y la deja: es una exageración cinematográfica, pero conozco personas que nunca aceptarían trabajar con alguien que usa espacios en lugar de TAB y viceversa.

Desmantelar este hábito, y lo dice alguien que usa espacios desde que puso el dedo en el teclado, es difícil, pero necesario: el código debe ser uniforme, debe ser coherente, debe ser legible. Tener estilos diferentes dentro del equipo es contraproducente, ralentiza el trabajo, aumenta la posibilidad de errores, dificulta el mantenimiento.

Dentro de una revisión de código es necesario discutir también estas cosas: no se trata de imponer un estilo propio, sino de encontrar un compromiso que pueda ser aceptado por todos y que, posiblemente, no sea una carga para nadie.

Una vez alcanzado un acuerdo, es importante respetarlo: si se decide usar espacios en lugar de TAB, no se puede retroceder.

Sin embargo, es algo difícil de aceptar y de implementar, tanto porque algunos programadores consideran el código como un hijo, que no se puede cambiar, como por el hecho de que, incluso imponiendo un estilo uniforme, tarde o temprano alguien no entenderá la especificación, otros configurarán el IDE de manera incorrecta o perderán las configuraciones en la primera actualización.

En estas situaciones, donde no es el propio lenguaje el que nos ayuda a mantener un estilo uniforme (sí, hay lenguajes que lo hacen por nosotros de manera absolutamente dictatorial), es importante utilizar herramientas que nos permitan automatizar lo más posible la formateo del código.

Dentro de uno de los proyectos más grandes que sigo, se decidió adoptar OpenRewrite, una herramienta que reformatea el código de manera automática, uniforme y coherente, reduciendo las diferencias entre diferentes estilos de

programación: uniformar, además de hacer el código más coherente, acelera la incorporación de nuevos miembros al equipo y el traspaso de un componente de un equipo a otro.

## **Resistencia al cambio**

Aunque la mayoría de los programadores está de acuerdo en la necesidad de una revisión de código, a menudo se niegan a aceptar los cambios propuestos, un poco por pereza, un poco por orgullo, un poco por costumbre.

Si has trabajado durante años de la misma manera, y funciona, ¿por qué cambiar?

### **Mi código nace perfecto**

Desafortunadamente no: no es así. Dentro de un equipo, el aspecto que debe prevalecer es la lectura del código por parte de todos, debe prevalecer la facilidad de incorporación de un nuevo recurso, debe prevalecer la simplicidad de modificación incluso después de meses de no tocar algunas líneas.

Por lo tanto, no importa si el IDE que usas formatea el código de cierta manera, no importa si siempre te ha gustado declarar variables con "\_", adherirse a un estándar difundido, espontáneamente o a través de automatizaciones, es una ventaja, incluso si al principio puede parecer una carga.

## **Código limpio**

Uno de los objetivos que debería tener una revisión de código es escribir código limpio, legible y mantenible.

A veces me he encontrado ante la elección de mantener un código que funcionaba, o desmontarlo porque, aunque superaba de manera excelente cualquier prueba funcional, estaba escrito de manera oscura, difícil de leer y de

mantener, y cada vez que ponía a una persona frente a ese código, su expresión siempre era la misma:

### No entiendo qué hace

A veces creemos que usar la última sintaxis propuesta por un lenguaje es la solución a todos nuestros problemas, pero no siempre es así. A veces los nuevos constructos se basan en conceptos difíciles de comprender y de explicar, a veces son solo caramelos para hacernos sentir más hábiles: "sugar code" como lo llaman algunos.

No siempre el código más compacto y conciso es el mejor: a veces es mejor escribir 10 líneas de código que hacen lo que deben de manera clara y legible, en lugar de una sola línea que hace lo mismo, pero que nadie entiende, para no ganar nada a nivel de rendimiento y funcionalidad.

### Automatizar lo posible

Curiosamente, los programadores aceptan de buen grado algo que es propuesto por un programa, en comparación con lo que es propuesto por alguien dentro del equipo.

Si en tu equipo también hay el mismo clima, puedes proponer la introducción, dentro de la línea de proyecto (esperando que tengas una), de procesos de normalización de código y análisis estático.

Los primeros servirán para disminuir la diferencia de estilo entre un programador y otro, los otros servirán para sugerir modificaciones, para hacernos entender dónde están ocultos los errores, dónde faltan pruebas, dónde la complejidad cognitiva es demasiado alta o dónde podemos reducir el código, sin perder claridad o eficacia.

### Pero a mí no me interesa

De vez en cuando paso mi tiempo libre estudiando proyectos de otros programadores, hago correr algún analizador y trato de corregir un poco de código: es un buen entrenamiento y me permite ver cómo otros abordan los problemas o subestiman las consecuencias de su código.

Durante estas incursiones, me he encontrado con enfoques completamente diferentes por parte de programadores individuales o equipos de programadores a los que presentaba mis Pull Requests.

De todos los proyectos que he analizado, me gustaría hablar de dos que me han impresionado particularmente por cómo el equipo gestionó mi contribución.

La primera Pull Request se refiere al JDK de OpenJDK. Me di cuenta de que, por distracción, 82 fuentes contenían dobles ":" al final de una línea.

Entiendes que no se trata de algo importante, podemos incluso decir que se trata de algo insignificante, pero decidí hacer una Pull Request para corregir el problema y ver cómo se comportaría el equipo de desarrollo.

Puedes encontrar la PR en esta dirección:

<https://github.com/openjdk/jdk/commit/ccad39237ab860c5c5579537f740177e3f1adcc9>

El enfoque, en mi opinión, fue interesante: primero se involucraron en el análisis de la modificación 8 personas, siendo las modificaciones horizontales a muchos paquetes de Java.

Después de una discusión horizontal de varios mantenedores y haber constatado que el problema era real, se abrió un issue:

<https://bugs.openjdk.org/browse/JDK-8282657>

También se discutió una posible modificación a las herramientas de construcción, que ya eliminaban los espacios al final de la línea, introduciendo la eliminación de los caracteres dobles ";".

Este enfoque denota un equipo que analiza cada modificación individual, que discute y que trata de entender si la modificación propuesta es realmente útil o si es solo un capricho de alguien.

En este caso, entendieron que se trataba de una modificación trivial, pero orientada a la limpieza del código, y por eso fue aceptada.

Un segundo caso, sin embargo, involucra una herramienta de seguridad realizada por un único mantenedor, también puesta en GitHub.

En ese caso, había propuesto una PR mucho más seria. Había código redundante, se utilizaban clases sincronizadas en procesos en los que no se necesitaba sincronización, algunos recursos se abrían sin un cierre explícito y así sucesivamente.

Por lo tanto, se trataba de una modificación orientada a mejorar la calidad del código y a reducir la posibilidad de errores, no algo trivial como la eliminación de un carácter.

Además, el paso a objetos no sincronizados llevaba a un aumento de rendimiento en torno al 20%: todo esto sin desnaturalizar el código, sino solo utilizando los constructos correctos y, a igualdad de interfaces, las clases correctas.

En este caso, estaba convencido de que no habría problemas para insertar este código en el proyecto, pero la respuesta fue negativa.

El mantenedor respondió que no tenía intención de integrar en su código las sugerencias que derivaban de un analizador sintáctico y que podía usarlas en mi fork y él nunca las integraría.

**Si no te gusta mi código, haz tu fork y modifícalo como quieras**

Esta es la clásica respuesta de quien no quiere cambiar, de quien no quiere aceptar que su código pueda ser mejorado, de quien no quiere aceptar que su código pueda ser cambiado por otros, a pesar de que las modificaciones puedan mejorar la legibilidad y traigan mejoras tangibles.

## **Conclusión**

Incluso cuando todo funciona, revisar el código es importante: permite mejorar el proyecto, mejorar el equipo y reducir lo que son las deudas técnicas.

Disminuir la deuda técnica, tener el coraje de desmontar incluso código que funciona, eliminar código que ya no se usa, escuchar a los revisores estáticos, uniformar el estilo de programación, automatizar lo más posible la formateo del código, son todos pasos que pueden mejorar el código y el equipo.

No tengas miedo de cuestionar lo que se escribió en el pasado y amplía la visión cuando debas poner manos al código: no mires solo la modificación individual, sino trata de entender el contexto en el que se hizo y si hay otras partes del código que podrían mejorarse.

Un producto mejor pasa por muchos pequeños pasos y no se dice que, a pesar de tener cientos de pruebas funcionando, el código sea perfecto: la revisión de código es un paso fundamental para garantizar la calidad del código y para evitar que errores puedan comprometer el funcionamiento del software.

## **Dimensión Profesional**

El crecimiento profesional de un desarrollador no se mide solo por la cantidad de lenguajes aprendidos o frameworks utilizados, sino por su capacidad de comprender el contexto humano, organizativo y económico en el que el software cobra vida. Esta sección aborda el lado menos “romántico” del oficio: dinámicas de estatus, motivaciones económicas, percepción del valor individual y fragilidad de las relaciones profesionales.

Analizamos cómo la cultura empresarial, el mercado laboral, los incentivos estructurales y los sesgos personales influyen en decisiones, comportamientos y oportunidades. Hablaremos del reconocimiento (o su ausencia), la inflación identitaria detrás del teclado, el mercenarismo funcional, la palanca económica de la remuneración y la miopía organizativa en la retención (o pérdida) del talento.

El objetivo es la conciencia: entender los mecanismos que determinan cómo somos evaluados, por qué ciertos perfiles son retenidos (o descartados) y cómo mantener integridad profesional mientras navegamos expectativas, egos, asimetrías de información y paradojas de la cultura de ingeniería moderna.

Esta dimensión no requiere solo habilidad técnica, sino lucidez, ética pragmática y una visión sistémica de la propia trayectoria. Comprenderla significa transformar el trabajo de simple entrega de código a la construcción intencional de una identidad profesional sólida y antifrágil en el tiempo.

## El "celodurismo" de los programadores



En el mundo de la programación, existe un fenómeno curioso que podríamos definir como "celodurismo". Este término describe una actitud de obstinada resistencia a la evolución de las herramientas y prácticas de desarrollo de software.

En tiempos pasados, cuando los recursos eran escasos y valiosos, los programadores debían ser increíblemente ingeniosos para exprimir cada gota de potencia de sus sistemas. Esta necesidad forjó hábitos y mentalidades que, en algunos casos, han sobrevivido mucho más allá de su utilidad práctica, transformándose en una especie de folclor profesional.

### 640K ought to be enough for anybody

Esta célebre frase atribuida a Bill Gates, aunque él ha negado su autoría, representa bien la actitud de muchos "celoduristas" hacia la innovación tecnológica. Para estos programadores, la idea de utilizar herramientas modernas como IDE avanzados, depuradores gráficos o asistentes de IA parece casi una herejía, una violación de los principios fundamentales de la programación "verdadera", un síntoma de debilidad e incapacidad.

## Las raíces históricas

En los albores de la informática, cuando las computadoras se alimentaban a base de tarjetas perforadas, programar significaba trabajar con sistemas dotados de memoria y procesadores limitados. Los programadores de la época debían ser verdaderos artistas de la optimización, capaces de escribir código que fuera a la vez funcional y extremadamente eficiente.

Esta era produjo obras maestras de ingenio y prácticas que aún sobreviven en entornos con recursos limitados: ¿alguna vez has intentado programar para microcontroladores o sistemas embebidos? Aquí, cada byte de memoria y cada ciclo de reloj cuentan, y el arte de la optimización sigue gozando de buena salud, aunque ya se empiezan a ver algunas grietas en el horizonte.

Con el paso del tiempo y la evolución del hardware, muchas de estas limitaciones han desaparecido, pero la ética del "hacer más con menos" ha quedado profundamente arraigada en la cultura de la programación, a veces transformándose en una actitud de resistencia hacia herramientas y prácticas que podrían simplificar y acelerar el trabajo de desarrollo.

"El código más eficiente es el que no tienes que escribir"

## El manifiesto del celodurismo

Para entender qué tipo de programadores sois, aquí tenéis una pequeña prueba para evaluar vuestro grado de "pureza".

Intentad determinar cuánto estáis a favor o en contra de estas afirmaciones:

### 1. El IDE no sirve para nada

Un programador debe ser espartano: cuantos menos herramientas use, más competente es.

"Para escribir código me basta con Notepad (o Vi) y la CLI"

Esta afirmación se pronuncia a menudo con una mezcla de orgullo y nostalgia. Quien la sostiene parece querer comunicar: "Soy un verdadero programador, no necesito ayudas". Esta posición ignora deliberadamente las ventajas que ofrecen los IDE modernos.

Personalmente, he visto programadores usar comandos como:

copy con: pippo.prg

con el mismo orgullo de quien obtiene una excelente calificación en la escuela.

No subestimemos, sin embargo, las ventajas de los IDE modernos y, como primera cosa, quitémonos de la cabeza que los IDE son simples editores de texto con algunos adornos. Son herramientas poderosas que integran numerosas funcionalidades para mejorar la productividad y la calidad del código:

- Depuración: Puntos de interrupción, inspección y alteración de variables en tiempo real, ejecución paso a paso del código, pila de llamadas y mucho más.
- Refactorización: Renombrar variables o funciones en todo el proyecto con un solo clic, extrayendo métodos o reorganizando el código de manera segura.
- Análisis estático: Identificación de posibles errores, violaciones de estilo o patrones problemáticos antes de la ejecución.
- Integración con sistemas de versionado: Gestión de ramas, commits y merges directamente desde la interfaz del IDE.
- Soporte para frameworks y bibliotecas: Autocompletado, no solo de las bibliotecas base, sino también sugerencias específicas para los frameworks utilizados en el proyecto.

Estos son solo algunos de los motivos por los cuales, cuando entro en un editor inventado en los años 70, me siento como si me hubieran quitado un brazo.

Para ser intelectualmente honestos, no podemos negar que, en algunas situaciones, un IDE podría ser excesivo, como cuando se necesita hacer una modificación rápida o cuando se trabaja en sistemas remotos con recursos limitados. En estos casos, editores minimalistas o cualquier cosa que abra una consola textual podrían ser la mejor opción.

Para el trabajo diario y para proyectos de cierta complejidad, rechazar categóricamente el uso de un IDE moderno significa privarse de herramientas que pueden mejorar significativamente la calidad del código y la productividad del programador y de todo el equipo (sí, porque el código no es solo tuyo, sino de todos los que trabajan en él).

## **2. La CLI es la solución a todos los problemas**

No nos engañemos, la CLI tiene un encanto innegable. Años de películas con hackers inclinados sobre teclados mecánicos y pantallas negras con letras verdes han formado generaciones de programadores. ¿Qué hay más poderoso que controlar una computadora escribiendo comandos textuales?

Este enfoque, aunque tiene el encanto del primer amor del instituto, tiene una serie de limitaciones que a menudo se pasan por alto:

- Curva de aprendizaje: Memorizar unos pocos comandos es fácil, memorizar decenas o cientos de comandos, con todas sus opciones, puede ser desalentador para los novatos y para quienes no usan la CLI diariamente.
- Visualización de datos: Algunas informaciones son simplemente más fáciles de comprender cuando se presentan gráficamente.
- Operaciones complejas: Ciertas actividades, como la gestión de ramas en un repositorio Git, pueden volverse mucho más intuitivas con una representación visual.

Git es un excelente ejemplo de cómo CLI y GUI pueden coexistir y complementarse mutuamente. Mientras que la CLI de Git ofrece un control granular y la posibilidad de automatizar operaciones a través de scripts, herramientas GUI como las integraciones en VSCode pueden hacer más accesibles operaciones complejas como:

- Visualizar el historial de commits con un gráfico de ramas
- Realizar rebase interactivos
- Resolver conflictos de merge con herramientas de diff visual

Como siempre, la verdad está en el medio y el enfoque más sabio es dominar ambas herramientas. Usar la CLI para operaciones rápidas y para crear scripts

automatizados, pero no dudar en pasar a una GUI cuando esta puede ofrecer una visión más clara o un flujo de trabajo más eficiente.

### **3. No usamos Windows porque los programadores usan Linux (o MacOS)**

Ya nouento las noches que he pasado leyendo y comentando las "guerras de religión" de los sistemas operativos.

"Winzoz" no funciona

Será bonito Linux que tiene 200 versiones y todas incompatibles  
Dejadlo, con MacOS todo funciona

Mira que Apple te hace pagar el doble de lo que vale ese ordenador

Se podría seguir indefinidamente, creando flames llenos de odio y cargados de ignorancia, pero la realidad es que cada sistema operativo tiene sus pros y contras, y es erróneo sostener que uno es superior a los otros de manera absoluta.

Un programador debería estar por encima de estas charlas de bar. Esto no limita su libertad de sentirse cómodo en un sistema en lugar de otro, pero es absolutamente contraproducente ponerse anteojeras e ignorar la existencia de otros sistemas operativos además del preferido.

Reflexionemos sobre el hecho de trabajar diariamente en plataformas diferentes y de las ventajas que este enfoque puede traer:

- Cross-platform: Desarrollar y probar en múltiples plataformas asegura que el software funcione correctamente para una amplia base de usuarios. "Write once, run anywhere" es un objetivo importante para muchas aplicaciones y, aunque de base los lenguajes aseguran que esto pueda ser cierto, cada programador pone de su parte para que no lo sea.
- Flexibilidad: Muchas empresas utilizan entornos mixtos y la capacidad de

adaptarse es una ventaja competitiva.

- Comprensión de las diferencias: Trabajar en diferentes sistemas ayuda a comprender mejor las peculiaridades de cada uno, mejorando la capacidad de depuración y optimización.

En lugar de aferrarse dogmáticamente a un solo sistema operativo, un enfoque más productivo es elegir la herramienta adecuada para el trabajo adecuado, manteniendo la flexibilidad de moverse entre diferentes plataformas cuando sea necesario.

#### **4. El depurador es para los débiles**

Desmintamos una idea

Un "verdadero programador" escribe código perfecto al primer intento

Esta historia circula en el mundo de la programación desde hace años. Más o menos todos los programadores famosos se han cosido esta imagen, y cada uno de nosotros está listo para contarla al final de la noche cuando las cervezas ya se han acabado.

La realidad es que la depuración y las pruebas son una parte esencial e inevitable del proceso de desarrollo de software y esta mentalidad de resistencia al uso de herramientas de depuración, viéndolas como una especie de "muleta", es algo que debe superarse.

Ya nouento las veces que he tomado un software considerado "concluido", lleno de pruebas funcionales, y poco a poco han surgido problemas, casos de uso no cubiertos, problemas de análisis y diseño: no, la ilusión de que la primera versión de un programa sea capaz de generar un software perfecto y que las pruebas sean suficientes para entender qué funciona o no funciona es solo una ilusión.

En estas situaciones: logs, depuración y nuevas pruebas se vuelven necesarios para entender qué no funciona y cómo resolverlo.

## 5. El código está todo en mi cabeza

El análisis, la compartición del conocimiento y la documentación son aspectos a menudo descuidados en la programación.

### Lo tengo todo en la cabeza

No existe una frase menos productiva que esta, que atrofia cualquier tipo de discusión y colaboración.

Existe una narrativa romántica del programador como genio solitario, capaz de mantener sistemas complejos enteros en su mente y de ver su software como Neo veía Matrix.

Este enfoque tiene un error de base: la mente humana, aunque maravillosa, tiene límites en la cantidad de información que puede mantener.

No todos somos Dennis Nedry, el programador de Jurassic Park que sabía todo el código del parque de memoria y, aunque lo fuéramos, recordemos qué le pasó.

Olvidar el pasado es una forma de protección que activa nuestro cerebro para evitar volverse loco. Este es el motivo por el cual es importante documentar el código, compartir el conocimiento y trabajar en equipo.

Pero incluso si la propia mente tuviera una capacidad infinita, hay otros motivos por los cuales el "código en mi cabeza" es un problema:

- Colaboración: Si el funcionamiento del código es claro solo en la mente de quien lo escribió, se vuelve difícil para otros contribuir o mantener el proyecto.
- Propensión a errores: Sin una documentación clara o comentarios en el código,

es fácil olvidar detalles importantes o hacer suposiciones erróneas.

- Onboarding complicado: Los nuevos miembros del equipo tendrán dificultades para comprender y contribuir al proyecto.

Es necesario que el "celodurista" se dé cuenta de que el código es un producto colaborativo y no compartir información no es el mejor enfoque para mantener su trabajo, sino la mejor manera de perderlo.

Animemos a estos programadores a documentar, escribir wikis, hacer revisiones de código, usar diagramas y esquemas, para compartir el conocimiento.

Un enfoque que valore la documentación y la compartición del conocimiento no solo hace el proyecto más robusto y mantenible, sino que también contribuye al crecimiento profesional de todo el equipo.

## **6. Los programadores no usan ChatGPT**

Seamos sinceros: las inteligencias artificiales son todo, menos inteligentes.

**Mira cuántos errores comete ChatGPT, no puede reemplazar a un programador**

Las alucinaciones, pero sobre todo el uso incorrecto y superficial de las IA, llevan a muchos programadores a pensar que son herramientas inutilizables.

No, las IA no son motores de búsqueda, son herramientas de agregación lingüística, que pueden aprender nuestro contexto de trabajo, y como tales deben ser usadas.

A estas teorías se superponen cada vez más las teorías de que en el futuro las IA reemplazarán a los programadores.

El programador "duro y puro" no usa estos instrumentos, porque no los necesita,

porque no funcionan, porque "yo soy mejor".

Todo cierto, pero solo en parte. La realidad es que las IA son herramientas que pueden ayudar a los programadores a escribir código más rápido y con menos errores, pero no basta una hora para llegar a este resultado, se necesitan semanas de uso para entender mejor cómo usar esta herramienta e identificar rápidamente sus pros y contras.

La IA debe ser vista como una herramienta de potenciación, como la evolución del autocompletado de los IDE modernos, pero capaz de extender el completado a un contexto más amplio y complejo.

Recientemente vi la película Atlas, no por Jennifer Lopez, como muchos de vosotros estaréis propensos a pensar, sino para enriquecer mi mente con nuevas imágenes sobre futuros posibles. Dentro de esta película, la IA se ve como un complemento del trabajo del hombre y ambos se mejoran y potencian, trabajando en simbiosis.

Para quienes pasaron las tardes de su infancia viendo Star Trek: es la evolución de la fusión mental de los vulcanianos o del simbionte de los Trill.

Hay muchos aspectos de la vida de un programador que se benefician de una IA:

- Generación de boilerplate: La IA puede producir rápidamente estructuras de código básicas, permitiendo a los programadores concentrarse en los aspectos más complejos y creativos.
- Depuración asistida: Modelos como ChatGPT pueden ayudar a identificar errores en el código y sugerir posibles soluciones.
- Exploración de nuevas tecnologías: La IA puede proporcionar explicaciones y ejemplos de uso para frameworks o bibliotecas con las que el programador no está familiarizado.

- Optimización: Sugerencias para mejorar la eficiencia o la legibilidad del código.
- Pruebas: Generación de pruebas automáticas para verificar el correcto funcionamiento de su trabajo.

En lugar de rechazar categóricamente el uso de las IA, los programadores deberían considerarlas como herramientas que pueden mejorar su productividad y la calidad de su trabajo.

## **Conclusión**

El "celodurismo" en la programación, aunque arraigado en la historia como símbolo de ingenio y optimización, corre el riesgo de convertirse en un freno a la innovación y la eficiencia en el mundo del desarrollo de software moderno.

Un enfoque más equilibrado reconoce el valor de la tradición y la experiencia, pero permanece abierto a las nuevas tecnologías y metodologías que pueden mejorar el proceso de desarrollo.

El verdadero signo de un programador experimentado no es la adhesión dogmática a prácticas del pasado, sino la capacidad de evaluar críticamente y adoptar las herramientas y prácticas más adecuadas para cada situación específica.

Admiro a los celoduristas por su tenacidad, pero estoy seguro de que si tuvieran más coraje podrían obtener grandes beneficios de una reevaluación de sus posiciones.

## **Los programadores son los nuevos mercenarios: la evolución del trabajo en el sector IT**



En el pasado, construir un producto con un equipo de programadores era sinónimo de estabilidad a largo plazo. Las variaciones en el personal eran raras, limitadas a eventos como jubilaciones, traslados personales, burnout y, ocasionalmente, ofertas de trabajo irresistibles. Pero el viento del cambio ha soplado fuerte: la difusión de Internet y la reciente pandemia han acelerado un proceso que parecía inevitable: la llegada del trabajo remoto.

Hoy en día, incluso alguien que vive en una casa aislada en los Alpes puede colaborar con una empresa en una metrópoli lejana con la misma facilidad con la que podría trabajar para la empresa del pueblo vecino. El mundo del trabajo IT se ha transformado, y con él, las reglas del juego.

## **De centralizados a descentralizados: un nuevo paradigma**

Hemos pasado de un modelo de fuerte localización de empresas y personal a un concepto revolucionario de trabajo distribuido en el territorio. Hoy en día, hablar de empresas "full remote" ya no es una herejía, sino una realidad consolidada. El personal trabaja desde casa o desde lugares diferentes, a menudo en países lejanos entre sí. Esta transformación ha abierto las puertas a un número creciente de profesionales, incluidos aquellos menos inclinados al cambio, que ahora se encuentran inundados por un flujo continuo de ofertas laborales, impensable solo hace unos años.

En la era digital, incluso las listas de empresas que ofrecen trabajo remoto se han convertido en una realidad. En Italia, por ejemplo, el proyecto GitHub "Awesome Italia Remote" recopila las empresas italianas que ofrecen trabajos Full Remote, completo con tecnologías requeridas y páginas para las candidaturas:

<https://github.com/italiaremove/awesome-italia-remote>

Esta metamorfosis del trabajo refleja lo que el economista Richard Baldwin ha definido como "la gran convergencia". En su libro "The Great Convergence: Information Technology and the New Globalization", Baldwin afirma: "El impacto

de la tecnología de la información está creando una nueva ola de globalización que permite que los servicios se proporcionen a distancia, cambiando radicalmente el panorama del trabajo global."

Esta nueva realidad es un terreno fértil de oportunidades y desafíos, tanto para los trabajadores como para las empresas. Para los profesionales del IT apasionados por su trabajo, libres de fuertes lazos empresariales y en busca de salarios más altos o entornos laborales estimulantes, esta podría considerarse una época dorada. La flexibilidad y las oportunidades han crecido exponencialmente, abriendo horizontes antes inimaginables.

## **De los años '80 a hoy: un viaje en el tiempo del trabajo IT**

Recuerdo con nostalgia mi primer empleo en una empresa de software. El equipo era un grupo unido de personas que trabajaban juntas desde hacía años, todas residentes a pocos kilómetros de distancia. En ese contexto, la estabilidad parecía un hecho, y las ofertas de trabajo externas eran raras y a menudo poco atractivas.

Ese "paraíso" empresarial, donde no era necesario ningún esfuerzo para retener a los empleados, es ahora un recuerdo desvaído.

Hoy en día, las empresas se enfrentan a un desafío titánico: mantener a sus talentos. La retención del personal se ha convertido en una misión empresarial crucial. Sustituir a un miembro del equipo no solo es costoso en términos económicos, sino que también requiere una inversión de tiempo valioso para el entrenamiento y la integración.

En el panorama actual, un programador ya no es solo un "code monkey". Es un profesional polifacético que concentra en sí mismo una constelación de competencias que van mucho más allá de la mera capacidad de escribir código: conocimientos de producto, comprensión de las dinámicas empresariales, empatía con los usuarios finales del software que desarrolla.

Los software modernos requieren profesionales con competencias cada vez más especializadas, tanto tecnológicas como de producto. Y son sobre todo estas últimas las que requieren tiempo para ser afinadas y perfeccionadas.

Como observa agudamente el economista del trabajo David Autor del MIT: "Las empresas están invirtiendo cada vez más en capital humano específico de la empresa, haciendo que los trabajadores sean más productivos en sus roles actuales."

La especialización es un arma de doble filo: por un lado representa un valor empresarial inestimable, por otro puede transformarse en un riesgo para la movilidad del personal. La concentración de competencias en pocas mentes brillantes puede hacer que la empresa sea vulnerable a la pérdida de conocimientos fundamentales en caso de dimisiones o jubilaciones.

Para enfrentar este desafío titánico, muchas empresas están implementando estrategias innovadoras de retención dignas de una novela de ciencia ficción. Estas incluyen programas de reembolso de gastos de formación, atractivas ofertas de stock options, revisiones salariales regulares, oportunidades de movilidad interna y asignaciones a corto plazo que parecen misiones espaciales. Además, están invirtiendo en la formación continua, tanto para las competencias técnicas como para las soft skills, ofreciendo mayor flexibilidad en las modalidades laborales y la posibilidad de experimentar con nuevas tecnologías como si fueran exploradores en tierras desconocidas.

Hasta hace unos años todas estas oportunidades estaban reservadas a unos pocos elegidos, a profesionales de alto nivel o a gerentes de largo recorrido. Hoy en día, se han convertido en la norma, un must have para cualquier empresa que quiera mantenerse competitiva en un mercado laboral cada vez más globalizado y competitivo.

En el reino del IT, la formación es la savia vital de los desarrolladores. Encontrar

empresas que inviertan en la formación, tanto internamente como ofreciendo presupuesto para el crecimiento individual, es como descubrir un oasis en el desierto. Representa un valor añadido inestimable para el crecimiento profesional y personal, un motivo para permanecer anclado a la empresa en lugar de dejarse tentar por sirenas que no garantizan iguales oportunidades de crecimiento.

Pero atención: el mercado laboral IT no es un monolito uniforme. Existen disparidades geográficas abismales: en algunas áreas, la formación es un espejismo y las empresas prefieren externalizar las competencias en lugar de cultivarlas en casa. La competencia se ha convertido en una arena global, donde las empresas locales se encuentran luchando no solo entre ellas, sino también contra los gigantes tecnológicos que parecen salidos de una película de ciencia ficción.

## **La disparidad entre grandes y pequeñas empresas: un abismo digital**

Cuando se habla de disparidad, el pensamiento corre inmediatamente a las diferencias geográficas, dictadas por el territorio en el que las personas trabajan. Pero en el mundo IT, donde la geografía está perdiendo significado, la verdadera disparidad está dada por lo que las empresas pueden ofrecer. Y las ofertas de las grandes empresas son cada vez más atractivas en comparación con las de las pequeñas realidades.

Esta disparidad crea una brecha salarial que, dentro de una zona geográfica, puede ser mitigada, pero que a nivel global se convierte en un cañón. Este abismo salarial es una espada de Damocles para las pequeñas empresas que buscan desesperadamente retener a los mejores talentos.

Podemos entonces hablar de una verdadera "migración digital": las personas permanecen físicamente en su territorio, pero su mente y sus competencias viajan a través de la red, trabajando para empresas ubicadas en cualquier rincón del globo. Este brain drain digital representa un desafío titánico para las empresas que buscan retener a los mejores talentos: crear razones para retener

a las personas se convierte en una empresa digna de Hércules.

## **¿Cómo enfrentar estos desafíos? Estrategias para un nuevo mundo del trabajo**

Enfrentar de manera correcta el desafío de la gestión del conocimiento es un paso fundamental para cualquier empresa que quiera sobrevivir en esta jungla digital. Es crucial evitar concentrar competencias fundamentales en una sola persona, como si fuera un oráculo insustituible. Hay que apuntar a crear proyectos autónomos y autosuficientes, como ecosistemas capaces de prosperar independientemente. La pérdida de un técnico experto puede ser un golpe devastador, llevando a una significativa reducción del capital empresarial, no solo en términos de competencias técnicas, sino también de conocimiento de los procesos y de la cultura empresarial.

Desde el punto de vista de los profesionales, es importante considerar que cambiar de trabajo solo por motivos económicos podría no ser siempre la elección más sabia, especialmente para los jóvenes al inicio de su odisea profesional. Pasar por diferentes posiciones y situaciones puede proporcionar una experiencia valiosa y una madurez inestimable, pero es esencial encontrar un equilibrio. El dinero es importante, claro, pero no es el único tesoro a buscar cuando se evalúa una oportunidad de trabajo.

Durante mis primeros años de trabajo, recuerdo una frase que me impactó como un rayo en un cielo despejado:

**no menos de 2, no más de 5**

Luego escuché este mantra en mil contextos diferentes, pero esa vez me sacudió particularmente. La frase se refería al número de años que un profesional debería pasar en una empresa antes de cambiar de trabajo. Menos de dos años podría interpretarse como falta de estabilidad, más de cinco como falta de ambición. Este concepto, que puede parecer un fósil en un mundo en rápida evolución, aún

tiene cierta validez, especialmente en un sector como el IT, donde la velocidad del cambio es comparable a la de la luz.

También hay otro aspecto, a menudo subestimado, que es importante aprender cuando se decide emprender la vida de programador: cualquier proyecto que se enfrenta, en los primeros meses de desarrollo, parece un jardín del Edén. ¿Problemas? Raros. Y aun cuando se presentan, se pueden superar con relativa facilidad, como saltar un charco.

Pero superados los primeros meses, los proyectos comienzan a crecer como jóvenes titanes, volviéndose cada vez más grandes y articulados. Las demandas, tanto de los clientes como del management, se transforman en montañas a escalar.

Con el paso del tiempo, dar respuestas consistentes y funcionales a las demandas se convierte en un desafío digno de Sísifo. Es en este momento cuando emergen los verdaderos programadores, aquellos capaces de hacer funcionar un producto utilizado de manera estable, teniendo en cuenta una miríada de aspectos que nunca emergen en la fase de análisis y durante las primeras versiones.

Alcanzar ese nivel significa haberse convertido en verdaderos profesionales, capaces de enfrentar cualquier tipo de desafío y de resolver cualquier tipo de problema. Cambiar de proyectos cada seis meses puede aumentar los propios conocimientos horizontales, pero a expensas de los conocimientos verticales, aquellos que a menudo marcan la diferencia entre realizar un producto software y hacerlo funcionar.

## **Conclusiones: navegar en el nuevo mundo del trabajo IT**

En conclusión, hemos entrado en una era en la que los equipos son fluidos como el agua y la estabilidad se ha convertido en un concepto del pasado, una reliquia de museo. Las empresas deben adaptarse a esta nueva realidad, ofreciendo condiciones competitivas y un entorno de trabajo estimulante para atraer y

retener talentos. Al mismo tiempo, los profesionales deben evaluar cuidadosamente las oportunidades, considerando no solo el aspecto económico sino también el crecimiento profesional y personal, como exploradores en busca del Santo Grial.

La tecnología está cambiando la naturaleza del trabajo más rápido de lo que muchas organizaciones pueden adaptarse, como un tren de alta velocidad que pasa mientras las estaciones intentan desesperadamente seguirle el ritmo.

Al mismo tiempo, los programadores deben tener cuidado de no dejarse engañar por las fáciles recolocaciones empresariales y el fácil dinero, porque corren el riesgo de dar la impresión de ser simples mercenarios del código y no personas apasionadas y competentes en su trabajo. El verdadero valor de un programador no se mide solo en líneas de código o en salario, sino en la capacidad de crear, innovar y dejar una huella duradera en el mundo digital.

En este nuevo Lejano Oeste tecnológico, solo quien sepa equilibrar ambición y lealtad, competencias técnicas y soft skills, podrá emerger como verdadero pionero del coding. El futuro pertenece a aquellos que sepan navegar estas aguas tumultuosas con sabiduría, adaptabilidad y una pasión inquebrantable por su oficio.

## Un mejor salario no es suficiente para motivar un cambio



Periódicamente dedico tiempo a hacer un balance de lo que he logrado, a reflexionar si mi comportamiento ha sido correcto y si podría haber mejorado en algún aspecto. A todos nos pasa cometer errores, subestimar o sobreestimar una situación; volver a enfrentarla en frío, con una mente crítica, ayuda a encontrar respuestas que "en caliente" a menudo no surgen.

Nosotros mismos cambiamos con el tiempo y, con nosotros, cambian nuestras prioridades, nuestros objetivos y nuestros valores.

El análisis de hoy está dedicado a una serie de eventos ocurridos a finales de 2023, que me hicieron reflexionar sobre lo difícil que es hacer que las personas cambien, incluso cuando se les presenta una propuesta atractiva, o al menos así lo consideraba yo.

De vez en cuando me toca integrar a mi equipo de desarrollo con nuevas personas, ya sea para reemplazar a quienes han dejado la empresa o para ampliar el equipo para nuevos proyectos.

Dejar una empresa es una práctica normal que, como ya he descrito, todo gerente debería prever, pero que a menudo se subestima.

Para citar a Jane Austen:

**Nadie es demasiado viejo para cambiar, porque el cambio es la única constante de la vida.**

Por lo tanto, no debemos alarmarnos si uno de nuestros colaboradores decide cambiar, sino que debemos estar preparados para gestionar el cambio, a veces de manera repentina.

Las razones para "dejar" son múltiples. En mi caso, se trata de programadores que han dejado la empresa para ir a otras realidades, de jubilaciones (sí, sucede, no es un espejismo), de reubicaciones: es un error forzar a un programador a

quedarse en un proyecto que no siente suyo. O logramos motivarlo, o es mejor trasladarlo a otro proyecto con diferentes estímulos, para no perjudicarlo a él mismo y, por reflejo, al grupo en el que está integrado.

Esta vez tenía que encontrar a una persona para reemplazar a un programador que se jubilaba: administración normal, gestionada con mucha anticipación, para no tener inconvenientes y poder gestionar de la mejor manera el traspaso de responsabilidades.

## **Fantástico, pero ¿dónde está el problema?**

El problema surgió durante la fase de selección de los nuevos programadores que no aceptaron mi propuesta. Y no: el motivo no era el salario, que era mayor y estaba en línea con las demandas de los candidatos.

En el pasado pensaba que el salario era el principal motivo para impulsar a un programador a cambiar de empresa, influenciado por las miles de veces que leí que era lo primero que las personas evaluaban al decidir cambiar de trabajo, pero tuve que reconsiderarlo.

Me enfocaba demasiado en el salario, víctima del hecho de que es uno de los pocos parámetros objetivos que podemos evaluar y del hecho de que en muchos foros de programadores es el parámetro más discutido y el primero que se evalúa en cada cambio.

Sin embargo, intentemos dar un paso atrás y veamos el tipo de necesidades que debía satisfacer.

La búsqueda estaba orientada a un programador Java experimentado interesado en unirse a nuestro equipo de desarrollo para un producto de backend utilizado a nivel europeo.

Enfocar la solicitud de inmediato creo que es un aspecto fundamental cuando se

aborda por primera vez a un candidato: se ahorra tiempo a ambos y se evitan pérdidas de tiempo innecesarias.

Muchos programadores odian trabajar en proyectos que no conocen, que no entienden, que no los estimulan. Hablar de inmediato sobre el proyecto y qué visibilidad permite tener, ayuda a que el candidato entienda cuán importante y visible será su trabajo.

El segundo aspecto que abordo es el tecnológico.

Trabajar con tecnologías consideradas "obsoletas" o "viejas" es un desincentivo para muchos programadores.

**"No quiero trabajar en Java, es un lenguaje viejo, no me interesa"**

es una frase que he escuchado muchas veces. Así como a menudo he oído decir:

**"¿Qué versión del lenguaje Java usan? ¿La 8? No gracias, no me interesa"**

Incluso si el lenguaje o las tecnologías utilizadas no son un problema, poder usar una versión reciente del lenguaje o tecnologías que no conocen es un incentivo para muchos programadores. Del mismo modo, hacer entender que el proyecto está en continua evolución y que se está intentando llevar el producto a una versión más reciente del lenguaje es un incentivo adicional, siempre que no se incumpla.

En mi caso se trataba de un proyecto iniciado en Java 11 con un porting a Java 17 completado, que hacía uso de varias bibliotecas y frameworks.

El punto importante es que quería hacer percibir que el proyecto estaba en continua evolución y que el trabajo del candidato sería importante para el éxito

del proyecto.

El tercer aspecto que abordo es el del salario.

Sabiendo que es un aspecto importante, indico ya en la primera entrevista el salario que puedo ofrecer, para no crear falsas expectativas: es una ventaja para todos. El candidato ya sabe a qué puede aspirar y yo sé si el candidato está interesado o no.

Enfocando de inmediato el proyecto, los objetivos y las compensaciones, el candidato ya tiene un panorama completo de la situación y puede decidir si continuar con el proceso de selección o no.

Un poco por la claridad, un poco por la demostración de que el stack tecnológico estaba actualizado, un poco por haber expuesto desde el principio el salario, recibí un número adecuado de candidaturas y, tras una fase de selección, identifiqué a algunos candidatos prometedores. Me parecían bien a mí, le parecían bien al equipo, le parecían bien a los gerentes y, sobre todo, todas las demandas del candidato estaban satisfechas: trabajar en un ámbito conocido, usar tecnologías recientes, tener un salario mayor.

A pesar de las premisas que me animaban a pensar que todo se podría resolver de manera positiva, recibí respuestas negativas que no esperaba:

"No me siento preparado"

## **Cuando el salario no es suficiente**

El motivo principal por el que algunos candidatos rechazaron nuestra propuesta fue no sentirse preparados para el cambio: el trabajo actual, aunque menos estimulante, menos interesante, menos pagado, era de todos modos un trabajo seguro y conocido.

El cambio, aunque para mejor, era percibido como un riesgo, un salto al vacío.

Esto me hizo reflexionar sobre el hecho de que, incluso si te encuentras en una empresa que no te satisface y puedes tener todas las razones para cambiar, si no estás inclinado al cambio no cambiarás, o al menos no lo harás por un cambio mínimo. Se necesitan motivaciones más fuertes o una dialéctica capaz de hacer percibir el cambio no como un momento de estrés, sino como un momento de crecimiento y valorización.

Este fenómeno lo he notado sobre todo entre las personas más maduras, que prefieren permanecer en contextos conocidos en lugar de enfrentar un cambio, incluso si es beneficioso.

La "zona de confort", aunque problemática, puede hacernos aceptar situaciones tóxicas como normales.

**El mayor obstáculo para el cambio no es la ignorancia o la resistencia, sino la ilusión de que lo que hacemos ahora es seguro.**

Esto también me hizo pensar en todas las veces que alguien me ha confesado estar insatisfecho con su trabajo, pero no querer cambiar por miedo a no encontrar algo mejor.

**El miedo al cambio es un sentimiento común, pero puede superarse con las estrategias adecuadas.**

Según lo que he podido observar, algunas estrategias que pueden ayudar a superar el miedo al cambio son:

- Pequeños Pasos: para quienes no aman el cambio "brutal", la técnica de los pequeños pasos puede ayudar. ¿Sientes que lo que estás haciendo no te satisface? Comienza a estudiar lo que te gustaría hacer. Cada día dedica tiempo a hacerlo y verás que con el tiempo tendrás suficientes conocimientos para dar el

salto que tanto deseabas.

- Mentalidad de Crecimiento: no veas los desafíos con miedo, sino como oportunidades para crecer y evolucionar tu situación. Tener una mentalidad orientada al crecimiento permite salir de manera casi transparente de tu jaula mental.
- Visualización: no veas el cambio con negatividad, sino intenta llevar la mente más allá y pensar en lo que podría suceder de positivo una vez cambiado.
- Apoyo Social: buscar el apoyo de colegas, amigos o mentores ayuda a entender mejor lo que puede suceder y puede proporcionar el aliento necesario para superar el obstáculo mental que impide cambiar.

### **Los errores que cometí**

Es fácil culpar a los candidatos si alguien decide no aceptar tu oferta. Abordar el tema de manera inteligente debería, en cambio, hacer que te preguntes: ¿dónde me equivoqué?

Si no somos capaces de entender qué no funcionó, no seremos capaces de mejorar.

En mi caso intenté buscar la respuesta más allá del salario, más allá del proyecto, más allá de las tecnologías utilizadas.

**¿Y si detrás de ese "no me siento preparado" en realidad hubiera un "no es exactamente lo que busco"?**

Intenté entender qué podría haber impulsado a un programador a no aceptar una propuesta que parecía atractiva.

Una de las razones que me di fue seguramente el contexto empresarial en el que

el candidato debería haber trabajado, o al menos la forma en que lo había descrito durante la entrevista.

Alguien habla de "atractivo corporativo", es decir, el atractivo que una empresa tiene para un candidato.

El atractivo corporativo es un conjunto de factores que van desde el estilo de trabajo hasta la cultura empresarial, desde los beneficios ofrecidos hasta la posibilidad de crecimiento, desde la posibilidad de formación hasta la posibilidad de trabajar en un ambiente estimulante.

El atractivo corporativo es un factor importante que a menudo se subestima, pero que puede marcar la diferencia entre un candidato que acepta la propuesta y uno que la rechaza.

Mostrar que tu empresa tiene valores, cree en la valorización de las personas, ofrece la posibilidad de crecer y formarse, ofrece un ambiente de trabajo estimulante y confortable, puede marcar la diferencia entre un candidato que acepta la propuesta y uno que la rechaza.

## **Conclusiones**

Pensar que solo ofreciendo un salario mayor se puede motivar un cambio es un error que he cometido a lo largo de mi vida.

No he incorporado a personas que podrían haber marcado la diferencia para mi equipo y para mi proyecto.

Esto me ha enseñado que el mayor trabajo que debería hacer cada gerente es hacer que su empresa sea atractiva para los candidatos, hacerles percibir que trabajar en esa empresa es una oportunidad de crecimiento y valorización.

Hacer percibir al candidato que podrá trabajar con autonomía, con las herramientas adecuadas, que crecerá año tras año y será una pieza importante para el éxito de la empresa.

En ese punto, muchos aspectos que antes se consideraban fundamentales pasan a un segundo plano.

El salario, el proyecto, las tecnologías utilizadas se convierten en solo un detalle, un aspecto secundario en comparación con el atractivo corporativo y la posibilidad de crecimiento que ofrece la empresa.

## **Quien pierde un programador pierde un tesoro**



Un viejo dicho dice: "Quien encuentra un amigo encuentra un tesoro". El significado positivo de esta frase radica en el hecho de que la amistad, la verdadera, aquella que te permite dormir tranquilo porque sabes que pase lo que pase siempre tendrás un amigo en quien confiar, que te ofrece ayuda por voluntad y no por obligación, es algo raro.

De la misma manera, un buen programador es un tesoro para una empresa. Es una persona que, gracias a sus habilidades técnicas, su experiencia y su capacidad para resolver problemas, puede marcar la diferencia entre el éxito y el fracaso de un proyecto.

Sin embargo, si miramos el otro lado de la moneda, "Quien pierde un programador pierde un tesoro". Con demasiada frecuencia se subestima la pérdida de alguien que con el tiempo ha acumulado experiencia y habilidades dentro de la empresa y ahora decide irse a otro lugar.

Demasiadas veces he visto empresas sufrir contratiempos debido a una persona que dejaba la empresa, ya fuera un programador, un gerente de proyectos o un analista. Este artículo pretende ser una reflexión sobre lo que significa perder un programador y cómo las empresas pueden evitar que esto suceda.

## **La vida profesional de un programador**

Los programadores, como todos los profesionales, atraviesan diferentes fases significativas durante su carrera laboral. Sin embargo, a menudo es posible identificar un patrón repetitivo que se manifiesta con particular frecuencia en este sector.

La primera fase es la de incorporación a un proyecto. Se trata de un período inicial, más o menos largo, donde la persona es introducida en el contexto laboral. Es un momento caracterizado por un estudio intenso alternado con momentos de trabajo práctico, a menudo bajo la supervisión de un programador más experimentado que actúa como guía.

Luego sigue el inicio de las actividades laborales propiamente dichas. En esta fase, el programador comienza a trabajar en las primeras tareas asignadas, se familiariza con el proyecto y con las personas del equipo. Aún no es completamente productivo, pero empieza a comprender las dinámicas del proyecto y del grupo de trabajo.

La tercera fase es la de madurez profesional. El programador comienza a trabajar de manera completamente autónoma, sabe con quién debe interactuar, conoce a fondo los procesos empresariales, comprende los problemas y desarrolla soluciones efectivas. Es un momento de gran crecimiento y satisfacción profesional.

Finalmente, llega la fase de asentamiento, donde el programador ya tiene casi todo el código del proyecto en sus manos. La emoción y la curiosidad que caracterizaban las fases iniciales tienden a disminuir gradualmente. Se pasa de una situación en la que "todo es posible", impregnada de euforia y curiosidad, a una fase de mayor conciencia, hasta llegar a veces a pensar que "todo ya está hecho" o que "nada más se puede hacer".

Es precisamente en este momento cuando el programador comienza a cuestionarse, haciéndose preguntas legítimas sobre su futuro profesional, sobre el trabajo actual, sobre el proyecto y sobre la empresa.

Como dice Aristóteles en el libro "Ética Nicomáquea":

**La felicidad no es un momento o un estado que se alcanza de una vez por todas, sino un proceso que dura toda la vida y requiere un esfuerzo constante en el ejercicio de la virtud.**

Por esta razón, las empresas deberían intervenir de manera proactiva para apoyar a sus programadores en esta fase.

## **La pérdida de estímulos**

Esta fase representa uno de los momentos más críticos de la vida laboral: la pérdida gradual de estímulos, ambiciones, perspectivas y objetivos. Todo parece gris, o al menos cristalizado en una situación que parece inmutable.

Hay profesionales que enfrentan esta fase de manera constructiva, canalizando sus energías en la mejora del código existente, en la mejor estructuración del proyecto, en la optimización de los procesos, en el perfeccionamiento de la documentación. Se dedican a aquellas actividades del backlog que con demasiada frecuencia se posponen y que encuentran espacio solo en momentos de aparente estancamiento.

Otros, en cambio, comienzan a desarrollar un profundo sentido de desánimo, aburrimiento, frustración. Empiezan a mirar más allá, en busca de nuevos estímulos, nuevos proyectos, nuevas realidades empresariales. Y del desánimo al salto hacia una nueva empresa, el paso es sorprendentemente corto.

Este proceso, bien visto, no es completamente negativo. En muchas profesiones se llega a un punto de saturación que impulsa al cambio. La ventaja peculiar del programador es que, si posee las habilidades adecuadas, puede realizar este cambio con relativa facilidad en comparación con otras profesiones. Puede transformar su situación laboral de manera rápida y, a menudo, sin siquiera tener que abandonar su puesto de trabajo.

## **La pérdida de un programador**

Contrariamente a lo que se podría pensar superficialmente, un programador no representa simplemente un elemento reemplazable en la cadena de valor de una empresa. Esto es particularmente cierto en contextos donde el equipo de trabajo no es muy numeroso y las habilidades son altamente específicas, concentradas en la mente de pocas personas clave.

No todos trabajan en grandes empresas, bien estructuradas y redundantes. A

menudo se encuentran en pequeñas empresas, donde el equipo de desarrollo es menor a diez personas y donde la pérdida de un programador puede tener consecuencias devastadoras.

Para complicar aún más la situación, están los proyectos que no tienen una documentación adecuada, son el resultado de años de trabajos superpuestos y donde los procesos no están estructurados de manera óptima. En estas situaciones, el conocimiento holístico, cuando está presente, permanece concentrado en manos de un número reducido de personas.

En este escenario, perder un programador se convierte en un problema serio, porque realmente se pierde un tesoro y, en algunos casos, incluso el control efectivo del proyecto que estaba siguiendo.

Incluso en las realidades empresariales más estructuradas, donde los equipos están bien organizados y el código está cuidadosamente documentado y probado, surge la tendencia a buscar a alguien que sepa replicar exactamente las habilidades y actividades del programador saliente.

Esta aspiración a menudo resulta irrealista, ya que cada programador posee su propio estilo distintivo, sus hábitos consolidados, sus habilidades específicas. No es nada sencillo encontrar a alguien que pueda reemplazarlo de manera perfectamente superponible.

En estas situaciones se escucha frecuentemente repetir: "llevo meses buscando a alguien, pero no lo encuentro". Esta afirmación, aparentemente banal, encierra en realidad varias verdades significativas: destaca cómo la persona a reemplazar poseía habilidades específicas que el equipo actual no puede cubrir y sugiere que probablemente la compensación que percibía estaba subdimensionada respecto al valor real que aportaba a la organización.

Otro aspecto frecuentemente subestimado en estos casos es la expectativa de que el nuevo programador pueda, desde el primer día, cubrir total o parcialmente

el trabajo del predecesor. Sorprendentemente, todavía hay quienes cultivan esta ilusión.

Es fundamental, en cambio, tener en cuenta que, durante un período significativo, el equipo no podrá mantener los mismos niveles de rendimiento. Esta problemática se vuelve aún más crítica si el programador saliente es uno de los más experimentados o si el equipo está compuesto por un número limitado de personas.

## **Cómo evitar la pérdida de un programador**

La conciencia del valor de un programador debería impulsar a las empresas a crear un entorno laboral que favorezca su permanencia y su desarrollo profesional. El elemento fundamental en este sentido es garantizar un nivel adecuado de autonomía en el trabajo diario. Un programador debería tener la libertad de organizar su propio tiempo y sus actividades, eligiendo las modalidades más efectivas para alcanzar los objetivos establecidos. Esta autonomía no significa aislamiento o falta de coordinación con el equipo, sino más bien la posibilidad de expresar al máximo su creatividad y sus habilidades técnicas sin excesivas restricciones procedimentales.

La empresa debe además asegurarse de que el programador tenga a su disposición todas las herramientas necesarias para realizar su trabajo de la mejor manera posible. Esto significa no solo proporcionar hardware y software adecuados, sino también garantizar el acceso a recursos de formación, documentación y soporte técnico. Con demasiada frecuencia las empresas subestiman el impacto negativo que herramientas obsoletas o inadecuadas pueden tener en la motivación y la productividad de un programador. La frustración derivada de tener que luchar diariamente con limitaciones técnicas puede ser un fuerte incentivo para buscar oportunidades en otro lugar.

Platón, en "La República, libro IV", decía, hablando de los alfareros:

**Si a causa de la pobreza no puede procurarse las herramientas u otros utensilios indispensables para su arte, realizará productos más deficientes y hará artesanos inferiores a los hijos u otros a quienes enseñe su oficio.**

Un aspecto crucial, a menudo subestimado, es hacer sentir al programador verdaderamente parte del proyecto en el que está trabajando. Esto va mucho más allá del simple involucramiento técnico: significa hacerlo partícipe de las decisiones estratégicas, escuchar sus opiniones sobre las elecciones arquitectónicas, involucrarlo en la planificación de las actividades futuras. Un programador que se siente verdaderamente parte del proyecto desarrolla un sentido de pertenencia y responsabilidad que va más allá de la simple relación profesional. Este vínculo emocional con el proyecto y con el equipo puede ser un poderoso factor de retención, especialmente en momentos de dificultad o cuando se presentan oportunidades alternativas.

Cuando un programador percibe que tiene voz en las decisiones que afectan su trabajo, que puede contar con herramientas a la altura de los desafíos que debe enfrentar y que es parte integral de un proyecto significativo, las probabilidades de que busque alternativas profesionales disminuyen sensiblemente. No se trata solo de retener a un profesional calificado, sino de crear las condiciones para que pueda expresar al máximo su potencial, contribuyendo así no solo al éxito del proyecto sino también al crecimiento de todo el equipo.

## **Conclusiones**

El recorrido profesional de un programador representa un viaje complejo, caracterizado por diferentes fases de crecimiento, momentos de entusiasmo y períodos de reflexión. Es importante que las empresas sean conscientes de estos momentos críticos y se comprometan activamente a crear un entorno de trabajo que favorezca la permanencia y el desarrollo de sus programadores.

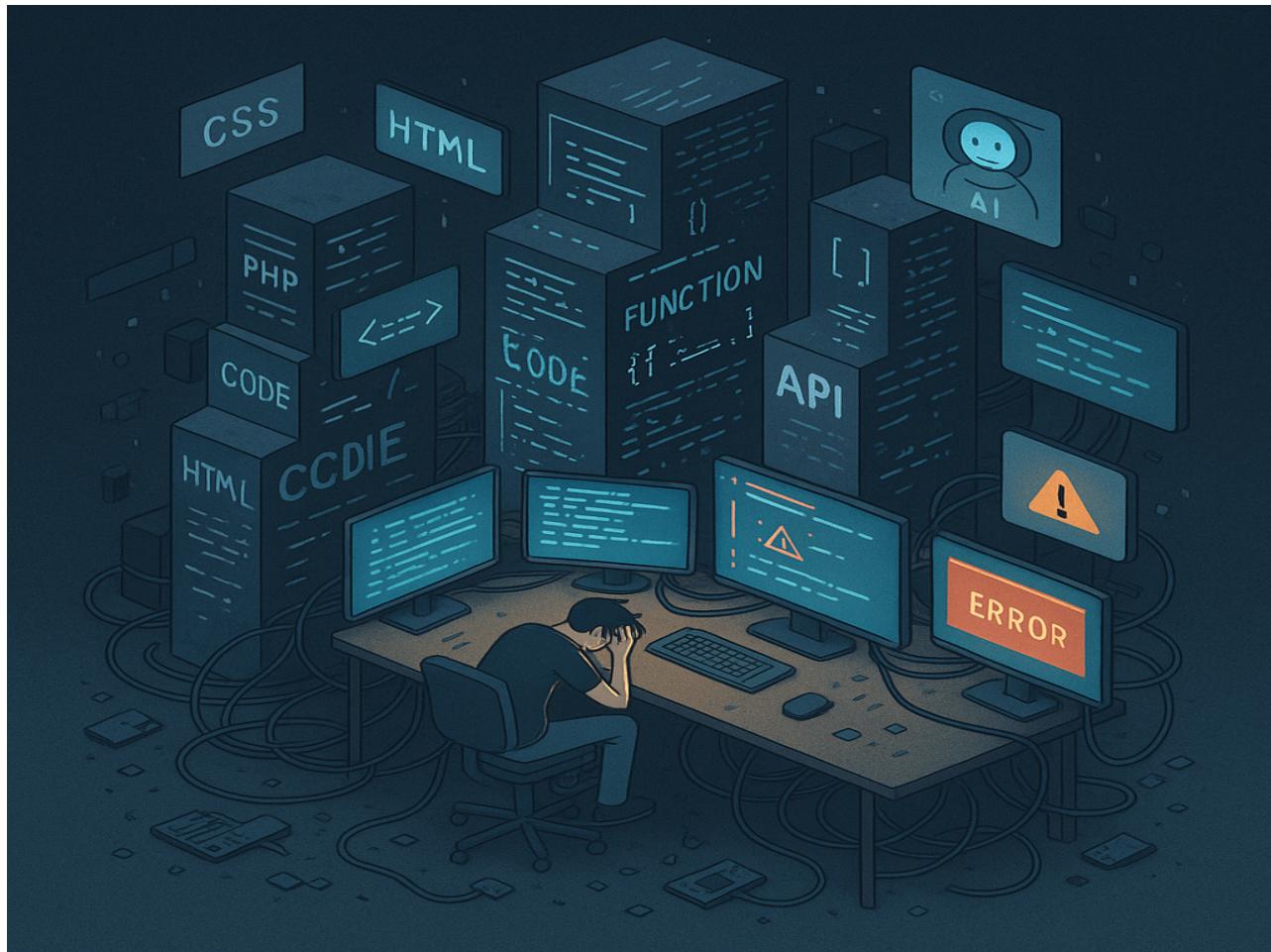
La metáfora del tesoro, con la que hemos abierto esta reflexión, adquiere aún más significado cuando se considera el conjunto de conocimientos, experiencia y

valor que un programador aporta consigo. No se trata solo de habilidades técnicas, sino de un patrimonio hecho de comprensión profunda de los procesos, de las dinámicas del proyecto y de las relaciones dentro del equipo.

Cuando una empresa pierde un programador, no pierde simplemente un recurso, sino un pedazo de su propia historia y de su futuro.

El desafío para las organizaciones modernas no es solo atraer talentos, sino crear un entorno en el que estos puedan crecer, expresarse y encontrar estímulos continuos. Solo a través de un compromiso concreto en valorar a sus programadores, proporcionándoles autonomía, herramientas adecuadas y un verdadero sentido de pertenencia, las empresas pueden esperar conservar este tesoro. En un mundo donde la tecnología evoluciona rápidamente y las habilidades son cada vez más valiosas, invertir en la satisfacción y el crecimiento de sus programadores ya no es solo una elección, sino una necesidad estratégica para el éxito a largo plazo.

**¿Y si el eslabón débil de la programación fuera el propio programador?**



Mientras las inteligencias artificiales prometen revolucionar el desarrollo de software acelerando cada fase del proceso, surge una paradoja inquietante: el propio programador podría haberse convertido en el principal obstáculo para la velocidad que exige el mercado. En un ecosistema donde la automatización genera código a ritmos inimaginables, la supervisión humana introduce latencias, controles y ralentizaciones que contradicen el objetivo principal de la eficiencia.

## **Un poco de historia**

Intentemos comprender el mundo de la programación, reflexionando sobre la evolución normal de un proyecto y pensando en cómo un programador se mueve a lo largo del tiempo.

Cuando empezamos a dar nuestros primeros pasos en este mundo, comenzamos a estudiar los primeros lenguajes y a conocer los primeros componentes, librerías y frameworks que pueden ayudarnos en nuestro trabajo.

El estudio de las mejores prácticas y un conocimiento cada vez más profundo de las herramientas utilizadas tienden a enfocar nuestro trabajo en un camino conocido que tendemos a replicar, en parte o totalmente, en cada nuevo proyecto.

Si reflexionamos sobre la programación, a menudo es un proceso lineal: hoy creamos una entrada de datos funcional, mañana implementamos controles eficientes, luego pensamos en mecanismos de abstracción de datos, creamos API, optimizamos el rendimiento, aumentamos las medidas de seguridad, y así sucesivamente.

Cada proyecto se convierte en la suma de las experiencias anteriores, debidamente reorganizadas: quizás sea mejor diseñar las API desde el principio, la seguridad sería mejor implementarla como primer paso, usemos una librería de controles consolidada, este servicio de autenticación, etc. Sin embargo, la suma de experiencias, en una perspectiva de evolución continua, puede ocultar una trampa insidiosa.

## **El creciente peso de la complejidad**

El verdadero problema no reside en el código en sí, sino en la complejidad que se acumula inexorablemente con el tiempo. Cada nuevo proyecto representa un punto de partida que rara vez es realmente nuevo: el programador introduce inmediatamente todo lo que ha aprendido en años anteriores, acumulando inconscientemente capas de complejidad.

Inicialmente, esta complejidad parece manejable, pero progresivamente se convierte en una carga cada vez más difícil de soportar. Cada nueva funcionalidad, cada error que corregir, cada solicitud del cliente añade una capa adicional que se sedimenta sobre el código existente, creando una estratificación que requiere competencias cada vez más especializadas para ser gestionada.

Esta dinámica choca frontalmente con las expectativas del mercado actual, donde la velocidad se ha convertido en el parámetro dominante. Los clientes quieren todo y lo quieren ya:

Crea un sitio web como el de Apple, pero en una semana, ya que lo tienes todo listo.

El prototipo debe estar listo para esta noche.

El error debe solucionarse hoy mismo.

y para terminar con una cita que habré oído un número infinito de veces:

¿Para cuándo quieres esta funcionalidad?

¡Para ayer!

La presión del tiempo, siempre presente, se vuelve cada vez más constante e ineludible.

## **La ilusión de la compresión temporal**

Frente a esta presión, la industria ha intentado diversas estrategias de automatización. En el pasado existían los CASE (Computer-Aided Software Engineering), fábricas de código especializadas para categorías específicas de problemas. Con la evolución de las necesidades, estas herramientas se han transformado en los frameworks modernos: más flexibles y de propósito general en comparación con la verticalidad de los CASE, pero todavía complejos y no siempre intuitivos en su uso.

La llegada de las inteligencias artificiales ha representado la promesa definitiva: programas que escriben código por nosotros, que resuelven problemas automáticamente, que permiten ahorrar tiempo y esfuerzo. Sin embargo, esta aparente solución choca con una realidad ineludible: el programador sigue siendo el controlador de este proceso: un controlador humano, con todas sus limitaciones y sus tiempos de reacción.

Incluso delegando la escritura del código a la IA, persisten tiempos irreductibles: aprendizaje, adaptación, depuración, pruebas y supervisión humana. Mientras esta supervisión siga siendo necesaria, el tiempo total no podrá reducirse drásticamente. Es precisamente en esta dinámica donde emerge el programador como potencial eslabón débil: el elemento humano que introduce sobrecarga y latencias en el proceso de automatización.

## **¿Pero cómo nos están ayudando y evolucionando las IA?**

El 2025 fue el año del "vibe coding": programación asistida por inteligencia artificial donde cada problema se resuelve a través de prompts iterativos. Si algo no funciona, se genera un nuevo prompt; si es necesario modificar algo, otro prompt más.

Esta evolución ha transformado la IA de una simple herramienta de apoyo a la principal herramienta de trabajo.

Llevo meses trabajando en modo "Vibe", primero con prompts introducidos en

varios LLM, luego a través de Copilot, después con herramientas CLI como Gemini y Copilot Agent, hasta llegar a herramientas más envolventes.

Después de un tiempo de darle vueltas, empiezas a ver sus límites e incongruencias, aunque a menudo piensas que son más tuyos que de la máquina que estás usando.

Como yo, otras figuras destacadas, como Jack Dorsey, han comenzado a utilizar la IA para sus "proyectos de fin de semana", creando aplicaciones como BitChat íntegramente a través de código generado automáticamente.

Sin embargo, la experiencia directa con estas herramientas revela una dinámica compleja. Confiar completamente en el "vibe" para generar grandes porciones de código produce un elevado número de elementos que hay que revisar y corregir. La capacidad interpretativa de la IA, aunque impresionante, puede desviarse fácilmente de la dirección prevista.

El enfoque más eficaz ha resultado ser el de la fragmentación: utilizar la IA para pequeñas tareas específicas, controlar y probar cada componente, y luego ensamblarlos manualmente. Este método permite mantener el control de calidad mientras se aprovecha la aceleración que ofrece la automatización. Es análogo al proceso de domesticación: es necesario conocer, comprender y guiar la herramienta antes de poder utilizarla eficazmente.

## **Cuando el programador se detiene: escenario futuro próximo**

La creciente aceleración de los procesos de desarrollo plantea una cuestión crítica: ¿qué ocurre cuando el programador no está disponible? En el pasado, la ausencia de un desarrollador simplemente implicaba una ralentización proporcional. Los ciclos de lanzamiento eran anuales, la presión por nuevas funcionalidades limitada.

En el presente, la aceleración ha transformado cada interrupción en un potencial

problema sistémico. En el futuro próximo (2-3 años), es plausible que los proyectos continúen evolucionando de forma no supervisada durante la ausencia del programador, con la IA generando autónomamente código, resolviendo issues y gestionando ramas.

A su regreso, el programador se encontraría frente a decenas de ramas que examinar, issues que comprender, errores que analizar para verificar la corrección del código generado automáticamente. La carga mental de esta situación es significativa: sentimiento de culpa por la ausencia, ansiedad por la supervisión fallida, presión por recuperar el control de un proceso que ha seguido evolucionando de forma autónoma.

Este escenario puede degenerar fácilmente en un círculo vicioso de burnout, abandono de la empresa o pérdida de interés profesional, resumible en la peligrosa actitud:

"Aceptemos esta pull request, total, la IA ya ha hecho todo el trabajo, ¿qué puede salir mal?"

### **El futuro del programador: más allá del presente inmediato**

Proyectándonos unos años en el futuro, la evolución del rol del programador parece inevitable. La IA ya supera a cualquier mecanógrafo en velocidad, empieza a producir código de calidad superior al de muchos desarrolladores, y puede cometer errores a una velocidad sin precedentes. Sin embargo, sigue siendo potencia pura sin pensamiento estratégico.

La transformación del rol del programador probablemente seguirá esta trayectoria: menos tiempo dedicado a la escritura directa de código, más tiempo en la interacción con las IA y en el trabajo orientado al negocio. Este cambio requerirá conciencia y preparación adecuada, ya que trastocará por completo la concepción actual de la programación, empujando a los profesionales más allá de sus competencias actuales.

En este nuevo paradigma, una persona describirá un problema en unas pocas frases y un modelo lingüístico especializado en el análisis generará los prompts para un modelo especializado en la escritura del código. El programador evolucionará de codificador manual a guía, probador y, quizás, perfeccionador del código generado por la IA.

Esta transformación implicará la desaparición de algunas prácticas actuales: la depuración tradicional, la necesidad de documentar minuciosamente el código, la importancia de variables con nombres explícitos. Paralelamente, probablemente surgirán nuevos lenguajes de programación diseñados para la interpretación por parte de las máquinas en lugar de para la legibilidad humana.

### **La mente humana: el límite infranqueable**

En este escenario de aceleración continua, eficiencia llevada al extremo y automatización desenfrenada, existe un elemento que no puede ser comprimido: la mente humana. Moldeada por millones de años de evolución, nuestra mente ha desarrollado mecanismos para resolver problemas complejos a través de procesos lentos y ponderados.

Privar a esta "máquina biológica" del tiempo necesario para procesar información y sobrecargarla de entradas continuas produce un solo resultado: el bloqueo funcional. En un mundo donde cada ralentización genera efectos dominó desastrosos, tomarse días de enfermedad o de descanso corre el riesgo de convertirse en un lujo inaccesible.

En las organizaciones estructuradas inadecuadamente, donde cada persona es individualmente responsable de actividades específicas sin una adecuada distribución de las cargas de trabajo, el peso mental y físico que cada programador debe soportar se volverá progresivamente insostenible.

### **Conclusiones: el eslabón débil que puede romper la cadena**

La programación está atravesando una transformación epocal con la llegada de las IA. Debemos aprender a convivir con estas tecnologías preservando nuestra humanidad y capacidad de pensamiento crítico. La velocidad que exige el mercado no siempre es sostenible, y a veces es preferible ralentizar para garantizar la calidad del trabajo.

Sin embargo, esta elección nos expone al riesgo de ser percibidos como el eslabón débil de la cadena productiva, el elemento que impide la compresión temporal y la obtención de resultados inmediatos. Ante esta perspectiva, deberemos interrogarnos sobre la necesidad y utilidad de nuestro rol, o si nos hemos convertido en una carga para la organización.

Concluyo con una reflexión de Stanisław Jerzy Lec que considero particularmente significativa:

**"El eslabón más débil de la cadena es también el más fuerte porque puede romperla."**

El programador, aunque pueda ser considerado el eslabón débil del proceso de automatización, también posee la clave para romper las cadenas de la complejidad y la velocidad impuestas por el sistema.

Nuestra fuerza reside en la capacidad de adaptación, aprendizaje y evolución. Aunque el futuro pueda etiquetarnos como el eslabón débil de la cadena, nuestra tarea es transformar esta aparente debilidad en el punto fuerte de nuestro trabajo y de nuestras organizaciones, de lo contrario nos volveremos pronto obsoletos y reemplazables por máquinas cada vez más inteligentes y autónomas.

## Por qué las startups tech están condenadas a morir



Durante los tórridos días de agosto me encontré charlando con un amigo de la infancia.

Ambos trabajamos en IT: yo en las mesas técnicas, donde uno se enfada por un bit fuera de lugar, y él en las mesas comerciales, donde se intenta vender las ideas que una masa de nerds engendra sin parar.

Orígenes diferentes, una forma paralela de ver la vida y el trabajo, pero con muchas similitudes cuando se habla de fracasos y éxitos.

La reflexión que comenzamos ante un capuchino y un croissant se transformó rápidamente en un análisis de las dinámicas que llevan al fracaso de las startups tecnológicas.

Guido, espero que me perdes si tomo prestadas algunas de tus intuiciones para escribir este artículo.

## **Bienvenidos al matadero de las buenas intenciones**

Según algunos estudios, el 90% de las startups está destinado a fracasar.

[explodingtopics](#)

Decir que el 90% fracasa es una forma gentil de describir un cementerio de ambiciones, una masacre de buenas intenciones tecnológicas: "en septiembre me pongo a dieta" es el paralelo de "en septiembre abro una startup que cambiará el mundo".

La verdad, esa que nadie se atreve a pronunciar en voz alta, es que la mayoría de estas "startups" nunca fueron verdaderas empresas. Eran experimentos de laboratorio, costosos hobbies tecnológicos disfrazados de empresas, liderados por genios del código que no tenían ni la menor idea de cómo se construye un negocio.

## Con nuestra startup ganamos un hackathon

Los inversores más experimentados ya exclaman "me importa un carajo" ante este tipo de afirmaciones, ya que este tipo de "premios" enmascaran la causa de muerte más común, más predecible y menos discutida: la suicida ausencia de perspicacia comercial. Hacer una competencia de quién escupe más lejos en un evento para técnicos no significa tener un modelo de negocio sostenible.

Una startup fundada y dirigida exclusivamente por técnicos no está simplemente *destinada* a fracasar; ya está muerta desde el inicio. Su fracaso no es una cuestión de *si*, sino de *cuándo* se acabará el dinero de los inversores o la paciencia de los fundadores. Se necesita estrategia, se necesita un alma comercial, se necesita una necesidad real y no ficticia, aspectos que la mayoría de las veces no están ni remotamente en las cuerdas de un técnico, que piensa más en la velocidad de su código que en escuchar lo que el mercado le grita.

En una reciente entrevista Enrico Pandian recordó la estrategia de las 2 semanas: "Si no puedes validar tu idea en 2 semanas, probablemente estés construyendo un castillo de arena".

[entrevista](#)

## Anatomía de un fracaso anunciado: la plaga mortal de las startups "Tech-Only"

[CB Insights](#)

Cuando se hacen estos análisis se usa una jerga políticamente correcta, intentando oscurecer la verdadera naturaleza del problema. Aquí no tenemos estas limitaciones y podemos afirmar sin filtros que no es el mercado el que está ausente: es el fundador que no se preocupó de buscarlo. Los fondos no se "agotan" por magia: se queman por decisiones estratégicas desastrosas. Si hasta ayer pasabas el día alineando bits, no puedes despertarte y ser, como por arte de magia, el mago de la estrategia empresarial.

## **Ser irrelevantes (No Market Need): crear soluciones que nadie necesita**

Esta feroz evidencia se manifestó brutalmente en el mundo blockchain: cientos de soluciones que daban garantías que nadie sentía necesidad de tener, excepto las personas que habían construido la startup.

¿Ideas que llegaron demasiado pronto o demasiado tarde? Nunca lo sabremos, pero lo que sí sabemos es que son ideas que interesan a pocos y no a la audiencia inmensa hacia la cual se proponían.

En el mundo de la IA estamos en la misma situación: un número infinito de productos, ya demasiado similares. Esta vez sin embargo tenemos la ventaja de que el hombre común tiene la clara sensación de que son herramientas capaces de aumentar su valor y donde hay percepción de valor puede haber mercado.

La ausencia de una necesidad de mercado es la causa de muerte número uno: un porcentaje impresionante de startups, cuyos números rondan el 40%, es la señal tangible de que se pueden tener ideas brillantes, que en la mayoría de los casos lo son solo dentro de nuestra computadora y no en la cabeza de las personas.

Esto no es un accidente; es un homicidio premeditado de la propia idea, perpetrado por fundadores enamorados de su solución tecnológica pero completamente desinteresados en el problema que debería resolver.

Este fenómeno, conocido como "tech-solutionism", es la enfermedad profesional de los equipos técnicos. Están entrenados para resolver puzzles complejos y crear arquitecturas elegantes; no están entrenados para salir de la oficina y verificar si a alguien, en el mundo real, le importa algo de su creación.

### [\*\*Why do startups fail? A core competency deficit model\*\*](#)

Los resultados son asombrosos: las dos competencias cuya ausencia estaba más

estrechamente correlacionada con el fracaso eran la "búsqueda de información" (Information-seeking) y la "orientación al cliente" (Customer service orientation).

Si sois programadores, ingenieros de software o técnicos, quizás tengáis idea de qué son estas habilidades, pero seguramente no habéis dedicado mucho tiempo a refinarlas.

Buscar información implica validar activamente las hipótesis sobre el mercado, mientras que la orientación al cliente significa dar prioridad a las necesidades de los usuarios sobre la pureza tecnológica: que un flujo de software esté optimizado le interesa a un cliente tanto como saber si el color de vuestra cocina es rojo o amarillo. Si en cambio falta (o no funciona) el campo "dirección de entrega": sí, esto sí que es un gran desequilibrio para vuestra solución software.

Un equipo compuesto exclusivamente por técnicos es, por su propia naturaleza, estructuralmente deficiente en estas áreas vitales. Por tanto, el "No Market Need" no es un fracaso del mercado; es un fracaso directo e inequívoco del equipo al realizar las funciones comerciales más básicas.

Un equipo homogéneo está destinado a construir productos brillantes pero perfectamente inútiles, porque nadie en su interior tiene la tarea, la competencia o el incentivo para hacerse la pregunta más simple e importante de todas: "¿A quién le sirve?"

### Silicon Valley

#### **Agotar los fondos (Ran out of Cash) pero estar llenos de código**

El agotamiento de fondos es la segunda causa de muerte más citada. Hablamos de estimaciones alrededor del 30-35%.

Quedarse sin dinero es una forma elegante de decir que las startups no fueron capaces de generar dinero nuevo.

En 1999 Jeff Bezos fue entrevistado y le preguntaron: "Tu empresa vale miles de millones, pero cada vez que miro el balance veo que está en pérdidas: tu empresa nunca ha tenido beneficios" y Jeff responde serenamente "Es cierto, pero estamos invirtiendo para crecer".

Si la empresa factura y crece cada año, no es problema perder dinero, siempre encontrarás un inversor. Si las migajas que te confían no producen ningún tipo de resultado, el problema es más profundo y estás destinado a agotar todos tus fondos y cerrar o ser absorbido por alguien que es capaz de hacer arrancar tu negocio, pero tu cabeza hipertécnica no es capaz de hacerlo.

El asesino invisible de las startups es una ecuación matemática completamente desconocida para los técnicos:

CAC > CTLV

El coste de adquisición de un cliente (Customer Acquisition Cost) es superior al valor que ese cliente generará en el tiempo (Customer Transactional Lifetime Value).

Un equipo de solo técnicos puede estar obsesionado con métricas de producto como el uptime del servidor, la latencia o la velocidad de carga, pero ignorar completamente las métricas de negocio que determinan la vida o muerte de la empresa.

La incapacidad de gestionar el flujo de caja es una consecuencia directa de este analfabetismo financiero. Los fundadores técnicos, impulsados por el deseo de crear el producto "perfecto", continúan invirtiendo recursos en desarrollo, contratando más ingenieros y añadiendo funcionalidades, sin una estrategia correspondiente para generar ingresos.

En mi pasado muy lejano recuerdo una charla con una empresa francesa que producía una librería para Clipper de la que ahora se me escapa el nombre.

Habían conseguido un enorme contrato que había permitido a la empresa una explosión positiva: contrataciones, ampliación de la oferta y beneficios.

¿Cómo nació este contrato? El departamento comercial había intuido una gran oportunidad en la venta de un software a un cliente. Ese software no existía cuando fue solicitado: en una tarde dibujaron con "PAINT" algunas pantallas del producto, diciendo que existía pero que necesitaban 6 meses para configurarlo en el cliente, dada la complejidad del proyecto.

Bien: el cliente se convenció, firmó el contrato, y en 6 meses trasladaron todo el equipo de proyecto a la realización de esa aplicación que no existía, pero que el cliente estaba convencido de que estaba lista.

Ahora imaginad un enfoque puramente técnico al mismo problema:

no tenemos la aplicación, pero podemos hacerla en 6 meses

El resultado técnico sería idéntico, pero esta frase, en la cabeza de un cliente, deja la sensación de que podríais no lograrlo; decir que ya la tenéis pero que hay que personalizarla genera una percepción de control completamente diferente.

En la misma línea circula en la red una entrevista de Claudio Cecchetto que, después de escuchar la cassette de "Hanno ucciso l'uomo ragno", preguntó a los 883: "Solo son 5 temas, ¿tenéis otros 2?" y ellos: "sí, están en el estudio, vamos a buscarlos y los traemos".

En realidad no existían: se encerraron dos días en una habitación y salieron con otras dos canciones.

Decir "no tenemos nada, pero podemos hacerlas en 2 días" habría transmitido incertidumbre y quizás no habría creado la oportunidad que en cambio lograron aprovechar.

Pero un técnico nunca habría dicho algo así: para decirlo se necesitaba alguien que, en el fondo, era más empresario que técnico.

¿Qué es el genio? Como decía Perozzi:

Fantasía, intuición, decisión y velocidad de ejecución

### **La Plaga de la Invisibilidad (Poor Marketing)**

Si desde algunos puntos de vista la capa de invisibilidad es una ventaja no despreciable, lo sabe bien Harry Potter y las innumerables veces que se la puso, pero si tienes una empresa: tener un producto que nadie conoce no es algo de lo que presumir.

El 14% de las startups fracasa por este mal oscuro. Si sigues el mantra: "si lo construyes, ellos vendrán" (build it and they will come), estás destinado a presentar tu producto solo en los salones de otros nerds como tú.

Los fundadores técnicos a menudo nutren un profundo desprecio por el marketing, considerándolo "paja", manipulación deshonesta o, en el mejor de los casos, una actividad secundaria que hacer "más tarde" cuando el producto sea perfecto. No: esta es una condena a muerte. El producto perfecto no existe. Pensad en muchas herramientas que empezasteis a usar: las primeras versiones eran muy limitadas, sin embargo os enganchasteis inmediatamente, siguiéndolas a nivel stalking.

Según Peter Thiel, cofundador de PayPal:

La causa número uno del fracaso de las empresas es la mala distribución, no el producto

No tengáis miedo de dar a conocer vuestro producto, de aliaros con quien tenga una cadena de distribución, una fuerza de ventas, un marketing potente: será él

quien os dé la posibilidad de expandiros y no vuestro producto.

La incapacidad de hacer marketing no es una simple brecha de competencias: es un prejuicio cultural, un punto ciego ideológico. En el análisis de los fracasos emerge un tema recurrente: la incapacidad de hacer marketing era función de fundadores a quienes les gustaba programar o construir productos pero no les gustaba la idea de promocionarlos.

## **Equipo equivocado (Not the Right Team)**

Tener el "equipo equivocado" es una de las principales causas de fracaso, responsable del cierre de casi un cuarto de las startups.

Tener un equipo equivocado no significa un equipo de personas incompetentes, sino un equipo demasiado homogéneo: imaginad un partido de fútbol con un equipo formado solo por delanteros: todos fenómenos, pero incapaces de contrarrestar al equipo adversario. La derrota se convierte en un elemento ineludible.

El verdadero problema, casi siempre, es la falta de un co-fundador con ADN comercial: si sois un equipo de solo programadores uno de vosotros debe tirar el teclado, o mejor aún debéis encontrar un NO programador, alguien que no es capaz de escribir un "Hello World" (ni siquiera con ChatGPT) pero es capaz de vender frigoríficos a los esquimales.

Una de las empresas de amigos que he visto crecer más está compuesta por una mente técnica brillante y un comercial con gran visión de mercado, cuya unión ha creado un equilibrio ganador.

## **Caso de Estudio: Stripe - Los Programadores que Aprendieron a Vender**

A primera vista, Stripe parece la excepción que confirma la regla. Fundada por

dos hermanos irlandeses, Patrick y John Collison, ambos prodigios de la programación, es una empresa construida por técnicos para otros técnicos (los desarrolladores).

Pero si no nos quedamos en este análisis superficial, descubrimos que su éxito no contradice nuestra tesis, sino que la refuerza de manera aún más potente.

Los hermanos Collison no tuvieron éxito *a pesar* de ser técnicos; tuvieron éxito porque unieron su genialidad técnica a una excepcional habilidad comercial y estratégica.

Entendieron que tenían que venderla activamente. Su estrategia inicial de go-to-market, ahora legendaria y conocida como la "Collison Installation", es una obra maestra de founder-led sales.

Cuando se encontraban con un cliente potencial, no le hacían una presentación en PowerPoint. Le pedían el portátil, y en pocos minutos, ante sus ojos, integraban Stripe en su producto.

En lugar de *hablar* del valor, lo *demostraban* de manera inequívoca.

## **Dejad de Escribir Código, Empezad a Vender**

Ser un buen programador no basta, no basta saber producir soluciones, se necesita un alma marketing, un alma de ventas que no tenga sus raíces en un lenguaje de programación, sino en experiencias y estudios diferentes.

El mejor vendedor que he conocido era un animador de pueblos de verano: tenía un gran don de empatía y simpatía. Te hacía hablar, entendía tus necesidades y sabía cómo persuadirte.

Si sois un grupo de programadores buscad a alguien así, que salga brutalmente

de vuestros esquemas, que sepa valorizar vuestros productos y hablar con las personas. Buscad a alguien con una red de ventas que sepa canalizar lo que estás construyendo. En ese punto el camino se vuelve cuesta abajo y sobre todo tendrás un equipo más amplio, capaz de afrontar mejor los desafíos del mercado.

A un programador no le hace falta un clon, le hace falta un socio. Necesita hablar con los clientes potenciales y entender qué buscan. No hace falta escribir código si no sabemos en qué dirección ir: mejor perder días hablando, que días escribiendo código que luego tiraremos porque no satisface ningún requisito.

El mundo está lleno de código elegante. Lo que escasea son las empresas capaces de transformar ese código en valor real para sus clientes.

Dejad de esconderos detrás de vuestro editor de texto. Salid y vended. Es la única manera de no acabar en el cementerio de los genios.

## **Biografía**

Matteo Baccan es un ingeniero de software y formador profesional con más de 30 años de experiencia en la industria de la tecnología de la información.

Ha trabajado para varias empresas y organizaciones, ocupándose del diseño, desarrollo, pruebas y gestión de aplicaciones web y de escritorio, utilizando diversos lenguajes y tecnologías. También es un apasionado educador en informática, autor de numerosos artículos, libros y cursos en línea dirigidos a todos los niveles de experiencia.

Administra un sitio web y un canal de YouTube donde comparte tutoriales en video, entrevistas, reseñas y consejos de programación.

Activo en comunidades de código abierto, participa regularmente en eventos y concursos de programación.

Se define a sí mismo como un "soñador realista" que ama experimentar, innovar y compartir sus conocimientos y pasiones, siguiendo el lema: "Nunca dejes de aprender, porque la vida nunca deja de enseñar".

## Indice

Prefacio	2
Agradecimientos	4
Introducción	5
El mundo de la programación ha cambiado y con él la manera de convertirse en programador	6
Convertirse en programador hoy . . . . .	8
La paradoja de la elección y la ansiedad por el rendimiento . . . . .	9
El impacto de la Inteligencia Artificial . . . . .	10
¿Qué queda (y qué se necesita) para no perderse? . . . . .	11
"Conviértete en programador después de los 40 años"	14
La realidad . . . . .	15
Sesgos de edad . . . . .	16
La propuesta formativa . . . . .	17
Conclusiones . . . . .	20
Artículos relacionados . . . . .	20
Los programadores que completan todas las tareas no han terminado su jornada laboral	22
Cerrar tareas no es el objetivo final . . . . .	23
El síndrome del caballo de carreras . . . . .	24
Premiando el individualismo . . . . .	24
El problema de "terminar las tareas" . . . . .	25
¿De quién es la responsabilidad? . . . . .	25
¿Qué estrategias podemos implementar para mitigar este problema? . . . . .	26
Gestión de "SuperStar" . . . . .	27
El efecto de la "cadena de montaje" . . . . .	28
Conclusiones . . . . .	29
La fuerza de admitir que no se sabe	30
El miedo a no saber . . . . .	31

Era ignorante y no lo sabía . . . . .	32
La ignorancia es un recurso . . . . .	33
Gestión de la ignorancia . . . . .	34
El coraje de preguntar . . . . .	35
El mito del sabelotodo . . . . .	36
Preguntar solo no basta . . . . .	37
La diferencia entre jamón cocido y jamón crudo . . . . .	37
El miedo al juicio . . . . .	38
Cultura de avanzar a través del fracaso . . . . .	38
Conclusiones . . . . .	39
<b>Más allá del Full Stack: Caminos de Crecimiento en el Desarrollo de Software</b>	<b>40</b>
<b>¿Puedes cambiar esta condición en el código? ¿Qué se necesita?</b>	<b>42</b>
El miedo de los senior . . . . .	43
Los senior no son malos . . . . .	44
¿El mantenimiento de un proyecto cambia las reglas? . . . . .	45
Caso de uso de un conocido banco italiano . . . . .	46
¿Cuál es la percepción de la modificación fuera del proyecto? . . . . .	47
Conclusiones . . . . .	48
<b>La adopción de nuevos frameworks podría hacer fracasar tu proyecto</b>	<b>50</b>
<b>¿Y los frameworks?</b> . . . . .	52
El horizonte temporal: una dimensión crucial . . . . .	53
El riesgo del abandono . . . . .	55
La importancia del análisis periódico . . . . .	56
El riesgo de los servicios no estándar . . . . .	56
Conclusión . . . . .	57
<b>El mito del desarrollador Full Stack: una realidad incómoda</b>	<b>59</b>
Mi enfoque sobre el término desarrollador Full Stack . . . . .	60
El mercado laboral y el desarrollador Full Stack . . . . .	62
La calidad del código y la evaluación del programador . . . . .	62

¿Y si trabajáramos en la "actitud de resolución de problemas" . . . . .	64
Conclusiones . . . . .	65
<b>"Veamos quién la tiene más grande: hagamos revisión de código"</b>	<b>67</b>
La revisión de código molesta . . . . .	68
El código no utilizado . . . . .	70
Yo uso TAB, quien usa espacios no es un buen programador . . . . .	70
Resistencia al cambio . . . . .	72
Código limpio . . . . .	72
Automatizar lo posible . . . . .	73
Pero a mí no me interesa . . . . .	73
Conclusión . . . . .	76
<b>Dimensión Profesional</b>	<b>77</b>
<b>El "celodurismo" de los programadores</b>	<b>78</b>
Las raíces históricas . . . . .	79
El manifiesto del celodurismo . . . . .	80
1. El IDE no sirve para nada . . . . .	80
2. La CLI es la solución a todos los problemas . . . . .	82
3. No usamos Windows porque los programadores usan Linux (o MacOS) . . . . .	83
4. El depurador es para los débiles . . . . .	84
5. El código está todo en mi cabeza . . . . .	85
6. Los programadores no usan ChatGPT . . . . .	86
Conclusión . . . . .	88
<b>Los programadores son los nuevos mercenarios: la evolución del trabajo en el sector IT</b>	<b>89</b>
De centralizados a descentralizados: un nuevo paradigma . . . . .	90
De los años '80 a hoy: un viaje en el tiempo del trabajo IT . . . . .	91
La disparidad entre grandes y pequeñas empresas: un abismo digital . . . . .	93
¿Cómo enfrentar estos desafíos? Estrategias para un nuevo mundo del trabajo . . . . .	94
Conclusiones: navegar en el nuevo mundo del trabajo IT . . . . .	95
<b>Un mejor salario no es suficiente para motivar un cambio</b>	<b>97</b>

Fantástico, pero ¿dónde está el problema? . . . . .	99
Cuando el salario no es suficiente . . . . .	101
Los errores que cometí . . . . .	103
Conclusiones . . . . .	104
<b>Quien pierde un programador pierde un tesoro</b>	<b>106</b>
La vida profesional de un programador . . . . .	107
La pérdida de estímulos . . . . .	108
La pérdida de un programador . . . . .	109
Cómo evitar la pérdida de un programador . . . . .	111
Conclusiones . . . . .	112
<b>¿Y si el eslabón débil de la programación fuera el propio programador?</b> <sup>114</sup>	
Un poco de historia . . . . .	115
El creciente peso de la complejidad . . . . .	116
La ilusión de la compresión temporal . . . . .	116
¿Pero cómo nos están ayudando y evolucionando las IA? . . . . .	117
Cuando el programador se detiene: escenario futuro próximo . . . . .	118
El futuro del programador: más allá del presente inmediato . . . . .	119
La mente humana: el límite infranqueable . . . . .	120
Conclusiones: el eslabón débil que puede romper la cadena . . . . .	120
<b>Por qué las startups tech están condenadas a morir</b>	<b>122</b>
Bienvenidos al matadero de las buenas intenciones . . . . .	123
Anatomía de un fracaso anunciado: la plaga mortal de las startups "Tech-Only"	124
Ser irrelevantes (No Market Need): crear soluciones que nadie necesita . . . . .	125
Agotar los fondos (Ran out of Cash) pero estar llenos de código . . . . .	126
La Plaga de la Invisibilidad (Poor Marketing) . . . . .	129
Equipo equivocado (Not the Right Team) . . . . .	130
Caso de Estudio: Stripe – Los Programadores que Aprendieron a Vender . . . . .	130
Dejad de Escribir Código, Empezad a Vender . . . . .	131
<b>Biografía</b>	<b>133</b>

