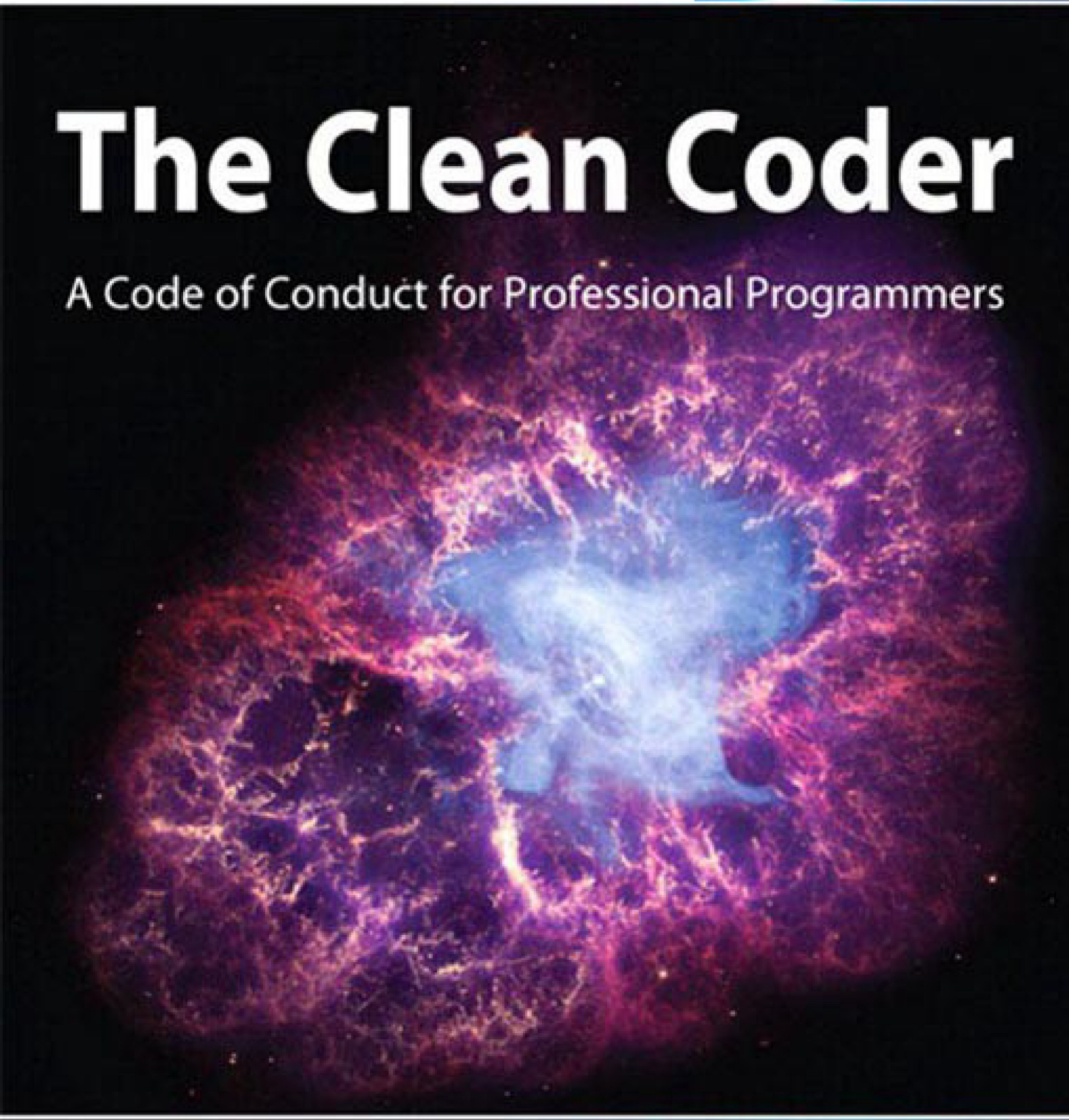




The Clean Coder

A Code of Conduct for Professional Programmers



Foreword by Matthew Heusser, Software Process Naturalist

Robert C. Martin

El codificador limpio

Código de conducta para programadores profesionales

Robert C. Martin



Upper Saddle River, NJ - Boston - Indianápolis - San Francisco
Nueva York - Toronto - Montreal - Londres - Múnich - París -
Madrid Ciudad del Cabo - Sidney - Tokio - Singapur - Ciudad de
México

Muchas de las denominaciones utilizadas por los fabricantes y vendedores para distinguir sus productos se reclaman como marcas comerciales. En los casos en que esas denominaciones aparecen en este libro, y el editor tenía conocimiento de una reclamación de marca, las denominaciones se han impreso con letras mayúsculas iniciales o en mayúsculas.

El autor y el editor han tenido cuidado en la preparación de este libro, pero no ofrecen ninguna garantía expresa o implícita de ningún tipo y no asumen ninguna responsabilidad por errores u omisiones. No se asume ninguna responsabilidad por daños incidentales o consecuentes en relación con el uso de la información o los programas contenidos en este libro.

La editorial ofrece excelentes descuentos en este libro cuando se pide en cantidad para compras al por mayor o ventas especiales, que pueden incluir versiones electrónicas y/o cubiertas personalizadas y contenidos particulares para su negocio, objetivos de formación, enfoque de marketing e intereses de marca. Para más información, póngase en contacto con

Ventas a empresas y gobiernos de
Estados Unidos (800) 382-3419
corpsales@pearsontechgroup.com

Para ventas fuera de Estados Unidos, póngase en contacto con

Ventas internacionales
international@pearson.com

Visítenos en la web: www.informit.com/ph

Datos de catalogación de la Biblioteca del Congreso

Martin, Robert C.

El programador limpio: un código de conducta para programadores profesionales / Robert Martin.

p. cm.

Incluye referencias bibliográficas e índice.

ISBN 0-13-708107-3 (pbk. : alk. paper)

1. Programación informática-Aspectos morales y éticos. 2. Programadores informáticos-Etica profesional. I. Título.

QA76.9.M65M367 2011

005.1092—dc222011005962

Copyright © 2011 Pearson Education, Inc.

Ilustraciones copyright 2011 de

JenniferKohnke.

Todos los derechos reservados. Impreso en los Estados Unidos de América. Esta publicación está protegida por derechos de autor, y debe obtenerse el permiso del editor antes de cualquier reproducción prohibida, almacenamiento en un sistema de recuperación o transmisión en cualquier forma o por cualquier medio, electrónico, mecánico, de fotocopia, de grabación o similar. Para obtener el permiso para utilizar el material de esta obra, envíe una solicitud por escrito a Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, o puede enviar su solicitud por fax al (201) 236-3290.

ISBN-13: 978-0-13-708107-3

ISBN-10:0-13-708107-3

Texto impreso en Estados Unidos en papel reciclado en RR Donnelley en Crawfordsville, Indiana.

Sexta impresión, julio de 2015

Entre 1986 y 2000 trabajé estrechamente con Jim Newkirk, un colega de Teradyne. Él y yo compartíamos la pasión por la programación y el código limpio. Pasábamos juntos noches, tardes y fines de semana jugando con diferentes estilos de programación y técnicas de diseño. Continuamente estábamos maquinando ideas de negocio.

Con el tiempo, formamos juntos Object Mentor, Inc. Aprendí muchas cosas de Jim mientras trabajábamos juntos. Pero una de las más importantes fue su actitud de *ética de trabajo*; fue algo que me esforcé por emular. Jim es un profesional. Estoy orgulloso de haber trabajado con él y de llamarle amigo.

Contenido

Prólogo

Prólogo

Agradecimientos

Sobre el autor En

la portada

Introducción previa a los requisitos

Capítulo 1. Profesionalidad

Cuidado con lo que pides

Asumir la responsabilidad

Primero, no dañar

la ética laboral

Bibliografía

Capítulo 2. Decir que

no Roles

adversarios Lo que

está en juego

Ser un "jugador de

equipo" El coste de decir

sí al código imposible

Capítulo 3. Decir sí

Un lenguaje de compromiso

Aprender a decir "sí"

Conclusión

Capítulo 4. Codificación

Preparación

La zona de flujo

Bloqueo del

escritor Depurar

el ritmo de

trabajo Llegar

tarde Ayuda

Bibliográfica

Capítulo 5. Desarrollo dirigido por pruebas

El jurado está presente

Las tres leyes del TDD Lo

que el TDD no es

Bibliografía

Capítulo 6. Practicando Algunos

antecedentes de la práctica de

The Coding DojoAmpliación

de la experiencia Conclusión

Bibliografía

Capítulo 7. Pruebas de

aceptación Pruebas de

aceptación Comunicación de

los requisitos Pruebas de

aceptación

Conclusión:

Capítulo 8. Estrategias de prueba

que la GC no debe

encontrar nada La

pirámidede la

automatización de pruebas

Conclusión

Bibliografía

Capítulo 9. Gestión del tiempo

Reuniones

Focus-Manna

Boxeo de tiempo y evasión

de tomates

Callejones sin salida

Marismas, ciénagas, pantanos y otros líos

Conclusión

Capítulo 10. Estimación

¿Qué es una

estimación? PERT

Tareas de estimación

La ley de los grandes

números Conclusión

Bibliografía

Capítulo 11. Evitar la

presión Manejar la

presión

Conclusión

Capítulo 12. Colaboración de

los programadores frente a

los cerebros de las personas

Conclusión:

Capítulo 13. Equipos y proyectos

¿Se mezcla?

Conclusión

ión

Bibliografía

Capítulo 14. Tutoría, aprendizaje y artesanía

Grados de fracaso

Tutoría

Aprendizaje

Artesanía

Conclusión

Apéndice A. Herramientas

Herramientas

Control de código

fuentes IDE/Editor

[Seguimiento de](#)
[problemas](#)
[Construcción](#)
[continua](#)
[Herramientas de](#)
[pruebas unitarias](#)
[Herramientas de prueba de](#)
[_____componentes](#)
[Herramientas de prueba de](#)
[_____integración](#)
[UML/MDA](#)
[Conclusión:](#)

[Índice](#)

Elogios para *The Clean Coder*

"El tío Bob" Martin sube definitivamente el listón con su último libro. Explica sus expectativas para un programador profesional sobre las interacciones de la dirección, la gestión del tiempo, la presión, sobre la colaboración y sobre la elección de las herramientas a utilizar. Más allá de TDD y ATDD, Martin explica lo que todo programador que se considere profesional no sólo debe saber, sino que debe seguir para hacer crecer la joven profesión del desarrollo de software."

-Markus Gärtner *Desarrollador Senior de Software* it-agileGmbH www.it-agile.de www.shino.de

"Algunos libros técnicos inspiran y enseñan; otros deleitan y divierten. Rara vez un libro técnico hace estas cuatro cosas. Los de Robert Martin siempre lo han hecho para mí y *The Clean Coder* no es una excepción. Lee, aprende y vive las lecciones de este libro y podrás llamarte con precisión un profesional del software."

-George Bullock *Director de programas de Microsoft Corp.*

Si una carrera de informática tuviera "lecturas obligatorias para después de graduarse", sería ésta. En el mundo real, el código defectuoso no desaparece cuando se acaba el semestre, no se obtiene sobresaliente por una codificación maratónica la noche anterior a la entrega de un trabajo y, lo peor de todo, hay que tratar con la gente. Así que los gurús de la codificación no son necesariamente profesionales.

The Clean Coder describe el camino hacia la profesionalidad... y lo hace de forma notablemente entretenida".

-Jeff Overbey Universidad de Illinois en Urbana-Champaign

"The Clean Coder" es mucho más que un conjunto de reglas o directrices. Contiene sabiduría y conocimientos duramente ganados que normalmente se obtienen a través de muchos años de prueba y error o trabajando como aprendiz de un maestro artesano. Si te consideras un profesional del software, necesitas este libro".

-R. L. Bogetti Diseñador jefe de sistemas Baxter Healthcare

www.RLBogetti.com

Prólogo

Usted ha escogido este libro, así que asumo que es un profesional del software. Eso es bueno; yo también lo soy. Y ya que tengo tu atención, déjame contarte por qué elegí este libro.

Todo comienza hace poco tiempo en un lugar no muy lejano. Se abre el telón, las luces y la cámara, Charley

Hace varios años trabajaba en una empresa de tamaño medio que vendía productos muy regulados. Ya sabes el tipo; nos sentábamos en una granja de cubículos en un edificio de tres plantas, los directores y los superiores tenían despachos privados, y conseguir que todos los que necesitabas estuvieran en la misma sala para una reunión llevaba una semana o más.

Estábamos operando en un mercado muy competitivo cuando el gobierno abrió un nuevo producto.

De repente, teníamos un conjunto completamente nuevo de clientes potenciales; todo lo que teníamos que hacer era conseguir que compraran nuestro producto. Eso significaba que teníamos que presentar la solicitud en un plazo determinado ante el gobierno federal, pasar una auditoría de evaluación en otra fecha y salir al mercado en una tercera fecha.

Una y otra vez, nuestros directivos nos recalcaron la importancia de esas fechas. Un solo desliz y el gobierno nos dejaría fuera del mercado durante un año, y si los clientes no podían apuntarse el primer día, entonces todos se apuntarían con otro y nos quedaríamos sin negocio.

Era el tipo de ambiente en el que algunos se quejan y otros señalan que "la presión hace diamantes".

Yo era director de proyectos técnicos, ascendido desde el departamento de desarrollo. Mi responsabilidad era poner en marcha el sitio web el día de la puesta en marcha, para que los clientes potenciales pudieran descargar la

información y, sobre todo, los formularios de inscripción. Mi compañero de fatigas era el gestor de proyectos de cara a la empresa, al que llamaré Joe. El papel de Joe era trabajar del otro lado, ocupándose de las ventas, el marketing y los requisitos no técnicos. También era el hombre al que le gustaba el comentario de "la presión crea diamantes".

Si usted ha trabajado mucho en la América corporativa, probablemente ha visto el señalamiento con el dedo, la culpabilización y la aversión al trabajo que es completamente

natural. Nuestra empresa tuvo una interesante solución a ese problema con Joe y conmigo.

Un poco como Batman y Robin, nuestro trabajo era hacer las cosas. Me reunía con el equipo técnico todos los días en un rincón; reconstruíamos el calendario cada día, averiguábamos la ruta crítica y eliminábamos todos los obstáculos posibles de esa ruta crítica. Si alguien necesitaba software, íbamos a buscarlo. Si les "encantaría" configurar el cortafuegos pero "caramba, es la hora de mi almuerzo", les invitaríamos a comer. Si alguien quería trabajar en nuestro ticket de configuración pero tenía otras prioridades, Joe y yo íbamos a hablar con el supervisor.

Luego el gerente.

Luego el director.

Conseguimos hacer las cosas.

Es un poco exagerado decir que pateamos sillas, gritamos y chillamos, pero utilizamos todas las técnicas de nuestra bolsa para hacer las cosas, inventamos algunas nuevas por el camino, y lo hicimos de una manera ética de la que estoy orgulloso hasta el día de hoy.

Me consideraba un miembro del equipo, no por encima de él para escribir una sentencia SQL o hacer un pequeño emparejamiento para sacar el código. En ese momento, pensaba en Joe de la misma manera, como un miembro del equipo, no por encima de él.

Con el tiempo me di cuenta de que Joe no compartía esa opinión. Ese fue un día muy triste para mí.

Fue el viernes a la 1:00 PM; el sitio web se puso en marcha muy temprano el lunes siguiente.

Habíamos terminado. *HECHO*. Todos los sistemas estaban listos; estábamos preparados. Tenía a todo el equipo técnico reunido para la reunión final de scrum y estábamos listos para dar el salto. Más que "sólo" el equipo técnico, teníamos a la gente de negocios de marketing, los propietarios del producto, con nosotros.

Estábamos orgullosos. Fue un buen

momento. Entonces Joe se pasó por aquí.

Dijo algo así como: "Malas noticias. El departamento jurídico no tiene los formularios de inscripción, así que aún no podemos salir a la luz".

No era gran cosa; habíamos estado retrasados por una cosa u otra durante todo el proyecto y teníamos la rutina de Batman/Robin bien aprendida. Yo estaba preparado, y mi respuesta fue básicamente: "Muy bien, compañero, hagamos esto una vez más. El departamento legal está en el tercer piso, ¿verdad?"

Entonces las cosas se pusieron raras.

En lugar de darme la razón, Joe preguntó: "¿De qué estás hablando, Matt?".

Dije: "Ya sabes. Nuestra canción y baile habitual. Estamos hablando de cuatro archivos PDF, ¿verdad? Que están hechos; ¿el departamento legal sólo tiene que aprobarlos? Vayamos a sus cubículos, echémosles el ojo y *acabemos con esto*".

Joe no estuvo de acuerdo con mi apreciación y respondió: "Saldremos a la luz a finales de la semana que viene. No es gran cosa".

Probablemente puedas adivinar el resto del intercambio; sonó más o menos así:

Matt: "¿Pero por qué? Podrían hacerlo en un par de horas".

Joe: "Podría llevar más que eso".

Matt: "Pero tienen *todo el fin de semana*. Mucho tiempo. Hagamos esto".

Joe: "Matt, estos son profesionales. No podemos mirarlos fijamente e insistir en que sacrifiquen su vida personal por nuestro pequeño proyecto".

Matt: (pausa) ". . . Joe. . . ¿qué crees que hemos estado haciendo con el equipo de ingeniería durante los últimos cuatro meses?"

Joe: "Sí, pero estos son profesionales".

Pausa.

Respira.

Qué. Lo hizo. Joe. Sólo. ¿Dijo?

En ese momento, pensé que el personal técnico era profesional, en el mejor sentido de la palabra.

Sin embargo, volviendo a pensar en ello, no estoy tan seguro.

Veamos esa técnica de Batman y Robin por segunda vez, desde una perspectiva diferente. Pensaba que estaba exhortando al equipo a su mejor

rendimiento, pero sospecho que Joe estaba jugando, con la suposición

implícita de que el cuerpo técnico era su oponente. Piensa en ello: ¿Por qué era necesario correr de un lado a otro, pateando sillas y apoyándose en la gente?

¿No deberíamos haber podido preguntar al personal cuándo iban a terminar, obtener una respuesta firme, creer la respuesta que nos dieron y no quemarnos por esa creencia?

Ciertamente, para los profesionales, deberíamos... y, al mismo tiempo, no podríamos. Joe no confiaba en nuestras respuestas y se sentía cómodo microgestionando al equipo técnico, y al mismo tiempo, por alguna razón, sí confiaba en el equipo jurídico y no estaba dispuesto a microgestionarlo.

¿De qué se trata?

De alguna manera, el equipo jurídico ha demostrado su profesionalidad de una forma que el equipo técnico no ha hecho.

De alguna manera, otro grupo había convencido a Joe de que no necesitaban una niñera, de que no estaban jugando y de que debían ser tratados como compañeros que eran respetados.

No, no creo que tenga nada que ver con los lujosos certificados que cuelgan de las paredes ni con unos cuantos años más de universidad, aunque esos años de universidad pueden haber incluido una buena dosis de deformación social implícita sobre cómo comportarse.

Desde aquel día, hace ya tantos años, me he preguntado cómo tendría que cambiar la profesión de técnico para que se le considere profesional.

Tengo algunas ideas. He escrito un poco en el blog, he leído mucho, he conseguido mejorar mi propia situación de vida laboral y ayudar a algunos otros. Sin embargo, no conocía ningún libro que expusiera un plan, que explicitara todo el asunto.

Entonces, un día, de la nada, recibí una oferta para revisar un primer borrador de un libro; el libro que tienes en tus manos ahora mismo.

Este libro le dirá paso a paso cómo presentarse e interactuar como un profesional. No con tópicos trillados, ni con apelaciones a trozos de papel, sino lo que puedes hacer y cómo hacerlo.

En algunos casos, los ejemplos son palabra por palabra.

Algunos de esos ejemplos tienen respuestas, contrarréplicas, aclaraciones, incluso consejos sobre qué hacer si la otra persona intenta "simplemente ignorarte".

Oye, mira eso, aquí viene Joe de nuevo, esta vez a la izquierda del escenario:

Oh, aquí estamos, de vuelta en BigCo, con Joe y yo, una vez más en el gran proyecto de conversión del sitio web.

Sólo que esta vez, imagínatelo un poco diferente.

En lugar de eludir los compromisos, el equipo técnico los asume realmente. En lugar de eludir las estimaciones o dejar que otro haga la planificación (y luego quejarse de ello), el equipo técnico se autoorganiza y asume compromisos reales.

Ahora imagina que el personal trabaja realmente en conjunto. Cuando los programadores están bloqueados por las operaciones, cogen el teléfono y el administrador del sistema se pone realmente a trabajar.

Cuando Joe se acerca a encender un fuego para que trabajen en el ticket 14321, no lo necesita; puede ver que el DBA está trabajando diligentemente, no navegando por la web. Del mismo modo, las estimaciones que recibe del personal le parecen francamente coherentes, y no tiene la sensación de que el proyecto tiene prioridad entre el almuerzo y la revisión del correo electrónico. Todos los trucos e intentos de manipular el calendario no se responden con un "lo intentaremos", sino con un "ese es nuestro compromiso; si quieres inventarte tus propios objetivos, siéntete libre".

Después de un tiempo, sospecho que Joe empezaría a considerar al equipo técnico como, bueno, profesionales. Y tendría razón.

¿Esos pasos para transformar tu comportamiento de técnico a profesional? Los encontrarás en el resto del libro.

Bienvenido al siguiente paso en tu carrera; sospecho que te va a gustar.

-Matthew Heusser

Software Process

Naturalist

Prefacio



A las 11:39 AM EST del 28 de enero de 1986, sólo 73,124 segundos después del lanzamiento y a una altitud de 48.000 pies, el transbordador espacial

Challenger se hizo añicos por el fallo del cohete impulsor sólido (SRB) derecho.

Siete valientes astronautas, incluida la profesora de secundaria Christa McAuliffe, se perdieron. La expresión de la cara de la madre de McAuliffe al ver la muerte de su hija a nueve millas de altura me persigue hasta el día de hoy.

El Challenger se rompió porque los gases de escape calientes del SRB, que estaba fallando, se filtraron por entre los segmentos de su casco, salpicando el cuerpo del tanque de combustible externo. La parte inferior del tanque principal de hidrógeno líquido estalló, encendiendo el combustible y haciendo que el tanque avanzara hasta chocar con el tanque de oxígeno líquido situado encima. Al mismo tiempo, el SRB se desprendió de su puntal de popa y giró alrededor de su puntal delantero. Su nariz perforó el tanque de oxígeno líquido. Estos vectores de fuerza aberrantes hicieron que toda la nave, que se movía muy por encima de mach 1,5, girara contra la corriente de aire. Las fuerzas aerodinámicas lo destrozaron todo rápidamente.

Entre los segmentos circulares del SRB había dos juntas tóricas concéntricas de goma sintética. Cuando los segmentos se atornillaron, las juntas tóricas se comprimieron, formando un sello hermético que los gases de escape no deberían haber podido penetrar.

Sin embargo, en la víspera del lanzamiento, la temperatura en la plataforma de lanzamiento descendió a 17°F, 23 grados por debajo de la temperatura mínima especificada para las juntas tóricas y 33 grados menos que en cualquier lanzamiento anterior. Como resultado, las juntas tóricas se volvieron demasiado rígidas para bloquear adecuadamente los gases calientes. Al encender el SRB se produjo un pulso de presión al acumularse rápidamente los gases calientes. Los segmentos del propulsor se hincharon y relajaron la compresión de las juntas tóricas. La rigidez de las juntas tóricas impidió que mantuvieran la estanqueidad, por lo que algunos de los gases calientes se filtraron y vaporizaron las juntas tóricas a lo largo de 70 grados de arco.

Los ingenieros de Morton Thiokol que diseñaron el SRB sabían que había problemas con las juntas tóricas, y habían informado de ellos a los directivos de Morton Thiokol y de la NASA siete años antes. De hecho, las juntas tóricas de los lanzamientos anteriores se habían dañado de forma similar, aunque no lo suficiente como para ser catastróficas. El lanzamiento más frío fue el que más daños sufrió. Los ingenieros habían diseñado una reparación para el problema, pero su aplicación se había retrasado mucho.

Los ingenieros sospechaban que las juntas tóricas se endurecían en frío. También sabían que las temperaturas para el lanzamiento del Challenger eran más frías que las de cualquier lanzamiento anterior y muy por debajo de la línea roja. En resumen, los ingenieros *sabían* que el riesgo era

demasiado alto. Los ingenieros actuaron en base a ese conocimiento. Redactaron memorandos en los que se alertaba de la existencia de grandes riesgos. Instaron enérgicamente a los directivos de Thiokol y de la NASA a no realizar el lanzamiento. En una reunión de última hora celebrada pocas horas antes del lanzamiento, esos ingenieros presentaron sus mejores datos. Se enfadaron, engatusaron y protestaron. Pero al final, los directivos les ignoraron.

Cuando llegó el momento del lanzamiento, algunos de los ingenieros se negaron a ver la transmisión porque temían una explosión en la plataforma. Pero a medida que el Challenger ascendía grácilmente hacia el cielo, empezaron a relajarse. Momentos antes de la destrucción, mientras veían al vehículo pasar por Mach 1, uno de ellos dijo que habían "esquivado una bala".

A pesar de todas las protestas y los memorandos, y de las peticiones de los ingenieros, los directivos creían que sabían más. Pensaban que los

ingenieros estaban exagerando. No confiaban en los datos de los ingenieros ni en sus conclusiones. Se lanzaron porque estaban bajo una inmensa presión financiera y política. *Esperaban* que todo fuera bien.

Estos gestores no fueron simplemente insensatos, fueron criminales. Las vidas de siete buenos hombres y mujeres, y las esperanzas de una generación que miraba hacia

Los viajes espaciales, se vieron truncados en aquella fría mañana porque aquellos directivos antepusieron sus propios miedos, esperanzas e intuiciones a las palabras de sus expertos. Tomaron una decisión que no tenían derecho a tomar. Usurparon la autoridad de las personas que realmente *sabían*: los ingenieros.

¿Pero qué pasa con los ingenieros? Ciertamente, los ingenieros hicieron lo que debían hacer. Informaron a sus jefes y lucharon con ahínco por su posición. Recurrieron a los canales adecuados e invocaron todos los protocolos correctos. Hicieron lo que pudieron, *dentro* del sistema, y aun así los directivos les pasaron por encima. Así que parece que los ingenieros pueden salir de rositas.

Pero a veces me pregunto si alguno de esos ingenieros se ha quedado despierto por la noche, atormentado por esa imagen de la madre de Christa McAuliffe, y deseando haber llamado a Dan Rather.

Sobre este libro

Este libro trata sobre la profesionalidad del software. Contiene muchos consejos pragmáticos en un intento de responder a preguntas como

- ¿Qué es un profesional del software?
- ¿Cómo se comporta un profesional?

- ¿Cómo puede un profesional enfrentarse a los conflictos, a las agendas apretadas y a los jefes poco razonables?
- ¿Cuándo y cómo debe un profesional decir "no"?
- ¿Cómo se enfrenta un profesional a la presión?

Pero dentro de los consejos pragmáticos de este libro encontrarás una actitud que lucha por abrirse paso. Es una actitud de honestidad, de honor, de respeto a sí mismo y de orgullo. Es la voluntad de aceptar la terrible responsabilidad de ser un artesano y un ingeniero. Esa responsabilidad incluye trabajar bien y trabajar limpio. Incluye comunicar bien y estimar fielmente. Incluye la gestión del tiempo y la toma de decisiones difíciles con respecto a los riesgos.

Pero esa responsabilidad incluye otra cosa, una cosa aterradora. Como ingeniero, tiene un conocimiento profundo de sus sistemas y proyectos que ningún gerente puede tener. Ese conocimiento conlleva la responsabilidad de *actuar*.

Bibliografía

[McConnell87]: Malcolm McConnell, *Challenger 'A Major Malfunction'*, Nueva York, NY: Simon & Schuster, 1987

[Wiki-Challenger]: "El desastre del transbordador espacial Challenger", http://en.wikipedia.org/wiki/Space_Shuttle_Challenger_disaster

Agradecimientos

Mi carrera ha sido una serie de colaboraciones y planes. Aunque he tenido muchos sueños y aspiraciones privadas, siempre parecía encontrar a alguien con quien compartirlos. En ese sentido me siento un poco como los Sith: "Siempre hay dos".

La primera colaboración que podría considerar profesional fue con John Marchese a la edad de 13 años. Él y yo planeamos construir ordenadores juntos. Yo era el cerebro y él el músculo. Le mostré dónde soldar un cable y él lo soldó. Le enseñé dónde montar un relé y él lo montó. Fue muy divertido y pasamos cientos de horas en ello. De hecho, construimos unos cuantos objetos de aspecto impresionante con relés, botones, luces, ¡incluso teletipos! Por supuesto, ninguno de ellos hacía realmente nada, pero eran muy impresionantes y trabajamos mucho en ellos. A John: ¡Gracias!

En mi primer año de instituto conocí a Tim Conrad en mi clase de alemán. Tim era *inteligente*. Cuando nos asociamos para construir un ordenador, él era el cerebro y yo la fuerza. Me enseñó electrónica y me dio mi primera introducción a un PDP-8. Él y yo construimos una calculadora electrónica de 18 bits con componentes básicos. Podía sumar, restar, multiplicar y dividir. Nos llevó un año de fines de semana y todas las vacaciones de

primavera, verano y Navidad. Trabajamos intensamente en ella. Al final, funcionó muy bien. A Tim: ¡Gracias!

Tim y yo aprendimos a programar ordenadores. No era fácil hacerlo en 1968, pero lo conseguimos. Conseguimos libros sobre el ensamblador del PDP-8, Fortran, Cobol y PL/1, entre otros. Los devoramos. Escribimos programas que no teníamos ninguna esperanza de ejecutar porque no teníamos acceso a un ordenador. Pero los escribimos de todos modos por puro amor.

Nuestra escuela secundaria comenzó un plan de estudios de informática en nuestro segundo año. Conectaron un teletipo ASR-33 a un módem de 110 baudios. Tenían una cuenta en el sistema de tiempo compartido Univac 1108 del Instituto Tecnológico de Illinois. Tim y yo nos convertimos inmediatamente en los operadores de facto de esa máquina. Nadie más podía acercarse a ella.

El módem se conectaba descolgando el teléfono y marcando el número. Cuando se oía el chirrido del módem que respondía, se pulsaba el botón "orig" del teletipo haciendo que el módem de origen emitiera su propio chirrido. Luego colgabas el teléfono y se establecía la conexión de datos.

El teléfono tenía un candado en el dial. Sólo los profesores tenían la llave. Pero eso no importaba, porque aprendimos que se podía marcar un teléfono (cualquier teléfono) tocando el número de teléfono en el gancho del interruptor. Yo era baterista, así que tenía muy buena sincronización y reflejos. Podía marcar ese módem, con el candado puesto, en menos de 10 segundos.

Teníamos dos teletipos en el laboratorio de informática. Uno era la máquina online y el otro era una máquina offline. Ambos eran utilizados por los estudiantes para escribir sus programas. Los estudiantes escribían sus programas en los teletipos con la perforación de la cinta de papel activada. Cada pulsación se grababa en la cinta. Los estudiantes escribían sus programas en IITran, un lenguaje interpretado muy potente. Los estudiantes dejaban sus cintas de papel en una cesta cerca de los teletipos.

Después de las clases, Tim y yo marcábamos el ordenador (a golpes, por supuesto), cargábamos las cintas en el sistema de lotes IITran y luego colgábamos. A 10 caracteres por segundo, no era un procedimiento rápido. Una hora más tarde, volvíamos a llamar y obteníamos las impresiones, de nuevo a 10 caracteres por segundo. El teletipo no separaba los listados de los alumnos expulsando las páginas. Se imprimía una tras otra, así que las separábamos con unas tijeras, pegábamos la cinta de papel de entrada a su listado y las colocábamos en la cesta de salida.

Tim y yo éramos los amos y dioses de ese proceso. Incluso los profesores nos dejaban tranquilos cuando estábamos en esa sala. Estábamos haciendo

su trabajo, y ellos lo sabían. Nunca nos pidieron que lo hiciéramos. Nunca nos dijeron que podíamos hacerlo. Nunca nos dieron la llave del teléfono. Simplemente nos mudamos, y ellos se fueron
-y nos dieron una correa muy larga. A mis profesores de matemáticas, el Sr. McDermit, el Sr. Fogel y el Sr. Robien: ¡gracias!

Luego, una vez terminados los deberes de los estudiantes, nos poníamos a jugar. Escribimos un programa tras otro para hacer cualquier cantidad de cosas locas y extrañas. Escribimos programas que graficaban círculos y parábolas en ASCII en un teletipo. Escribimos programas de paseo

aleatorio y generadores de palabras aleatorias. Hemos escrito
calculó el factorial 50 hasta el último dígito. Nos pasamos horas y horas inventando programas para escribir y luego conseguir que funcionaran.

Dos años después, Tim, nuestro compadre Richard Lloyd y yo fuimos contratados como programadores en ASC Tabulating en Lake Bluff, Illinois. Tim y yo teníamos entonces 18 años. Habíamos decidido que la universidad era una pérdida de tiempo y que debíamos empezar nuestras carreras inmediatamente. Fue aquí donde conocimos a Bill Hohri, Frank Ryder, Big Jim Carlin y John Miller. Ellos dieron a algunos jóvenes la oportunidad de aprender lo que era la programación profesional. La experiencia no fue del todo positiva ni negativa. Fue ciertamente educativa. A todos ellos, y a Richard que catalizó e impulsó gran parte de ese proceso: Gracias.

Después de dejarlo y fundirme a los 20 años, hice una temporada como reparador de cortacéspedes trabajando para mi cuñado. Lo hacía tan mal que tuvo que despedirme. ¡Gracias, Wes!

Un año después, más o menos, acabé trabajando en Outboard Marine Corporation. Para entonces ya estaba casado y tenía un bebé en camino. También me despidieron.
Gracias, John, Ralph y Tom.

Luego me fui a trabajar a Teradyne, donde conocí a Russ Ashdown, Ken Finder, Bob Copithorne, Chuck Studee y CK Srithran (ahora Kris Iyer). Ken era mi jefe. Chuck y CK eran mis compañeros. Aprendí mucho de todos ellos. Gracias, chicos.

Luego estaba Mike Carew. En Teradyne, él y yo nos convertimos en el dúo dinámico. Escribimos varios sistemas juntos. Si querías hacer algo, y hacerlo rápido, tenías a Bob y a Mike para hacerlo. Nos divertimos mucho juntos. Gracias, Mike.

Jerry Fitzpatrick también trabajaba en Teradyne. Nos conocimos jugando juntos a Dragones y Mazmorras, pero rápidamente formamos una colaboración. Escribimos un software en un Commodore 64 para ayudar a los usuarios de D&D. También iniciamos un nuevo proyecto en Teradyne

llamado "La recepcionista electrónica". Trabajamos juntos durante varios años, y él se convirtió, y sigue siendo, un gran amigo. ¡Gracias, Jerry!

Pasé un año en Inglaterra mientras trabajaba para Teradyne. Allí me asocié con Mike Kergozou. Él y yo planeamos juntos todo tipo de cosas, aunque la mayoría de esos planes tenían que ver con bicicletas y pubs. Pero era un

programador dedicado y muy centrado en la calidad y la disciplina (aunque, tal vez, él no estaría de acuerdo). Gracias, Mike.

Al volver de Inglaterra en 1987, empecé a maquinar con Jim Newkirk. Ambos dejamos Teradyne (con meses de diferencia) y nos unimos a una empresa emergente llamada Clear Communications. Allí pasamos varios años trabajando juntos para conseguir los millones que nunca llegaron.

Pero seguimos maquinando. Gracias, Jim.

Al final fundamos juntos Object Mentor. Jim es la persona más directa, disciplinada y centrada con la que he tenido el privilegio de trabajar. Me enseñó tantas cosas que no puedo enumerarlas aquí. En cambio, le he dedicado este libro.

Hay tantos otros con los que he maquinado, tantos otros con los que he colaborado, tantos otros que han tenido un impacto en mi vida profesional: Lowell Lindstrom, Dave Thomas, Michael Feathers, Bob Koss, Brett Schuchert, Dean Wampler, Pascal Roy, Jeff Langr, James Grenning, Brian Button, Alan Francis, Mike Hill, Eric Meade, Ron Jeffries, Kent Beck, Martin Fowler, Grady Booch, y una lista interminable de otros.

Gracias a todos.

Por supuesto, la mayor colaboradora de mi vida ha sido mi encantadora esposa, Ann Marie. Me casé con ella cuando tenía 20 años, tres días después de que ella cumpliera los 18. Durante 38 años ha sido mi compañera constante, mi timón y mi vela, mi amor y mi vida. Espero poder pasar otras cuatro décadas con ella.

Y ahora, mis colaboradores y socios intrigantes son mis hijos. Trabajo estrechamente con mi hija mayor, Angela, mi encantadora madre gallina e intrépida asistente. Ella me mantiene en el camino correcto y nunca me deja olvidar una fecha o un compromiso. Planifico los planes de negocio con mi hijo Micah, el fundador de 8thlight.com. Su cabeza para los negocios es mucho mejor que la mía. Nuestra última aventura, cleancoders.com, es muy emocionante.

Mi hijo menor, Justin, acaba de empezar a trabajar con Micah en 8th Light. Mi hija menor, Gina, es ingeniera química y trabaja en Honeywell. Con ellos dos, ¡los planes serios acaban de empezar!

Nadie en tu vida te enseñará más que tus hijos. ¡Gracias, niños!

Sobre el autor



Robert C. Martin ("Tío Bob") es programador desde 1970. Es fundador y presidente de Object Mentor, Inc., una empresa internacional de desarrolladores y gestores de software con gran experiencia, especializada en ayudar a las empresas a realizar sus proyectos. Object Mentor ofrece servicios de consultoría de mejora de procesos, consultoría de diseño de software orientado a objetos, formación y desarrollo de habilidades a las principales empresas del mundo.

Martin ha publicado decenas de artículos en diversas revistas especializadas y es ponente habitual en conferencias y ferias internacionales.

Es autor y editor de numerosos libros, entre ellos:

- *Diseño de aplicaciones C++ orientadas a objetos mediante el método Booch*
- *Patrones Lenguajes de Diseño de Programas 3*
- *Más joyas de C++*
- *La programación extrema en la práctica*
- *Desarrollo ágil de software: Principios, patrones y prácticas*
- *UML para programadores de Java*
- *Código limpio*

Líder en la industria del desarrollo de software, Martin fue durante tres años redactor jefe del *C++ Report*, y fue el primer presidente de la Agile Alliance.

Robert es también el fundador de Uncle Bob Consulting, LLC, y cofundador con su hijo Micah Martin de The Clean Coders LLC.

En la portada



La impresionante imagen de la portada, que recuerda al ojo de Sauron, es M1, la nebulosa del Cangrejo. M1 está situada en Tauro, aproximadamente un grado a la derecha de Zeta Tauri, la estrella que se encuentra en la punta del cuerno izquierdo del toro. La nebulosa del Cangrejo es el remanente de una supernova que voló sus entrañas por todo el cielo en la fecha más bien propicia del 4 de julio de 1054 d.C. A una distancia de 6.500 años luz, esa explosión apareció a los observadores chinos como una nueva estrella, más o menos tan brillante como Júpiter. De hecho, ¡era visible *durante el día*! A lo largo de los seis meses siguientes, desapareció lentamente de la vista.

La imagen de la portada es un compuesto de luz visible y de rayos X. La imagen visible fue tomada por el telescopio Hubble y forma la envoltura exterior. El objeto interior, que parece una diana de tiro con arco azul, fue tomado por el telescopio de rayos X Chandra.

La imagen visible muestra una nube de polvo y gas que se expande rápidamente y que contiene elementos pesados procedentes de la explosión de la supernova. Esa nube tiene ahora 11 años luz de diámetro, pesa 4,5 masas solares y se expande a la furiosa velocidad de 1500 kilómetros por segundo. La energía cinética de esa antigua explosión es, como mínimo, impresionante.

En el centro del objetivo hay un punto azul brillante. Ahí es donde está el *púlsar*. Fue la formación del púlsar lo que provocó la explosión de la estrella en primer lugar. Casi una masa solar de material en el núcleo de la estrella condenada implosionó en una esfera de neutrones de unos 30 kilómetros de diámetro. El

La energía cinética de esa implosión, unida a la increíble avalancha de neutrinos creada cuando se formaron todos esos neutrones, desgarró la estrella y la hizo estallar.

El púlsar gira unas 30 veces por segundo; y parpadea mientras gira. Podemos verlo parpadear en nuestros telescopios. Esos pulsos de luz son la razón por la que lo llamamos púlsar, que es la abreviatura de estrella

pulsante.

Introducción previa a los requisitos

(No te saltes esto, lo vas a necesitar).



Supongo que has cogido este libro porque eres programador informático y te intriga la noción de profesionalidad. Y así debe ser. La profesionalidad es algo que nuestra profesión necesita urgentemente.

Yo también soy programador. He sido programador durante 42¹ años; y en ese tiempo -déjame *decirte*- he visto de todo. Me han despedido. He sido alabado. He sido jefe de equipo, gerente, gruñón e incluso director general. He trabajado con programadores brillantes y he trabajado con babosos.² He trabajado en sistemas de software/hardware embebidos de alta tecnología y he trabajado en sistemas corporativos de nóminas. He programado en COBOL, FORTRAN, BAL, PDP-8, PDP-11, C, C++, Java, Ruby, Smalltalk y una plétora de otros lenguajes y sistemas. He trabajado con ladrones de sueldo poco fiables, y he trabajado con profesionales consumados. Es esta última clasificación la que constituye el tema de este libro.

En las páginas de este libro intentaré definir lo que significa ser un programador profesional. Describiré las actitudes, disciplinas y acciones que considero esencialmente profesionales.

¿Cómo sé cuáles son estas actitudes, disciplinas y acciones? Porque tuve que aprenderlas por las malas. Verás, cuando conseguí mi primer trabajo como

programador, profesional era la última palabra que habría utilizado paradescribirme.

El año era 1969. Yo tenía 17 años. Mi padre había presionado a una empresa local llamada ASC para que me contratara como programador temporal a tiempo parcial. (Sí, mi padre podía hacer cosas así. Una vez le visalir delante

de un coche que iba a toda velocidad con la mano extendida ordenando que se detuviera. El coche se detuvo. Nadie dijo "no" a mi padre). La empresa me puso a trabajar en la sala donde se guardaban todos los manuales de los ordenadores de IBM. Me hicieron poner años y años de actualizaciones en los manuales. Fue aquí donde vi por primera vez la frase: "Esta página se ha dejado intencionadamente en blanco".

Tras un par de días actualizando manuales, mi supervisor me pidió que escribiera un sencillo programa EasyCoder³. Me encantó que me lo pidieran. Nunca había escrito un programa para un ordenador real. Sin embargo, había inhalado los libros de Autocoder y tenía una vaga idea de cómo empezar.

El programa consistía simplemente en leer los registros de una cinta y reemplazar los ID de esos registros con nuevos ID. Los nuevos IDs comenzaban en 1 y se incrementaban en 1 por cada nuevo registro. Los registros con los nuevos IDs debían escribirse en una nueva cinta.

Mi supervisor me mostró una estantería en la que había muchos montones de barajas rojas y azules. Imagínate que compras 50 barajas de cartas, 25 rojas y 25 azules. Luego apilas esas barajas una encima de otra. Ese es el aspecto de estas pilas de cartas. Tenían rayas rojas y azules, y las rayas tenían unas 200 cartas cada una. Cada una de esas rayas contenía el código fuente de la biblioteca de subrutinas que los programadores solían utilizar. Los programadores simplemente tomaban el mazo superior de la pila, asegurándose de no tomar nada más que tarjetas rojas o azules, y luego lo ponían al final de su mazo de programas.

Escribí mi programa en unos formularios de codificación. Los formularios de codificación eran grandes hojas rectangulares de papel divididas en 25 líneas y 80 columnas. Cada línea representaba una tarjeta. Escribías tu programa en el formulario de codificación utilizando letras mayúsculas y un lápiz del nº 2. En las últimas 6 columnas de cada línea se escribía un número de secuencia con ese lápiz del número 2. Normalmente, incrementabas el número de secuencia en 10 para poder insertar tarjetas más tarde.

El formulario de codificación iba a parar a los perforadores de llaves. Esta empresa tenía varias docenas de mujeres que tomaban los formularios de codificación de una gran cesta de entrada y luego

"tecleaban" en las máquinas de perforación de teclas. Estas máquinas eran muy parecidas a las máquinas de escribir, salvo que los caracteres se perforaban en tarjetas en lugar de imprimirse en papel.

Al día siguiente, los teclistas me devolvieron mi programa por correo interno. Mi pequeño mazo de tarjetas perforadas estaba envuelto por mis formularios de codificación y una banda elástica. Revisé las tarjetas en busca de errores de perforación. No había ninguno. Entonces puse el mazo

de la biblioteca de subrutinas al final de mi mazo de programas, y luego llevé el mazo arriba a los operadores de la computadora.

Los ordenadores estaban detrás de puertas cerradas con llave en una sala con control ambiental y suelo elevado (para todos los cables). Llamé a la puerta y un operador me quitó austeramente mi cubierta y la puso en otra cesta dentro de la sala de ordenadores. Cuando se pusieran a ello, harían funcionar mi cubierta.

Al día siguiente me devolvieron la baraja. Estaba envuelta en un listado con los resultados de la carrera y se mantenía unida con una goma elástica. (¡En aquella época usábamos muchas gomas!)

Abrí el listado y vi que la compilación había fallado. Los mensajes de error del listado me resultaban muy difíciles de entender, así que se lo llevé a mi supervisor. Lo miró, murmuró en voz baja, hizo algunas anotaciones rápidas en el listado, cogió mi cubierta y me dijo que le siguiera.

Me llevó a la sala de fichajes y se sentó en una máquina de fichar vacía. Una por una, corrigió las tarjetas que estaban equivocadas y añadió una o dos tarjetas más. Me explicó rápidamente lo que estaba haciendo, pero todopasó como un rayo.

Llevó la nueva cubierta hasta la sala de ordenadores y llamó a la puerta. Dijo unas palabras mágicas a uno de los operadores y entró en la sala de ordenadores tras él. Me hizo una señal para que le siguiera. El operador preparó las unidades de cinta y cargó la pletina mientras nosotros observábamos. Las cintas giraron, la impresora parloteó y todo terminó. El programa había funcionado.

Al día siguiente, mi supervisor me dio las gracias por mi ayuda y me despidió. Al parecer, ASC no consideraba que tuviera tiempo para cuidar a un joven de 17 años.

Pero mi relación con ASC apenas había terminado. Unos meses más tarde, conseguí un trabajo a tiempo completo en el segundo turno de ASC

operando impresoras fuera de línea. Estas impresoras imprimían correbasura a partir de imágenes impresas que se almacenaban en cinta. Mi trabajo consistía en

cargar las impresoras con papel, cargar las cintas en las unidades de cinta, arreglar los atascos de papel y, por lo demás, simplemente ver cómo funcionan las máquinas.

Era el año 1970. La universidad no era una opción para mí, ni me atraía especialmente. La guerra de Vietnam seguía haciendo estragos y los campus eran un caos. Había seguido inhalando libros sobre COBOL, Fortran, PL/1, PDP-8 e IBM 360 Assembler. Mi intención era eludir la escuela y conducir tan duro como pudiera para conseguir un trabajo de programación.

Doce meses después conseguí ese objetivo. Me ascendieron a programador a tiempo completo en ASC. Yo, y dos de mis buenos amigos, Richard y Tim, también de 19 años, trabajamos con un equipo de otros tres programadores escribiendo un sistema de contabilidad en tiempo real para un sindicato de camioneros. La máquina era un Varian 620i. Era un sencillominiordenador de arquitectura similar a un PDP-8, salvo que tenía una palabra de 16 bits y dos registros. El lenguaje era ensamblador.

Escribimos cada línea de código en ese sistema. Y me refiero a *cada* línea. Escribimos el sistema operativo, los cabezales de interrupción, los controladores de E/S, el *sistema de archivos* para los discos, el intercambiador de superposición e incluso el enlazador reubicable. Por no hablar de todo el código de la aplicación. Escribimos todo esto en 8 meses trabajando 70 y 80 horas a la semana para cumplir un plazo infernal. Mi salario era 7.200 dólares al año.

Entregamos ese sistema. Y luego lo dejamos.

Renunciamos repentinamente, y con malicia. Verás, después de todo ese trabajo, y de haber entregado un sistema exitoso, la empresa nos dio un aumento del 2%. Nos sentimos engañados y maltratados. Varios de nosotros conseguimos trabajo en otro sitio y simplemente dimitimos.

Yo, sin embargo, adopté un enfoque diferente y muy desafortunado. Un compañero y yo irrumpimos en el despacho del jefe y renunciamos juntos en voz alta. Esto fue emocionalmente muy satisfactorio, por un día.

Al día siguiente me di cuenta de que no tenía trabajo. Tenía 19 años, estaba en el paro y no tenía ningún título. Me entrevisté para algunos puestos de programación, pero esas entrevistas no fueron bien. Así que trabajé en el taller de reparación de cortacéspedes de mi cuñado durante cuatro meses.

Por desgracia, era un pésimo reparador de cortacéspedes. Al final tuvo quedespedirme. Caí en una mala racha.

Me quedaba despierto hasta las 3 de la mañana todas las noches comiendo pizza y viendo viejas películas de monstruos en el viejo televisor en blanco y negro con orejas de conejo de mis padres. Sólo algunos de los fantasmas eran personajes en las películas. Me quedaba en la cama hasta la 1 PM porque

no quería enfrentarme a mis días tristes. Hice un curso de cálculo en un colegio comunitario local y lo suspendí. Estaba destrozado.

Mi madre me llevó aparte y me dijo que mi vida era un desastre, y que había sido una idiota por renunciar sin tener un nuevo trabajo, y por renunciar tan emocionalmente, y por renunciar junto con mi amigo. Me dijo que nunca se renuncia sin tener un nuevo trabajo, y que siempre se renuncia con calma, con frialdad y solo. Me dijo que debería llamar a mi antiguo jefe y rogarle que me devolviera mi antiguo trabajo. Me dijo:

"Tienes que comer un poco de tarta de humildad".

Los chicos de diecinueve años no son conocidos por su apetito por el pastel de la humildad, y yo no era una excepción. Pero las circunstancias habían hecho mella en mi orgullo. Al final llamé a mi jefe y le di un buen bocado a ese pastel de humildad. Y funcionó. Se alegró de volver a contratarme por 6.800 dólares al año, y yo lo acepté con gusto.

Pasé otros dieciocho meses trabajando allí, vigilando mis P y Q e intentando ser un empleado tan valioso como pudiera. Me recompensaron con ascensos y aumentos de sueldo, y con una paga regular. La vida era buena. Cuando dejé la empresa, lo hice en buenos términos y con una oferta de un trabajo mejor en el bolsillo.

Se podría pensar que he aprendido la lección, que ahora soy una profesional. Nada más lejos de la realidad. Esa fue sólo la primera de las muchas lecciones que tuve que aprender. En los años siguientes me despedirían de un trabajo por no respetar las fechas críticas, y casi me despiden de otro por filtrar inadvertidamente información confidencial a un cliente. Me puse al frente de un proyecto condenado al fracaso y lo llevé a la ruina sin pedir la ayuda que sabía que necesitaba. Defendí agresivamente mis decisiones técnicas aunque fueran contrarias a las necesidades de los clientes.

Contrataría a una persona totalmente no cualificada, cargando a mi empleador con un enorme lastre con el que lidiar. Y lo peor de todo es que hice que despidieran a otras dos personas por mi incapacidad para liderar.

Así que piense en este libro como un catálogo de mis propios errores, un borrador de mis propios crímenes, y un conjunto de pautas para que usted evite caminar en mis primeros zapatos.

1. Profesionalidad



"Oh, riéte, Curtin, viejo amigo. Es una gran broma que nos ha gastado el Señor, o el destino, o la naturaleza, lo que prefieras. Pero quien o lo

que sea que nos la jugó, ciertamente tenía sentido del humor. ¡Ja!"

-Howard, El tesoro de la Sierra Madre

¿Así que quieres ser un desarrollador de software profesional? Quieres llevar la cabeza alta y declarar al mundo: "¡Soy un *profesional!* "Quieres que la gente te mire con respeto y te trate con deferencia. Quieres que las madres te señalen y digan a sus hijos que sean como tú. Lo quieres todo. ¿No es así?

Cuidado con lo que pides

La profesionalidad es un término cargado. Ciertamente es una insignia de honor y orgullo, pero también es un marcador de responsabilidad y rendición de cuentas. Ambos van de la mano, por supuesto. No se puede estar orgulloso y honrado de algo de lo que no se puede responder.

Es mucho más fácil ser un no profesional. Los no profesionales no tienen que responsabilizarse del trabajo que hacen: lo dejan en manos de sus empleadores. Si un no profesional comete un error, el empleador limpia el desastre. Pero cuando un profesional comete un error, *limpia el desastre*.

¿Qué pasaría si permitieras que un error se colara en un módulo y le costara a tu empresa 10.000 dólares? El no profesional se encogería de hombros, diría "cosas que pasan" y empezaría a escribir el siguiente módulo. El profesional haría un cheque a la empresa por 10.000 dólares.

1

Sí, se siente un poco diferente cuando es tu propio dinero, ¿no? Pero esa sensación es la que tiene un profesional todo el tiempo. De hecho, ese sentimiento es la esencia de la profesionalidad. Porque, como ves, la profesionalidad consiste en asumir responsabilidades.

Asumir la responsabilidad

Ha leído la introducción, ¿verdad? Si no es así, vuelve a hacerlo ahora; establece el contexto de todo lo que sigue en este libro.

Aprendí a asumir responsabilidades sufriendo las consecuencias de no asumirlas.

En 1979 trabajaba para una empresa llamada Teradyne. Era el "ingeniero responsable" del software que controlaba un sistema basado en mini y microordenadores que medía la calidad de las líneas telefónicas. El miniordenador central estaba conectado a través de líneas telefónicas dedicadas o de marcación de 300 baudios a docenas de microordenadores satélite que controlaban el hardware de medición. Todo el código estaba escrito en ensamblador.

Nuestros clientes eran los directores de servicio de las principales compañías telefónicas. Cada uno era responsable de 100.000 líneas telefónicas o más.

Mi sistema ayudaba a estos gestores de áreas de servicio a encontrar y reparar averías y problemas en las líneas telefónicas antes de que sus clientes se dieran cuenta. Esto reducía los índices de quejas de los clientes que las comisiones de servicios públicos medían y utilizaban para regular las tarifas que las compañías telefónicas podían cobrar. En resumen, estos sistemas eran increíblemente importantes.

Todas las noches, estos sistemas se someten a una "rutina nocturna" en la que el miniordenador central indica a cada uno de los microordenadores satélite que compruebe todas las líneas telefónicas bajo su control. Cada mañana, el ordenador central sacaba la lista de líneas defectuosas, junto con sus características de fallo. Los responsables de las áreas de servicio utilizaban este informe para programar a los reparadores para que arreglaran los fallos antes de que los clientes pudieran quejarse.

En una ocasión envié una nueva versión a varias docenas de clientes. "Enviar" es exactamente la palabra correcta. Escribí el software en cintas y las envié

cintas a los clientes. Los clientes cargaban las cintas y luego reiniciaban los sistemas.

La nueva versión corregía algunos defectos menores y añadía una nueva función que nuestros clientes habían demandado. Les habíamos dicho que proporcionaríamos esa nueva función en una fecha determinada. A duras penas conseguí enviar las cintas de un día para otro para que llegaran en la fecha prometida.

Dos días después recibí una llamada de nuestro jefe de servicio de campo, Tom. Me dijo que varios clientes se habían quejado de que la "rutina nocturna" no se había completado y que no habían recibido ningún informe. Se me encogió el corazón porque, para poder enviar el software a tiempo, me había olvidado de probar la rutina. Había probado gran parte de las demás funciones del sistema, pero probar la rutina me llevaba horas, y tenía que enviar el software. Ninguna de las correcciones de errores estaba en el código de la rutina, así que me sentía seguro.

Perder un informe nocturno era un *gran problema*. Significaba que los reparadores tenían menos que hacer y que estarían sobrecargados más tarde. Significaba que algunos clientes podrían notar una avería y quejarse. Perder los datos de una noche es suficiente para que un jefe de área de servicio llame a Tom y le eche la bronca.

Encendí nuestro sistema de laboratorio, cargué el nuevo software y comencé una rutina. Tardó varias horas, pero luego abortó. La rutina falló. Si hubiera ejecutado esta prueba antes de enviarla, las áreas de servicio no habrían perdido datos y los gerentes de las áreas de servicio no estarían asando a Tom ahora mismo.

Llamé a Tom para decirle que podía duplicar el problema. Me dijo que la

mayoría de los otros clientes le habían llamado con la misma queja. Entonces me preguntó cuándo podría solucionarlo. Le dije que no lo sabía, pero que estaba trabajando en ello. Mientras tanto, le dije que los clientes deberían volver al antiguo software. Se enfadó conmigo diciendo que eso era un doble golpe para los clientes, ya que habían perdido toda una noche de datos y no podían utilizar la nueva función que se les había prometido.

El fallo fue difícil de encontrar y las pruebas duraron varias horas. La primera solución no funcionó. Tampoco la segunda. Me llevó varios intentos, y por tanto varios días, averiguar qué había fallado. Durante todo ese tiempo, Tom me llamaba cada pocas horas para preguntarme cuándo lo iba a arreglar. También se aseguró de que yo supiera las broncas que estaba recibiendo de los responsables del área de servicio, y lo embarazoso que era para él decirles que volvieran a poner las cintas antiguas.

Al final, encontré el defecto, envié las nuevas cintas y todo volvió a la normalidad. Tom, que no era mi jefe, se calmó y dejamos atrás todo el episodio. Mi jefe se acercó a mí cuando todo terminó y me dijo: "Seguro que no vuelves a hacerlo". Estuve de acuerdo.

Al reflexionar me di cuenta de que enviar sin probar la rutina había sido una irresponsabilidad. La razón por la que descuidé la prueba fue para poder decir que había enviado a tiempo. Se trataba de salvar la cara. No me había preocupado por el cliente, ni por mi empleador. Sólo me preocupaba mi propia reputación. Debería haber asumido la responsabilidad desde el principio y haberle dicho a Tom que las pruebas no estaban completas y que no estaba preparado para enviar el software a tiempo. Eso habría sido duro, y Tom se habría enfadado. Pero ningún cliente habría perdido datos, y ningún jefe de servicio habría llamado.

Primero, no hacer daño

Entonces, ¿cómo asumimos la responsabilidad? Hay algunos principios. Recurrir al juramento hipocrático puede parecer arrogante, pero ¿qué mejor fuente hay? Y, de hecho, ¿no tiene sentido que la primera responsabilidad, y el primer objetivo, de un aspirante a profesional sea utilizar sus poderes para el bien?

¿Qué daño puede hacer un desarrollador de software? Desde un punto de vista puramente de software, puede dañar tanto la función como la estructura del software. Exploraremos cómo evitar hacer precisamente eso.

No hacer daño a la función

Está claro que queremos que nuestro software funcione. De hecho, la mayoría de nosotros somos programadores porque conseguimos que algo funcionara una vez y queremos volver a tener esa sensación. Pero no somos los únicos que queremos que el software funcione. Nuestros clientes y

empleadores también quieren que funcione. De hecho, nos pagan para que creamos un software que funcione tal y como ellos quieren.

Dañamos el funcionamiento de nuestro software cuando creamos errores. Por lo tanto, para ser profesionales, no debemos crear errores.

"¡Pero espera!" Te oigo decir. "Eso no es razonable. El software es demasiado complejo para crearlo sin errores".

Por supuesto, tienes razón. El software *es* demasiado complejo para crearlo sin errores. Desgraciadamente, eso no te libra de los errores. El cuerpo humano es demasiado

complejo de entender en su totalidad, pero los médicos siguen haciendo un juramento de no hacer daño. Si ellos no se libran de un compromiso *así*, ¿cómo podemos hacerlo nosotros?

"¿Nos está diciendo que debemos ser perfectos?" ¿Oigo que objetáis?

No, te digo que debes ser responsable de tus imperfecciones. El hecho de que seguramente se produzcan errores en tu código no significa que no seas responsable de ellos. El hecho de que la tarea de escribir un software perfecto sea prácticamente imposible no significa que no seas responsable de la imperfección.

A un profesional le corresponde rendir cuentas de sus errores, aunque éstos sean prácticamente seguros. Así que, mi aspirante a profesional, lo primero que debes practicar es pedir disculpas. Las disculpas son necesarias, pero insuficientes. No puedes seguir cometiendo los mismos errores una y otra vez. A medida que vayas madurando en tu profesión, tu índice de errores debería disminuir rápidamente hacia la asíntota de cero. No llegará nunca a cero, pero es tu responsabilidad acercarte lo más posible a ella.

El control de calidad no debe encontrar nada

Por lo tanto, cuando publiques tu software, debes esperar que el control de calidad no encuentre problemas. Es muy poco profesional enviar a propósito un código que sabes que es defectuoso al control de calidad. ¿Y qué código sabes que es defectuoso?

Cualquier código del que no estés *seguro*.

Algunas personas utilizan el control de calidad como los cazadores de errores. Les envían código que no han comprobado a fondo. Dependen de que el control de calidad encuentre los errores y los comunique a los desarrolladores. De hecho, algunas empresas recompensan al departamento de control de calidad en función del número de errores que encuentren. Cuantos más fallos, mayor es la recompensa.

No importa que este sea un comportamiento desesperadamente caro que perjudica a la empresa y al software. No importa que este comportamiento arruine los calendarios y socave la confianza de la empresa en el equipo de desarrollo. No importa que este comportamiento sea simplemente perezoso.

irresponsable. Liberar código al departamento de control de calidad que no se sabe si funciona es poco profesional. Viola la regla de "no hacer daño".

¿Encontrará el control de calidad errores? Probablemente, así que

preparate para disculparte, y luego averigua por qué esos errores se te escaparon y haz algo para evitar que vuelvan a ocurrir.

Cada vez que el departamento de control de calidad, o peor aún, un *usuario*, encuentra un problema, usted debería sorprenderse, sentirse apenado y estar decidido a evitar que se repita.

Debes *saber* que funciona

¿Cómo puede *saber* que su código funciona? Es fácil. Pruébalo. Pruébalo de nuevo. Pruébalo hacia arriba. Pruébalo hacia abajo. Pruébalo de siete maneras hasta el domingo.

Quizás te preocupa que probar tanto tu código te lleve demasiado tiempo. Al fin y al cabo, tienes horarios y plazos que cumplir. Si te pasas todo el tiempo probando, nunca conseguirás escribir nada más. Buena observación. Así que automatiza tus pruebas. Escribe pruebas unitarias que puedas ejecutar en un momento dado, y ejecuta esas pruebas tan a menudo como puedas.

¿Qué parte del código debe probarse con estas pruebas unitarias automatizadas? ¿Es necesario responder a esta pregunta? Todo. Todo. De. Todo.

¿Sugiero una cobertura de pruebas del 100%? No, no lo estoy *sugiriendo*. Lo *exijo*. Cada línea de código que escribas debe ser probada. Punto.

¿No es eso poco realista? Por supuesto que no. Sólo escribes código porque esperas que se ejecute. Si esperas que se ejecute, debes *saber* que funciona. La única forma de saberlo es probándolo.

Soy el principal colaborador y committer de un proyecto de código abierto llamado FITNESSE. En el momento de escribir este artículo hay 60 bloques en FITNESSE. 26 de esos 60 están escritos en más de 2000 pruebas unitarias. Emma informa que la cobertura de esas 2000 pruebas es de ~90%.

¿Por qué mi cobertura de código no es mayor? Porque Emma no puede ver todas las líneas de código que se ejecutan. Creo que la cobertura es mucho mayor que eso. ¿La cobertura es del 100%? No, el 100% es una asíntota.

¿Pero no hay código difícil de probar? Sí, pero sólo porque ese código ha sido *diseñado* para ser difícil de probar. La solución es diseñar el código para que sea *fácil de probar*. Y la mejor manera de hacerlo es escribir primero las pruebas, antes de escribir el código que las pasa.

Se trata de una disciplina conocida como Desarrollo Dirigido por Pruebas (TDD), de la que hablaremos en un capítulo posterior.

Control de calidad automatizado

Todo el procedimiento de control de calidad de FITNESSE es la ejecución de las pruebas unitarias y de aceptación. Si esas pruebas se superan, hago el envío. Esto significa que mi procedimiento de control de calidad

tarda unos tres minutos, y puedo ejecutarlo en un capricho.

Ahora bien, es cierto que nadie muere si hay un fallo en FITNESSE. Tampoco nadie pierde millones de dólares. Por otro lado, FITNESSE tiene muchos miles de usuarios, y una lista de errores *muy* pequeña.

Ciertamente, algunos sistemas son tan críticos que una breve prueba automatizada es insuficiente para determinar si están listos para su despliegue. Por otro lado, como desarrollador necesitas un mecanismo relativamente rápido y fiable para saber que el código que has escrito funciona y no interfiere con el resto del sistema. Así que, como mínimo, tus pruebas automatizadas deberían decirte que es *muy probable que* el sistema pase el control de calidad.

No hacer daño a la estructura

El verdadero profesional sabe que ofrecer una función a expensas de la estructura es una tontería. Es la estructura de tu código la que le permite ser flexible. Si se compromete la estructura, se compromete el futuro.

El supuesto fundamental en el que se basan todos los proyectos de software es que el software es fácil de cambiar. Si se viola este supuesto creando estructuras inflexibles, se socava el modelo económico en el que se basa toda la industria.

En resumen: *debe poder realizar cambios sin costes desorbitados.*

Por desgracia, demasiados proyectos se hunden en un pozo de alquitrán de mala estructura. Las tareas que antes llevaban días empiezan a tardar semanas, y luego meses. La dirección, desesperada por recuperar el impulso perdido, contrata a más desarrolladores para acelerar las cosas. Pero estos desarrolladores no hacen más que añadirse al marasmo, profundizando el daño estructural y aumentando el impedimento.

Se ha escrito mucho sobre los principios y patrones de diseño de software que sustentan estructuras flexibles y mantenibles. Los ²desarrolladores profesionales de software los memorizan y se esfuerzan por adaptar su software a ellos. Pero hay un truco que muy pocos desarrolladores de software siguen: *Si quieres que tu software sea flexible, tienes que flexionarlo.*

La única manera de demostrar que tu software es fácil de cambiar es hacer cambios fáciles en él. Y cuando descubras que los cambios no son tan fáciles como pensabas, perfecciona el diseño para que el siguiente cambio sea más sencillo.

¿Cuándo haces estos sencillos cambios? *Todo el tiempo. Cada vez*

que miras un módulo haces pequeños y ligeros cambios en él para mejorar su estructura. Cada vez que lees el código ajustas la estructura.

Esta filosofía se llama a veces *refactorización despiadada*. Yo la llamo "la regla del Boy Scout": Siempre revisa un módulo más limpio que cuando lo sacaste. Haz siempre algún acto de bondad al azar en el código cada vez que lo veas.

Esto es completamente contrario a la forma en que la mayoría de la gente piensa en el software. Piensan que hacer una serie continua de cambios en el software que funciona es *peligroso*. No. Lo que es peligroso es permitir que el software permanezca estático. Si no lo flexibilizas, cuando necesites cambiarlo, lo encontrarás rígido.

¿Por qué la mayoría de los desarrolladores temen hacer cambios continuos en su código? Tienen miedo de romperlo. ¿Por qué temen romperlo?

Porque no tienen pruebas.

Todo vuelve a las pruebas. Si tienes un conjunto de pruebas automatizadas que cubren prácticamente el 100% del código, y si ese conjunto de pruebas se puede ejecutar rápidamente por capricho, entonces *simplemente no tendrás miedo de cambiar el código*. ¿Cómo demuestras que no tienes miedo de cambiar el código? Lo cambias todo el tiempo.

Los desarrolladores profesionales están tan seguros de su código y sus pruebas que son enloquecedoramente despreocupados a la hora de hacer cambios aleatorios y oportunos. Cambian el nombre de una clase por capricho. Se darán cuenta de un método largo mientras leen un módulo y lo repartirán como una cuestión de rutina.

Transformarán una sentencia switch en un despliegue polimórfico, o colapsarán una jerarquía de herencia en una cadena de comandos. En resumen, tratan el software como un escultor trata la arcilla: le dan formay lo moldean continuamente.

Ética del trabajo

Tu carrera es *tu* responsabilidad. No es responsabilidad de tu empleador asegurarse de que eres comercializable. No es responsabilidad de tu empleador formarte, ni enviarte a conferencias, ni comprarte libros. Estas cosas son *tu* responsabilidad. Ay del desarrollador de software que confía su carrera a su empleador.

Algunos empresarios están dispuestos a comprarte libros y enviarte a clases de formación y conferencias. Eso está bien, te están haciendo un favor. Pero nunca caigas en la trampa de pensar que esto es responsabilidad de tu empleador. Si su empleador no hace estas cosas por usted, debe encontrar la manera de hacerlas usted mismo.

Tampoco es responsabilidad de tu empleador darte el tiempo que

necesitas para aprender. Algunos empleadores pueden proporcionar ese tiempo. Algunos empleadores pueden incluso exigirte que te tomes ese tiempo. Pero, de nuevo, le están haciendo un favor, y usted debe estar debidamente agradecido. Esos favores no son algo que debas esperar.

Usted debe a su empleador una determinada cantidad de tiempo y esfuerzo. En aras del argumento, utilicemos la norma estadounidense de 40 horas semanales. Estas 40 horas deben dedicarse a *los* problemas de tu empleador, no a los *tuyos*.

Debe prever que trabajará 60 horas a la semana. Las primeras 40 son para su empresa. Las 20 restantes son para ti. Durante esas 20 horas restantes debes leer, practicar, aprender y mejorar tu carrera.

Te oigo pensar: "¿Pero qué pasa con mi familia? ¿Y mi vida? ¿Debo sacrificarlos por mi empleador?".

No estoy hablando de *todo* tu tiempo libre. Hablo de 20 horas extra a la semana. Eso es aproximadamente tres horas al día. Si utilizas la hora del almuerzo para leer, escuchar podcasts en tu viaje al trabajo y dedicas 90 minutos al día a aprender un nuevo idioma, lo tendrás todo cubierto.

Haz las cuentas. En una semana hay 168 horas. Dale a tu empleador 40, ya tu carrera otras 20. Quedan 108. Otras 56 para dormir, lo que deja 52 para todo lo demás.

Tal vez no quiera asumir ese tipo de compromiso. Está bien, pero entonces no debes considerarte un profesional. Los profesionales dedican *tiempo* a cuidar su profesión.

Tal vez pienses que el trabajo debe quedarse en el trabajo y que no debes llevarlo a casa. Estoy de acuerdo. No deberías trabajar para tu empresa durante esas 20 horas. En su lugar, debería trabajar en su carrera.

A veces estas dos cosas están alineadas entre sí. A veces el trabajo que haces para tu empleador es muy beneficioso para tu carrera. En ese caso, dedicar parte de esas 20 horas es razonable. Pero recuerda que esas 20 horas son para *ti*. Hay que utilizarlas para que seas más valioso como profesional.

Tal vez piense que esto es una receta para el agotamiento. Por el contrario, es una receta para *evitar el agotamiento*. Es de suponer que te convertiste en desarrollador de software porque te apasiona el software y tu deseo de ser un profesional está motivado por esa pasión. Durante esas 20 horas deberías estar haciendo aquellas cosas que *refuerzan* esa pasión. Esas 20 horas deben ser *divertidas*.

Conozca su campo

¿Sabe lo que es un gráfico Nassi-Shneiderman? Si no es así, ¿por qué no?
¿Sabe la diferencia entre una máquina de estado de Mealy y una de

Moore? Deberías. ¿Podrías escribir un quicksort sin buscarlo? ¿Sabes qué significa el término "Análisis de Transformación"? ¿Podrías realizar una descomposición funcional con Diagramas de Flujo de Datos? ¿Qué significa el término "Tramp Data"? ¿Ha oído el término "Conascencia"? ¿Qué es una Tabla de Parnas?

Una gran cantidad de ideas, disciplinas, técnicas, herramientas y terminologías decoran los últimos cincuenta años de nuestro campo.

¿Cuánto sabe usted de todo esto? Si quiere ser un profesional, debe conocer una parte considerable de todo ello y aumentar constantemente el tamaño de esa parte.

¿Por qué hay que saber estas cosas? Al fin y al cabo, ¿no está progresando nuestro campo tan rápidamente que todas estas viejas ideas se han vuelto irrelevantes? La primera parte de esta pregunta parece obvia a primera vista. Ciertamente, nuestro campo está progresando y a un ritmo feroz. Sin embargo, lo interesante es que ese progreso es en muchos aspectos periférico. Es cierto que ya no esperamos 24 horas para compilar. Es cierto que escribimos sistemas que tienen un tamaño de gigabytes. Es cierto que trabajamos en medio de una red que se extiende por todo el mundo y que proporciona acceso instantáneo a la información. Por otro lado, estamos escribiendo las mismas sentencias `if` y `while` que escribíamos hace 50 años. Mucho ha cambiado. Mucho no ha cambiado.

La segunda parte de la pregunta no es cierta. Muy pocas ideas de los últimos 50 años se han vuelto irrelevantes. Algunas se han dejado de lado, es cierto. La noción de hacer desarrollo en cascada ha caído ciertamente en

desgracia. Pero eso no significa que no debemos saber qué es, y cuáles son sus puntos buenos y malos.

Sin embargo, en general, la gran mayoría de las ideas ganadas con esfuerzo en los últimos 50 años son tan valiosas hoy como lo fueron entonces. Tal vez sean incluso más valiosas ahora.

Recuerda la maldición de Santayana: "Los que no pueden recordar el pasado están condenados a repetirlo".

He aquí una lista *mínima* de las cosas que todo profesional del software debería conocer:

- Patrones de diseño. Debería ser capaz de describir los 24 patrones del libro GOF y tener un conocimiento práctico de muchos de los patrones de los libros POSA.
- Principios de diseño. Debe conocer los principios de SOLID y tener una buena comprensión de los principios de los componentes.
- Métodos. Debe entender XP, Scrum, Lean, Kanban, Waterfall, Análisis Estructurado y Diseño Estructurado.
- Disciplinas. Debe practicar TDD, diseño orientado a objetos,

programación estructurada, integración continua y programación en parejas.

- Artefactos: Deberá saber utilizar: UML, DFDs, diagramas de estructura, redes de Petri, diagramas y tablas de transición de estados, diagramas de flujo y tablas de decisión.

Aprendizaje continuo

El frenético ritmo de cambio de nuestro sector hace que los desarrolladores de software deban seguir aprendiendo una gran cantidad de cosas para mantenerse al día. Ay de los arquitectos que dejen de codificar: rápidamente se verán irrelevantes. Ay de los programadores que dejen de aprender nuevos lenguajes: verán cómo la industria les pasa por encima. Ay de los desarrolladores que no aprendan nuevas disciplinas y técnicas: sus compañeros destacarán mientras ellos declinan.

¿Visitaría usted a un médico que no se mantuviera al día con las revistas médicas? ¿Contrataría a un abogado fiscal que no se mantuviera al día con las leyes y los precedentes fiscales? ¿Por qué deberían los empresarios contratar a promotores que no se mantienen al día?

Leer libros, artículos, blogs, tweets. Ir a conferencias. Acude a grupos de usuarios. Participa en grupos de lectura y estudio. Aprende cosas que estén fuera de tu zona de confort. Si eres un programador .NET, aprende

Java. Si es un programador de Java

programador, aprende Ruby. Si eres programador de C, aprende Lisp. Si quieres estrujar tu cerebro de verdad, aprende Prolog y Forth.

Practica

Los profesionales practican. Los verdaderos profesionales trabajan duro para mantener sus habilidades afiladas y preparadas. No basta con hacer el trabajo diario y llamarlo práctica. Hacer tu trabajo diario es rendimiento, no práctica. La práctica es cuando ejercitas específicamente tus habilidades *fuera* del desempeño de tu trabajo con el único propósito de refinar y mejorar esas habilidades.

¿Qué puede significar para un desarrollador de software ejercer? A primera vista, el concepto parece absurdo. Pero párate a pensar un momento.

Piensa en cómo los músicos dominan su oficio. No es actuando. Es practicando. ¿Y cómo practican? Entre otras cosas, tienen ejercicios especiales que realizan. Escalas, estudios y ejecuciones. Lo hacen una y otra vez para entrenar sus dedos y su mente, y para mantener el dominio de su habilidad.

Entonces, ¿qué pueden hacer los desarrolladores de software para practicar? En este libro hay un capítulo entero dedicado a diferentes técnicas de práctica, así que no entraré en muchos detalles aquí. Una técnica que utilizo

con frecuencia es la repetición de ejercicios sencillos, como el juego de los bolos los factores primos. A estos ejercicios los llamo *kata*. Hay muchos katas entre los que elegir.

Una kata suele presentarse en forma de un problema de programación sencillo que hay que resolver, como escribir la función que calcula los factores primos de un número entero. El objetivo de la kata no es averiguar cómo resolver el problema; eso ya lo sabes. El objetivo de la kata es entrenar tus dedos y tu cerebro.

Hago una o dos katas cada día, a menudo como parte de la adaptación al trabajo. Puedo hacerla en Java, en Ruby, en Clojure o en algún otro lenguaje en el que quiera mantener mis habilidades. Utilizaré la kata para perfeccionar una habilidad concreta, como acostumbrar mis dedos a pulsar las teclas de acceso directo, o utilizar ciertas refactorizaciones.

Piensa en la kata como un ejercicio de calentamiento de 10 minutos por la mañana y un enfriamiento de 10 minutos por la tarde.

Colaboración

La segunda mejor manera de aprender es colaborar con otras personas. Los desarrolladores de software profesionales hacen un esfuerzo especial para programar juntos, practicar juntos, diseñar y planificar juntos. Así aprenden mucho los unos de los otros y consiguen hacer más cosas más rápido y con menos errores.

Esto no significa que tengas que pasar el 100% de tu tiempo trabajando con otros. El tiempo a solas también es muy importante. Por mucho que me guste programar en pareja con otros, me vuelve loca si no puedo escaparme sola de vez en cuando.

Mentoring

La mejor manera de aprender es enseñar. Nada te meterá los hechos y los valores en la cabeza más rápido y con más fuerza que tener que comunicárselos a las personas de las que eres responsable. Así que el beneficio de la enseñanza está fuertemente a favor del profesor.

Del mismo modo, no hay mejor manera de incorporar a nuevas personas a una organización que sentarse con ellas y enseñarles los entresijos.

Los profesionales se responsabilizan personalmente de la tutela de los jóvenes. No dejarán que un joven se pasee sin supervisión.

Conozca su dominio

Es responsabilidad de todo profesional del software entender el dominio de las soluciones que está programando. Si está escribiendo un sistema de contabilidad, debe conocer el campo de la contabilidad. Si estás escribiendo una aplicación de viajes, debes conocer el sector de los viajes. No es

necesario ser un experto en la materia, pero hay una cantidad razonable de diligencia debida que debe realizar.

Al iniciar un proyecto en un nuevo dominio, lee uno o dos libros sobre el tema. Entreviste a sus clientes y usuarios sobre los fundamentos y las bases del dominio. Pasa algún tiempo con los expertos e intenta comprender sus principios y valores.

Es el peor tipo de comportamiento poco profesional codificar simplemente a partir de una especificación sin entender por qué esa especificación tiene sentido para el negocio. Más bien, debes saber lo suficiente sobre el dominio para poder reconocer y desafiar los errores de las especificaciones.

Identifíquese con su empleador/cliente

Los problemas de tu empresa son *tus* problemas. Hay que entender cuáles son esos problemas y trabajar para encontrar las mejores soluciones. A la hora de desarrollar un sistema, debe ponerse en el lugar de su empleador y asegurarse de que las funciones que está desarrollando van a satisfacer realmente las necesidades de su empleador.

Es fácil que los desarrolladores se identifiquen con los demás. Es fácil caer en una actitud de *nosotros contra ellos* con su empleador. Los profesionales lo evitan a toda costa.

Humildad

La programación es un acto de creación. Cuando escribimos código estamos creando algo de la nada. Estamos imponiendo audazmente el orden sobre el caos. Estamos ordenando con confianza, con detalles precisos, los comportamientos de una máquina que, de otro modo, podría causar un daño incalculable. Por eso, programar es un acto de suprema arrogancia.

Los profesionales saben que son arrogantes y no son falsamente humildes. Un profesional conoce su trabajo y se enorgullece de él. Un profesional confía en sus capacidades y asume riesgos audaces y calculados basados en esa confianza. Un profesional no es tímido.

Sin embargo, un profesional también sabe que habrá momentos en los que fallará, sus cálculos de riesgo serán erróneos, sus habilidades se quedarán cortas; se mirará al espejo y verá a un tonto arrogante que le sonríe.

Por eso, cuando un profesional se encuentre en el punto de mira de una broma, será el primero en reírse. Nunca ridiculizará a los demás, sino que aceptará el ridículo cuando sea merecido y se reirá cuando no lo sea. No rebajará a otro por cometer un error, porque sabe que él puede ser el siguiente en fracasar.

Un profesional comprende su suprema arrogancia, y que el destino acabará

por darse cuenta y nivelar su puntería. Cuando esa puntería se conecta, lo mejor que puedes hacer es seguir el consejo de Howard: Ríete.

Bibliografía

[PPP2001]: Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.

2. Decir no



"Hazlo; o no lo hagas. No hay que intentarlo".

-Yoda

A principios de los años 70, yo y dos de mis amigos de diecinueve años trabajábamos en un sistema de contabilidad en tiempo real para el sindicato de camioneros de Chicago para una empresa llamada ASC. Si te vienen a la mente nombres como Jimmy Hoffa, deberían. En 1971 no te metías con los teamsters.

Nuestro sistema debía entrar en funcionamiento en una fecha determinada. De esa fecha dependía *mucho* dinero. Nuestro equipo había trabajado 60, 70y 80 horas semanales para intentar cumplir el calendario.

Una semana antes de la fecha de entrada en funcionamiento, por fin conseguimos montar el sistema en su totalidad. Había muchos errores y problemas que resolver, y trabajamos frenéticamente en la lista. Apenas había tiempo para comer y dormir, y mucho menos para pensar.

Frank, el gerente de ASC, era un coronel retirado de las Fuerzas Aéreas. Era uno de esos gerentes ruidosos y directos. Era su camino o la carretera, y te ponía en esa carretera dejándote caer desde 10.000 pies sin paracaídas. Los jóvenes de diecinueve años apenas podíamos hacer contacto visual conél.

Frank dijo que tenía que estar hecho para la fecha. Eso era todo. La fecha llegaría, y habríamos terminado. Punto. Sin discusión. Cambio y fuera.

Mi jefe, Bill, era un tipo simpático. Llevaba bastantes años trabajando con Frank y entendía lo que era posible con él, y lo que no. Nos dijo que íbamos a ir en vivo en la fecha, pase lo que pase.

Así que nos pusimos en marcha en la fecha. Y fue un desastre total.

Había una docena de terminales semidúplex de 300 baudios que conectaban la sede de Teamster en Chicago con nuestra máquina a cincuenta kilómetros al norte, en los suburbios. Cada uno de esos terminales se bloqueaba cada 30 minutos más o menos. Ya habíamos visto este problema antes, pero no habíamos simulado el tráfico que los empleados de entrada de datos del sindicato estaban introduciendo de repente en nuestro sistema.

Para empeorar las cosas, las hojas sueltas que se imprimían en los teletipos ASR35 que también estaban conectados a nuestro sistema por líneas telefónicas de 110 baudios se congelaban en medio de la impresión.

La solución a estos bloqueos era reiniciar. Así que tenían que conseguir que todos los que tuvieran el terminal activo terminaran su trabajo y se detuvieran. Cuando todos estuvieran parados, nos llamarían para reiniciar. La gente que había sido congelada tendría que empezar de nuevo. Y esto sucedía más de una vez por hora.

Después de medio día de esto, el director de la oficina de Teamster nos dijo que apagáramos el sistema y no lo volviéramos a poner en marcha hasta que lo tuviéramos funcionando. Mientras tanto, ellos habían perdido medio día de trabajo e iban a tener que volver a introducirlo todo con el sistema antiguo.

Oímos los lamentos y rugidos de Frank por todo el edificio. Se prolongaron durante mucho, mucho tiempo. Entonces Bill, y el analista de nuestro sistema, Jalil, se acercaron a nosotros y nos preguntaron cuándo podríamos tener el sistema estable. Les dije: "cuatro semanas".

La mirada en sus rostros era de horror y luego de determinación. "No", dijeron, "debe estar funcionando para el viernes".

Así que dije: "Mira, apenas conseguimos que este sistema funcionara la semana pasada. Tenemos que sacudir los problemas y las cuestiones. Necesitamos cuatro semanas".

Pero Bill y Jalil fueron inflexibles. "No, realmente tiene que ser el viernes. ¿Puedes al menos intentarlo?"

Entonces nuestro jefe de equipo dijo: "Vale, lo intentaremos".

El viernes fue una buena elección, la carga del fin de semana fue mucho menor. Pudimos encontrar más problemas y corregirlos antes de que llegara el lunes. Aun así, todo el castillo de naipes estuvo a punto de derrumbarse de nuevo. La congelación

Los problemas seguían sucediendo una o dos veces al día. También hubo otros problemas. Pero poco a poco, tras unas semanas más, conseguimos que el sistema se calmara y que la vida normal pareciera



Y entonces, como les dije en la introducción, todos renunciamos. Y se quedaron con una verdadera crisis en sus manos. Tuvieron que contratar a un nuevo grupo de programadores para tratar de resolver el enorme flujo de problemas que les llegaba del cliente.

¿A quién podemos culpar de esta debacle? Está claro que el estilo de Frank es parte del problema. Sus intimidaciones le dificultaron escuchar la verdad.

Ciertamente, Bill y Jalil deberían haber presionado a Frank mucho más de lo que lo hicieron. Ciertamente, nuestro jefe de equipo no debería haber cedido a la demanda del viernes. Y ciertamente yo debería haber seguido diciendo "no" en lugar de ponerme en la cola de nuestro jefe de equipo.

Los profesionales dicen la verdad al poder. Los profesionales tienen el valor de decir no a sus jefes.

¿Cómo decir que no a tu jefe? Después de todo, ¡es tu *jefe*! ¿No se supone que debes hacer lo que dice tu jefe?

No. No si eres un profesional.

Los esclavos no pueden decir que no. Los obreros pueden dudar en decir que no. Pero de los profesionales se *espera que digan que no*. De hecho, los buenos gestores anhelan a alguien que tenga las agallas para decir que no. Es la única manera de conseguir que se haga algo.

Roles adversos

Uno de los críticos de este libro realmente odiaba este capítulo. Dijo que casi le hizo dejar el libro. Él había creado equipos en los que no había relaciones de enfrentamiento; los equipos trabajaban juntos en armonía y sin confrontación.

Me alegro por este crítico, pero me pregunto si sus equipos son realmente tan libres de enfrentamientos como él supone. Y si lo son, me pregunto si son tan eficientes como podrían serlo. Mi propia experiencia ha sido que las decisiones difíciles se toman mejor a través de la confrontación de papeles adversos.

Los directivos son personas que tienen un trabajo que hacer, y la mayoría de los directivos saben hacer ese trabajo bastante bien. Parte de ese trabajo consiste en perseguir y defender sus objetivos

tan agresivamente como puedan.

Por la misma razón, los programadores también son personas con un trabajo que hacer, y la mayoría de ellos saben cómo hacer ese trabajo bastante bien. Si son profesionales, perseguirán y defenderán *sus* objetivos con la mayor agresividad *posible*.

Cuando tu jefe te dice que la página de acceso tiene que estar lista para mañana, está persiguiendo y defendiendo uno de sus objetivos. Está haciendo su trabajo. Si sabe perfectamente que conseguir que la página de inicio de sesión esté lista para mañana es imposible, entonces no está haciendo su trabajo si dice "Vale, lo intentaré". La única manera de hacer su trabajo, en ese momento, es decir "No, eso es imposible".

¿Pero no tienes que hacer lo que dice tu jefe? No, tu jefe cuenta con que defiendas tus objetivos con la misma agresividad que él defiende los suyos. Así es como los dos llegaréis *al mejor resultado posible*.

El mejor resultado posible es el objetivo que usted y su jefe comparten. El truco está en encontrar ese objetivo, y eso suele requerir una negociación.

La negociación a veces puede ser agradable.

Mike: "Paula, necesito que la página de acceso esté hecha para mañana". Paula: "¡Oh, vaya! ¿Tan pronto? Bueno, vale, lo intentaré".

MIKE: "Bien, eso es genial. Gracias".

Fue una pequeña y agradable conversación. Se evitó toda confrontación. Ambas partes se fueron sonriendo. Muy bien.

Pero ambas partes se comportaron de manera poco profesional. Paula sabe muy bien que la página de acceso le va a llevar más de un día, así que está mintiendo. Puede que ella no piense que es una mentira. Tal vez piense que *realmente lo va a intentar*, y tal vez mantenga alguna escasa esperanza de que realmente lo consiga. Pero al final, sigue siendo una mentira.

Mike, por otro lado, aceptó el "Lo intentaré" como "Sí". Eso es una tontería. Debería haber sabido que Paula estaba tratando de evitar la confrontación, por lo que debería haber presionado diciendo: "Pareces vacilante. ¿Estás segura de que puedes hacerlo mañana?"

Aquí hay otra conversación agradable.

MIKE: "Paula, necesito la página de acceso hecha para mañana".

Paula: "Oh, lo siento Mike, pero me va a llevar más tiempo que eso". Mike: "¿Cuándo crees que podrás tenerlo hecho?"

Paula: "¿Qué tal dentro de dos semanas?"

Mike: (garabatea algo en su agenda) "Vale, gracias".

Por muy agradable que fuera, también fue terriblemente disfuncional y absolutamente poco profesional. Ambas partes fallaron en su búsqueda del

mejor resultado posible. En lugar de preguntar si dos semanas estarían bien, Paula debería haber sido más asertiva: "Me va a llevar dos semanas, Mike".

Mike, en cambio, acepta la fecha sin rechistar, como si sus propios objetivos no importaran. Uno se pregunta si no se limitará a informar a su jefe de que la demostración al cliente tendrá que posponerse por culpa de Paula. Ese tipo de comportamiento pasivo-agresivo es moralmente reprochable.

En todos estos casos, ninguna de las partes ha perseguido un objetivo común aceptable. Ninguna de las partes ha buscado el mejor resultado posible. Intentemos esto.

Mike: "Paula, necesito la página de inicio de sesión hecha para mañana". Paula: "No, Mike, es un trabajo de dos semanas".

Mike: "¿Dos semanas? Los arquitectos lo estimaron en tres días y ya han pasado cinco".

Paula: "Los arquitectos se equivocaron, Mike. Hicieron sus estimaciones antes de que el departamento de marketing de productos conociera los requisitos. Tengo al menos diez días más de trabajo en esto. ¿No has visto mi estimación actualizada en la wiki?"

Mike: (con aspecto severo y temblando de frustración) "Esto no es aceptable Paula. Los clientes van a venir a una demostración mañana, y tengo que mostrarles la página de inicio de sesión funcionando."

Paula: "¿Qué parte de la página de acceso necesitas que funcione para mañana?"

MIKE: "¡Necesito *la página de inicio de sesión!* Necesito poder iniciar *la sesión*".

Paula: "Mike, puedo darte una maqueta de la página de inicio de sesión que te permitirá conectarte. Ya la tengo funcionando. En realidad no comprobará tu nombre de usuario y contraseña, y no te enviará por correo electrónico una contraseña olvidada. No tendrá el banner de noticias de la compañía "Times-squaring" alrededor de la parte superior, y el botón de ayuda y el texto hover no funcionarán. No almacenará una cookie para recordarte para la próxima vez, y no pondrá ninguna

restricciones de permisos para ti. Pero podrás conectarte. ¿Serás suficiente?"

MIKE: "¿Podré conectarme?"

Paula: "Sí, podrás conectarte".

Mike: "¡Estupendo Paula, eres una salvavidas!" (se aleja bombeando el aire y diciendo "¡Sí!")

Llegaron al mejor resultado posible. Lo hicieron diciendo que no y luego elaborando una solución mutuamente aceptable para ambos. Actuaron como profesionales. La conversación fue un poco adversa y hubo algunos momentos incómodos, pero eso es de esperar cuando dos personas persiguen asertivamente objetivos que no están perfectamente alineados.

¿Y el por qué?

Tal vez pienses que Paula debería haber explicado *por qué* la página de inicio de sesión iba a tardar tanto. Mi experiencia es que el *por qué* es mucho menos importante que el *hecho*. Ese hecho es que la página de acceso requerirá dos semanas. El por qué tardará dos semanas es sólo un detalle.

Aun así, saber el porqué podría ayudar a Mike a entender, y por tanto a aceptar, el hecho. Es justo. Y en situaciones en las que Mike tiene la experiencia técnica y el temperamento para entender, tales explicaciones podrían ser útiles. Por otro lado, Mike podría estar en desacuerdo con la conclusión. Mike podría decidir que Paula lo estaba haciendo mal. Podría decirle que no necesita tantas pruebas, ni tanta revisión, o que el paso 12 podría omitirse. Dar demasiados detalles puede ser una invitación a la microgestión.

Lo que está en juego

El momento más importante para decir que no es cuando hay más en juego. Cuanto mayor es lo que está en juego, más valioso es el no.

Esto debería ser evidente. Cuando el coste del fracaso es tan alto que la supervivencia de la empresa depende de ello, hay que estar absolutamente decidido a dar a los directivos la mejor información posible. Y eso a menudo significa decir que no.

Don (Director de Desarrollo): "Así pues, nuestra estimación actual para la finalización del proyecto Golden Goose es de doce semanas a partir de hoy,

con una incertidumbre de más o menos cinco semanas".

Charles (director general): (se queda mirando durante quince segundos mientras su cara se enrojece) "¿Quieres decir que podríamos estar a diecisiete semanas del parto?"

Don: "Es posible, sí".

Charles: (se levanta, Don se levanta un segundo después) "¡Maldita sea Don! Se suponía que esto debía estar hecho hace tres semanas. Tengo a Galitron llamándome todos los días preguntando dónde está su maldito sistema. ¿No voy a decirles que tienen que esperar otros cuatro meses? Tienes que hacerlo mejor".

Chuck, te dije *hace tres meses*, después de todos los despidos, que

necesitaríamos cuatro meses más. Quiero decir, Cristo Chuck, ¡recortaste mi personal un veinte por ciento! ¿Le dijiste entonces a Galitron que llegaríamos tarde?"

Charles: "Sabes muy bien que no lo hice. No podemos permitirnos perder esa orden, Don. (Charles hace una pausa, su cara se pone blanca) Sin Galitron, estamos realmente jodidos. Lo sabes, ¿verdad? Y ahora con este retraso, me temo que... ¿Qué le voy a decir a la junta? (Vuelve a sentarse lentamente en su asiento, intentando no derrumbarse) Don, tienes que hacerlo mejor".

Don: "No hay nada que pueda hacer Chuck. Ya hemos pasado por esto. Galitron no quiere recortar el alcance, y no aceptan ninguna liberación intermedia. Quieren hacer la instalación una vez y terminar con ella. Simplemente no puedo hacer eso más rápido. *No* va a suceder".

Charles: "Maldita sea. Supongo que no importaría que te dijera que tu trabajo está en juego".

Don: "Despedirme no va a cambiar la estimación, Charles".

Charles: "Hemos terminado aquí. Vuelve a tu equipo y mantén este proyecto en marcha. Tengo que hacer unas llamadas muy duras".

Por supuesto, Charles debería haberle dicho a Galitron que no hace tres meses, cuando se enteró del nuevo presupuesto. Al menos ahora está haciendo lo correcto llamándoles (y a la junta). Pero si Don no se hubiera mantenido firme, esas llamadas podrían haberse retrasado aún más.

Ser un "jugador de equipo"

Todos hemos oído lo importante que es ser un "jugador de equipo". Ser unjugador de equipo significa jugar en tu posición tan bien como puedas, y ayudar a

a sus compañeros de equipo cuando se meten en un lío. Un jugador de equipo se comunica con frecuencia, está pendiente de sus compañeros y ejecuta sus propias responsabilidades lo mejor posible.

Un jugador de equipo no es alguien que dice que sí todo el tiempo. Considere este escenario:

Paula: "Mike, tengo esas estimaciones para ti. El equipo está de acuerdo en que estaremos listos para hacer una demostración en unas ocho semanas, más o menos".

Mike: "Paula, ya hemos programado la demostración para dentro de seis semanas".

Paula: "¿Sin escucharnos primero? Vamos Mike, no puedes presionarnos con eso".

MIKE: "Ya está hecho".

Paula: (suspiro) "Vale, mira, volveré al equipo y averiguaré qué podemos entregar con seguridad en seis semanas, pero no será todo el sistema.

Faltarán algunas funciones y la carga de datos estará incompleta".

Mike: "Paula, el cliente espera ver una demostración completa".

Paula: "Eso no va a suceder Mike".

MIKE: "Maldita sea. Bien, elabora el mejor plan que puedas y avísame mañana".

Paula: "Eso puedo hacerlo".

MIKE: "¿No hay algo que puedas hacer para traer esta fecha? Tal vez haya una manera de trabajar de forma más inteligente y creativa".

Paula: "Somos muy creativos, Mike. Tenemos un buen manejo del problema, y la fecha va a ser de ocho o nueve semanas, no de seis".

Mike: "Podrías trabajar horas extras".

Paula: "Eso sólo hace que vayamos más despacio, Mike. ¿Recuerdas el lío que armamos la última vez que ordenamos horas extras?"

MIKE: "Sí, pero eso no tiene por qué ocurrir esta vez".

Paula: "Será como la última vez, Mike. Confía en mí. Van a ser ocho o nueve semanas, no seis".

MIKE: "De acuerdo, dame tu mejor plan, pero sigue pensando encómo hacerlo en seis semanas. Sé que se os ocurrirá algo".

Paula: "No, Mike, no lo haremos. Te conseguiré un plan para seis semanas, pero le faltarán muchas funciones y datos. Así es como va a ser".

MIKE: "Vale, Paula, pero apuesto a que podéis hacer milagros si lo intentáis". (Paula se aleja negando con la cabeza).

Más tarde, en la reunión de estrategia del Director...

Don: "Bien Mike, como sabes el cliente vendrá para una demostración en seis semanas. Esperan ver todo funcionando".

Mike: "Sí, y estaremos preparados. Mi equipo se está dejando la piel en esto y vamos a conseguirlo. Tendremos que trabajar algunas horas extras y ser bastante creativos, ¡pero lo conseguiremos!"

Don: "Es estupendo que tú y tu personal seáis tan jugadores de equipo".

¿Quiénes eran los *verdaderos* jugadores del equipo en este escenario?

Paula jugaba para el equipo, porque representaba lo que se podía y no se podía hacer de la mejor manera posible. Defendió agresivamente su posición, a pesar de las insinuaciones y los halagos de Mike. Mike jugaba en un equipo de uno.

Mike es para Mike. Está claro que no está en el equipo de Paula porque acaba de comprometerla a algo que ella dijo explícitamente que no podía hacer. Tampoco está en el equipo de Don (aunque él no estaría de acuerdo) porque acaba de mentir descaradamente.

Entonces, ¿por qué Mike hizo esto? Quería que Don lo viera como un jugador de equipo, y tiene fe en su capacidad para engatusar y manipular a Paula para que *intente* cumplir el plazo de seis semanas. Mike no es malvado; simplemente confía demasiado en su capacidad para conseguir que la gente haga lo que él quiere.

Intentando

Lo peor que podría hacer Paula en respuesta a las manipulaciones de Mike es decir "Vale, lo intentaremos". Odio canalizar a Yoda aquí, pero en este caso tiene razón. *No hay ningún intento*.

¿Quizás no te gusta esa idea? Tal vez piense que *intentarlo* es algo positivo. Después de todo, ¿habría descubierto Colón América si no lo hubiera intentado?

La palabra *intentar* tiene muchas definiciones. La definición con la que discrepo aquí es "aplicar un esfuerzo extra". ¿Qué esfuerzo extra podría aplicar Paula para tener la demo lista a tiempo? Si *hay un* esfuerzo extra que podría aplicar, entonces ella y su equipo no deben haber aplicado todo su esfuerzo antes. Deben haber estado reservando algo de esfuerzo. ¹

La promesa de intentarlo es una admisión de que te has estado conteniendo, de que tienes una reserva de esfuerzo extra que puedes aplicar. La promesa de intentarlo es una admisión de que el objetivo es alcanzable mediante la aplicación de este esfuerzo extra; además, es un compromiso de aplicar ese esfuerzo extra para lograr el objetivo. Por lo tanto, al prometer que lo vas a intentar te estás comprometiendo a conseguirlo. Esto hace que la carga recaiga sobre ti. Si tu "intento" no te lleva al resultado deseado, habrás fracasado.

¿Tienes una reserva extra de energía que has estado reteniendo? Si aplicas esas reservas, ¿serás capaz de cumplir el objetivo? O, al prometer que lo vas a intentar, ¿te estás exponiendo al fracaso?

Al prometer que lo intentarás, estás prometiendo cambiar tus planes. Después de todo, los planes que tenías eran insuficientes. Al prometer que lo vas a intentar, estás diciendo que tienes un nuevo plan. ¿Cuál es ese nuevo plan? ¿Qué cambio vas a hacer en tu comportamiento? ¿Qué cosas diferentes vas a hacer porque ahora lo estás "intentando"?

Si no tienes un nuevo plan, si no haces un cambio en tu comportamiento, si haces todo exactamente igual que antes de prometer "intentarlo", entonces ¿qué significa intentarlo?

Si no estás guardando algo de energía en reserva, si no tienes un nuevo plan,

si no vas a cambiar tu comportamiento, y si estás razonablemente seguro de tu estimación original, entonces prometer que lo vas a intentar es fundamentalmente deshonesto. Estás *mintiendo*. Y probablemente lo estás haciendo para salvar la cara y evitar una confrontación.

El enfoque de Paula fue mucho mejor. Seguía recordándole a Mike que la estimación del equipo era incierta. Siempre decía "ocho o nueve semanas". Insistió en la incertidumbre y nunca se echó atrás. Nunca sugirió que podría haber algún esfuerzo adicional, o algún nuevo plan, o algún cambio de comportamiento que pudiera reducir esa incertidumbre.

Tres semanas después...

Mike: "Paula, la demostración es dentro de tres semanas, y los clientes exigen ver el FILE UPLOAD funcionando".

Paula: "Mike, eso no está en la lista de características que acordamos". MIKE: "Lo sé, pero lo exigen".

Paula: "Vale, eso significa que o bien SINGLE SIGN-ON o bien BACKUP tienen que ser eliminados de la demo".

Mike: "¡Claro que no! Esperan que esas características también funcionen".

Paula: "Entonces, esperan que todas las características funcionen. ¿Es eso lo que me estás diciendo? Te dije que eso no iba a suceder".

MIKE: "Lo siento, Paula, pero el cliente no va a ceder en esto. Quieren verlo todo".

Paula: "Eso no va a pasar, Mike. Simplemente no va a pasar". Mike: "Vamos Paula, ¿no podéis al menos intentarlo?"

Paula: "Mike, podría *intentar* levitar. Podría *intentar* cambiar el plomo en oro. Podría *intentar* cruzar el Atlántico a nado. ¿Crees que lo lograría?"

MIKE: "Ahora estás siendo poco razonable. No estoy pidiendo el *imposible*".

Paula: "Sí, Mike, *lo eres*".

(Mike sonríe, asiente y se da la vuelta para alejarse).

Mike: "Tengo fe en ti Paula; sé que no me vas a defraudar". Paula: (hablando a la espalda de Mike) "Mike, estás soñando. Esto *no es* va a terminar bien".

(Mike se limita a saludar sin darse la vuelta).

Agresión pasiva

Paula tiene que tomar una decisión interesante. Ella sospecha que Mike no

le está diciendo a Don acerca de sus estimaciones. Ella podría dejar que Mike caminara por el extremo del acantilado. Ella podría asegurarse de que las copias de todos los memorandos apropiados estaban en el archivo, de modo que cuando el desastre golpea ella puede mostrar *lo que* le dijo a Mike, y *cuando* ella le dijo. Esto es una agresión pasiva. Ella simplemente dejaría que Mike se colgara.

O puede intentar evitar el desastre comunicándose directamente con Don. Esto es arriesgado, sin duda, pero también es lo que realmente significa ser un jugador de equipo. Cuando un tren de mercancías se acerca a ti y eres el único que puede verlo, puedes salirte de la vía tranquilamente y ver cómo todos los demás son arrollados, o puedes gritar "¡Tren! Sal de la vía!"

Dos días después...

Paula: "Mike, ¿le has dicho a Don lo de mis estimaciones? ¿Le ha dicho al cliente que la demo no tendrá la función de CARGA DE ARCHIVOS funcionando?"

MIKE: "Paula, dijiste que lo harías funcionar para mí".

Paula: "No, Mike, no lo hice. Te dije que era imposible. Aquí tienes una copia del memorándum que te envié después de nuestra charla".

Mike: "Sí, pero ibas a *intentarlo* de todas formas, ¿no?"

Paula: "Ya hemos tenido esa discusión Mike. ¿Recuerdas el oro y el plomo?"

Mike: (suspira) "Mira, Paula, sólo tienes que hacerlo. Sólo tienes que hacerlo. Por favor, haz lo que sea necesario, pero sólo tienes que hacer que esto suceda por mí".

Paula: "Mike. Te equivocas. No tengo que hacer que esto suceda por ti. Lo que *tengo que* hacer, si no lo haces, es decírselo a Don".

Mike: "Eso sería pasar por encima de mí, no lo harías". Paula:

"No quiero Mike, pero lo haré si me obligas". Mike: "Oh, Paula..."

Paula: "Mira, Mike, las características *no* se van a hacer a tiempo para la demo. Tienes que meterte esto en la cabeza. Deja de intentar convencerme de que trabaje más. Deja de engañarte a ti mismo pensando que voy a sacar un conejo de un sombrero. Afronta el hecho de que tienes que decírselo a Don, y tienes que decírselo *hoy*".

MIKE: (Con los ojos muy abiertos) "¿Hoy?"

Paula: "Sí, Mike. Hoy. Porque mañana espero tener una reunión contigo y con Don sobre qué características incluir en la demo. Si esa reunión no se produce mañana, me veré obligada a ir yo misma a ver a Don. Aquí tienes una copia del memorándum que explica precisamente eso".

MIKE: "¡Sólo te estás cubriendo el culo!"

Paula: "Mike, estoy tratando de cubrir nuestros *dos* traseros. Te imaginas la debacle si el cliente viene aquí esperando una demostración completa y no podemos cumplirla?".

¿Qué pasa al final con Paula y Mike? Te dejo que elabores las posibilidades. La cuestión es que Paula se ha comportado de forma muy profesional.

Ha dicho que no en todos los momentos adecuados, y de todas las maneras correctas. Ha dicho que no cuando se le ha presionado para que modifique sus estimaciones. Ha dicho no cuando se le ha manipulado, engatusado y rogado. Y, lo que es más importante, dijo no al autoengaño e inacción de Mike. Paula jugaba para el equipo. Mike necesitaba ayuda, y ella utilizó todos los medios a su alcance para ayudarle.

El coste de decir sí

La mayoría de las veces queremos decir que sí. De hecho, los equipos sanos se esfuerzan por encontrar la manera de decir que sí. El director y los desarrolladores de los equipos bien gestionados negociarán entre sí hasta llegar a un plan de acción de mutuo acuerdo.

Pero, como hemos visto, a veces la única manera de llegar al sí *correcto* es no tener miedo de decir no.

Considere la siguiente historia que John Blanco publicó en su blog.² Se reproduce aquí con permiso. Al leerla, pregúntese cuándo y cómo debería haber dicho que no.

¿Es imposible un buen código?

Cuando llegas a la adolescencia decides que quieres ser desarrollador de software. Durante tus años de instituto, aprendes a escribir software utilizando principios orientados a objetos. Cuando te gradúas en la universidad, aplicas todos los principios que has aprendido a áreas como la inteligencia artificial o los gráficos 3D.

Y cuando llegas al circuito profesional, comienzas tu interminable búsqueda para escribir un código de calidad comercial, mantenible y "perfecto" que resista la prueba del tiempo.

Calidad comercial. Huh. Eso es bastante divertido.

Me considero afortunado, *me encantan los patrones* de diseño. Me gusta estudiar la teoría de la perfección de la codificación. No tengo ningún problema en iniciar una discusión de una hora sobre por qué la elección de la jerarquía de herencia de mi compañero de XP es errónea, que HAS-A es mejor que IS-A en muchos casos. Pero algo me ha estado molestando últimamente y me estoy preguntando algo...

... ¿Es imposible un buen código en el desarrollo de software moderno?

La típica propuesta de proyecto

Como desarrollador contratado a tiempo completo (y a tiempo parcial), me paso los días (y las noches) desarrollando aplicaciones móviles para clientes. Y lo que he aprendido en los muchos años que llevo haciendo esto

es que las exigencias del trabajo de los clientes me impiden escribir las aplicaciones de verdadera calidad que me gustaría.

Antes de empezar, permítanme decir que no es por falta de intentos. Me encanta el tema del código limpio. No conozco a nadie que persiga ese diseño de software perfecto como yo. Es la ejecución lo que me resulta más esquivo, y no por la razón que crees.

Déjame contarte una historia.

Hacia finales del año pasado, una empresa bastante conocida puso un RFP (Request for Proposal) para tener una aplicación construida para ellos. Se trata de un gran minorista, pero en aras del anonimato lo llamaremos Gorilla Mart. Dicen que necesitan crear una presencia en el iPhone y que les gustaría tener una aplicación producida para el Viernes Negro. ¿El problema? Ya es el 1 de noviembre. Eso deja poco menos de 4 semanas para crear la aplicación. Ah, y en este momento Apple sigue tardando dos semanas en aprobar las aplicaciones. (Ah, los buenos tiempos.) Así que, espera, esta aplicación tiene que ser escrita en... ¿¡DOS SEMANAS!?

Sí. Tenemos dos semanas para escribir esta aplicación. Y, por desgracia, hemos ganado la licitación. (En los negocios, la importancia del cliente importa.) Esto va a pasar.

"Pero no pasa nada", dice el ejecutivo nº 1 de Gorilla Mart. "La aplicación es sencilla. Solo tiene que mostrar a los usuarios algunos productos de nuestro catálogo y permitirles buscar las ubicaciones de las tiendas. Ya lo hacemos en nuestra web. También te daremos los gráficos. Probablemente puedas... ¿cuál es la palabra? sí, ¡codificarla!"

El ejecutivo de Gorilla Mart nº 2 interviene. "Y solo necesitamos un par de cupones que el usuario pueda mostrar en la caja registradora. La aplicación será de usar y tirar. Vamos a sacarla adelante, y luego para la fase II haremos algo más grande y mejor desde cero".

Y entonces ocurre. A pesar de los años de constantes recordatorios de que cada función que pide un cliente siempre será más compleja de escribir que de explicar, vas a por ello. Realmente crees que esta vez se puede hacer en dos semanas. Sí. ¡Podemos hacerlo! ¡Esta vez es diferente! Son sólo unos pocos gráficos y una llamada de servicio para obtener la ubicación de una tienda. ¡XML! No hay problema. Podemos hacerlo. ¡Estoy emocionado! ¡Vamos!

Sólo hace falta un día para que tú y la realidad os conozcáis de nuevo.

Yo: Entonces, ¿puede darme la información que necesito para llamar a su servicio web de localización de tiendas? El cliente: ¿Qué es un servicio web?

Yo:

Y así es exactamente como ocurrió. Su servicio de localización de tiendas, que se encuentra justo donde se supone que está en la esquina superior derecha de su sitio web, no es un servicio web. Está generado por código Java. ¡X-nay con la API-ay. Y para colmo, está alojado por un socio estratégico de Gorilla Mart.

Entra la nefasta "tercera parte".

En términos de clientes, una "tercera parte" es algo parecido a Angelina Jolie. A pesar de la promesa de que podrás tener una conversación esclarecedora durante una buena comida y, con suerte, ligar

..... después
s -lo siento, es
no va a suceder. Vas a tener que fantasear con ello mientras te ocupas de tus asuntos.

En mi caso, lo único que pude sacar de Gorilla Mart fue una instantánea de sus listados de tiendas actuales en un archivo de Excel. Tuve que escribir el código de búsqueda de tiendas desde cero.

El doble golpe llegó ese mismo día: Querían los datos de los productos y los cupones en línea para poder cambiarlos semanalmente. Así que, ¡a por la codificación! Dos semanas para escribir una aplicación para el iPhone ahora

..... se convierten en dos semanas para escribir una aplicación para el iPhone, un backend PHP, e integrarlos juntos. ¿Qué? ¿También quieren que me encargue del control de calidad?

Para compensar el trabajo extra, la codificación tendrá que ir un poco más rápido. Olvida esa fábrica abstracta. Usa un gran bucle for en lugar del compuesto, ¡no hay tiempo!

Un buen código se ha vuelto imposible.

Dos semanas para la finalización

Déjenme decirles que esas dos semanas fueron bastante miserables. En primer lugar, dos de los días se eliminaron debido a las reuniones de todo el día para mi próximo proyecto. (Eso amplía lo corto que iba a ser el plazo). Al final, realmente tenía ocho días para hacer las cosas. La primera semana trabajé 74 horas y la siguiente... Dios... Ni siquiera lo recuerdo, se ha borrado de mis sinapsis.

Probablemente sea algo bueno.

Me pasé esos ocho días escribiendo código con furia. Utilicé todas las herramientas disponibles para hacerlo: copiar y pegar (también conocido como código reutilizable), números mágicos (para evitar la duplicación de definir constantes y luego, ¡joder!, volver a escribirlas), y ¡absolutamente NINGÚN test unitario! (¿Quién necesita barras rojas en un momento como éste?, ¡sólo me desmotivaría!)

Era un código bastante malo y nunca tuve tiempo de refactorizarlo. Sin embargo, teniendo en cuenta el tiempo, en realidad era bastante estelar, y al fin y al cabo era un código "de usar y tirar", ¿no? ¿Te suena algo de esto? Pues espera, la cosa se pone mejor.

Cuando estaba dando los últimos toques a la aplicación (los últimos toques eran escribir todo el código del servidor), empecé a mirar el código base y me pregunté si tal vez merecía la pena.

Después de todo, la aplicación estaba terminada. He sobrevivido.

"Oye, acabamos de contratar a Bob, y está muy ocupado y no ha podido hacer la llamada, pero dice que deberíamos exigir a los usuarios que proporcionen su dirección de correo electrónico para conseguir los cupones. No ha visto la aplicación, pero cree que sería una gran idea. También queremos un sistema de informes para obtener esos correos electrónicos del servidor. Uno que sea bonito y no demasiado caro. (Espera, esa última parte era de Monty Python.) Hablando de cupones, tienen que poder caducar después de un número de días que especifiquemos. Ah, y....."

Demos un paso atrás. ¿Qué sabemos sobre lo que es un buen código? Un buen código debe ser extensible. Mantenible. Debe prestarse a ser modificado. Debe leerse como prosa. Bueno, esto no era un buen código.

Otra cosa. Si quieres ser un mejor desarrollador, debes tener siempre presente esto, inevitablemente: El cliente siempre ampliará el plazo. Siempre querrán más funciones. Siempre querrán cambios-tarde. Y esta es la fórmula de lo que se puede esperar:

$$\begin{aligned} & (\text{Número de ejecutivos})^2 \\ & + 2 * \# \text{ de nuevos ejecutivos} \\ & + \# \text{ de Bob's Kids} \\ & = \text{DÍAS AÑADIDOS EN EL ÚLTIMO MOMENTO} \end{aligned}$$

Ahora, los ejecutivos son gente decente. Yo creo que sí. Mantienen a su familia (suponiendo que Satanás haya aprobado que tengan una). Quieren que la aplicación tenga éxito (¡tiempo de promoción!). El problema es que todos quieren reclamar directamente el éxito del proyecto. Cuando todo está dicho y hecho, todos quieren señalar alguna característica o decisión de diseño que puedan llamar suya.

Así que, volviendo a la historia, añadimos un par de días más al proyecto y conseguimos terminar la función de correo electrónico. Y entonces me derrumbé de agotamiento.

Los clientes nunca se preocupan tanto como tú

Los clientes, a pesar de sus protestas, a pesar de su aparente urgencia, nunca se preocupan tanto como tú de que la aplicación esté a tiempo. La tarde en que di por terminada la aplicación, envié un correo electrónico con la compilación final a todos los interesados, ejecutivos (¡silencio!), gerentes, etc. "¡YA ESTÁ HECHO! ¡OS TRAIGO LA V1.0! ALABADO SEA TU NOMBRE". Le di a "Enviar", me recosté en mi silla y, con una sonrisa de satisfacción, empecé a fantasear con

que la empresa me subiría a hombros y encabezaría una procesión por la calle 42 mientras me coronaban como "El mejor desarrollador de la historia". Como mínimo, mi cara aparecería en toda su publicidad, ¿no?

Es curioso, no parecían estar de acuerdo. De hecho, no estaba seguro de lo que pensaban. No escuché nada. Ni una palabra. Resulta que la gente de Gorilla Mart estaba ansiosa y ya había pasado a lo siguiente.

¿Crees que miento? Comprueba esto. Fui a la tienda de Apple sin rellenar una descripción de la aplicación. Había solicitado una a Gorilla Mart, y no me habían contestado y no había tiempo para esperar. (Véase el párrafo anterior.) Les escribí de nuevo. Y otra vez. Puse a algunos de nuestros directivos a trabajar en ello. Dos veces me contestaron y dos veces me dijeron: "¿Qué necesitas otra vez?". ¡NECESITO LA DESCRIPCIÓN DE LA APLICACIÓN!

Una semana después, Apple empezó a probar la aplicación. Este suele ser un momento de alegría, pero en cambio fue un momento de temor mortal. Como era de esperar, ese mismo día la aplicación fue rechazada. Fue la excusa más triste y pobre para permitir un rechazo que pueda imaginar: "A la aplicación le falta una descripción". Funcionalmente perfecto; no hay descripción de la aplicación. Y por esta razón Gorilla Mart no tenía su app lista para el Black Friday. Estaba bastante molesto.

Había sacrificado a mi familia por un súper sprint de dos semanas, y nadie en Gorilla Mart podía molestarse en crear una descripción de la aplicación con una semana de tiempo. Nos la dieron una hora después del rechazo; al parecer, esa era la señal para ponerse manos a la obra.

Si antes estaba molesto, a la semana y media me ponía lívido. Verás, todavía no nos habían dado datos reales. Los productos y cupones del servidor eran falsos. Imaginarios. El código del cupón era 1234567890. Ya sabes, falso. (Bolonia se escribe baloney cuando se usa en ese contexto, BTW.)

Y fue esa fatídica mañana cuando revisé el Portal y ¡LA APP ESTABA DISPONIBLE! ¡Con datos falsos y todo! Grité horrorizado y llamé a quien pude y grité: "¡Necesito los datos!" y la mujer al otro lado me preguntó si necesitaba a los bomberos o a la policía, así que colgué al 911. Pero luego llamé a Gorilla Mart y dije: "¡Necesito datos!". Y nunca olvidaré la respuesta:

Oh, hola John. Tenemos un nuevo vicepresidente y hemos decidido no lanzarlo. Sácalo de la App Store, ¿quieres?

Al final, resultó que al menos 11 personas registraron sus direcciones de correo electrónico en la base de datos, lo que significaba que había 11 personas que potencialmente podrían entrar en un Gorilla Mart con un cupón de iPhone falso a cuestas. Vaya, eso podría ponerse feo.

Cuando todo estaba dicho y hecho, el cliente había dicho una cosa correctamente todo el tiempo: El código era un desecho. El único problema es que, en primer lugar, nunca fue liberado.

¿Resultado? Prisa por terminar, lentitud en la comercialización

La lección de la historia es que los interesados, ya sea un cliente externo o la dirección interna, han descubierto cómo conseguir que los desarrolladores escriban código rápidamente.

¿Efectivamente? No. ¿Rápido? Sí. Así es como funciona:

semana, justo cuando habías trabajado un turno de 20 horas para que todo estuviera bien. Es como el burro y la zanahoria, salvo que no te tratan tan bien como al burro.

Es un libro de jugadas brillante. ¿Puedes culparlos por pensar que funciona? Pero no ven el código de Dios. Y así sucede, una y otra vez, a pesar de los resultados.

En una economía globalizada, en la que las corporaciones se rigen por el todopoderoso dólar y el aumento del precio de las acciones implica despidos, personal sobrecargado de trabajo y deslocalización, esta estrategia que te he mostrado de recortar los costes de los desarrolladores — está haciendo que el buen código sea obsoleto. Como desarrolladores, nos van a pedir/decir/convencer para que escribamos el doble de código en la mitad de tiempo si no tenemos cuidado.

Código imposible

En la historia, cuando John pregunta "¿Es imposible un buen código?", en realidad está preguntando "¿Es imposible la profesionalidad?". Después de todo, no fue sólo el código el que sufrió en su historia de disfunción. Fue su familia, su empleador, su cliente y los usuarios. *Todos* perdieron ³en esta aventura. Y perdieron debido a la falta de profesionalidad.

Entonces, ¿quién actuó de forma poco profesional? John deja claro que cree que fueron los ejecutivos de Gorilla Mart. Después de todo, su libro de jugadas era una acusación bastante clara de su mal comportamiento. ¿Pero *fue* su comportamiento malo? No lo creo.

La gente de Gorilla Mart quería la opción de tener una aplicación para el iPhone en el Black Friday. Estaban dispuestos a pagar para tener esa opción. Encontraron a alguien dispuesto a proporcionar esa opción. Entonces, ¿cómo puedes culparlos?

Sí, es cierto, hubo algunos fallos de comunicación. Al parecer, los ejecutivos no sabían lo que era realmente un servicio web, y se produjeron todos los problemas normales de que una parte de una gran empresa no sepa lo que hace otra. Pero todo eso era de esperar. John incluso lo admite cuando dice: "A pesar de años de constantes recordatorios de que cadafunción

que pide un cliente siempre será más complejo de escribir que de explicar.

.
."

Así que si el culpable no fue Gorilla Mart, ¿entonces quién?

Tal vez fuera el empleador directo de John. John no lo dijo explícitamente, pero hubo un indicio cuando dijo, entre paréntesis, "En losnegocios, la importancia del cliente importa". Entonces, ¿el empleador de John hizo promesas irrazonables a Gorilla Mart? ¿Presionaron a John, directa o indirectamente, para que esas promesas se hicieran realidad? John no lo dice, así que sólo podemos preguntarnos.

Aun así, ¿dónde está la responsabilidad de John en todo esto? Yo pongo la culpa directamente en John. John es quien aceptó el plazo inicial de dos semanas, sabiendo perfectamente que los proyectos suelen ser más complejos de lo que parecen. John es quien aceptó la necesidad de escribir el servidor PHP. John es el que aceptó el registro del correo electrónico, y

la caducidad del cupón. John es el que trabajó días de 20 horas y semanas de 90 horas. John es el que se sustrajo de su familia y de su vida para cumplir con este plazo.

¿Y por qué hizo esto Juan? Nos lo cuenta con toda claridad: "Le di a "Enviar", me recosté en mi silla y, con una sonrisa de satisfacción, empecé a fantasear con que la compañía me subiría a hombros y encabezaría una procesión por la calle 42 mientras me coronaban como "El mejor promotor de la historia". En resumen, John intentaba ser un héroe. Vio su oportunidad de ser aclamado, y fue a por ello. Se inclinó y agarró el anillo de bronce.

Los profesionales suelen ser héroes, pero no porque lo intenten. Los profesionales se convierten en héroes cuando consiguen hacer un trabajo bien hecho, a tiempo y dentro del presupuesto. Al intentar convertirse en el hombre del momento, en el salvador del día, John no estaba actuando como un profesional.

John debería haber dicho que no al plazo original de dos semanas. O si no, debería haber dicho que no cuando descubrió que no había servicio web. Debería haber dicho que no a la solicitud de registro por correo electrónico y a la caducidad de los cupones. Debería haber dicho que no a cualquier cosa que requiriera horribles horas extras y sacrificios.

Pero sobre todo, John debería haber dicho que no a su propia decisión interna de que la única manera de terminar este trabajo a tiempo era hacer un gran lío. Fíjate en lo que dijo John sobre el buen código y las pruebas unitarias:

"Para compensar el trabajo extra, la codificación tendrá que ir un poco más rápido. Olvida esa fábrica abstracta. Usa un gran bucle for en lugar del compuesto, ¡no hay tiempo!"

Y otra vez:

"Me pasé esos ocho días escribiendo código con furia. Utilicé todas las herramientas disponibles para hacerlo: copiar y pegar (también conocido como código reutilizable), números mágicos (para evitar la duplicación de la definición de constantes y luego, ¡juego!, volver a escribirlas), y ¡absolutamente NINGÚN test unitario! (¿Quién necesita barras rojas en un momento como éste?, ¡sólo me desmotivaría!)"

Decir que sí a esas decisiones fue el verdadero quid del fracaso. John aceptó que la única forma de tener éxito era comportarse de forma poco profesional, por lo que cosechó la recompensa correspondiente.

Esto puede sonar duro. No es esa la intención. En capítulos anteriores he descrito cómo he cometido el mismo error en mi carrera, más de una vez. La tentación de ser un héroe y "resolver el problema" es enorme. Lo que todos tenemos que comprender es que decir que sí a abandonar nuestras disciplinas profesionales *no* es la forma de resolver los problemas.

Abandonar esas disciplinas es la forma de crear problemas. Con esto, por fin puedo responder a la pregunta inicial de John: "¿Es imposible un buen código? ¿Es imposible la profesionalidad?"
Respuesta: Yo digo que *no*.

3. Decir sí



¿Sabías que yo inventé el buzón de voz? Es cierto. En realidad éramos tres los que teníamos la patente del correo de voz. Ken Finder, Jerry Fitzpatrick y yo. Fue a principios de los 80, y trabajábamos para una empresa llamada Teradyne. Nuestro director general nos había encargado un nuevo tipo de producto, y nosotros inventamos "El Recepcionista Electrónico", o ER para abreviar.

Todos sabéis lo que es ER. Urgencias es una de esas horribles máquinas que atienden el teléfono de las empresas y te hacen todo tipo de preguntas descerebradas que tienes que responder pulsando botones. ("Para inglés, pulse 1").

Nuestra ER atendía el teléfono de una empresa y te pedía que marcaras el nombre de la persona que querías. Le pedía que pronunciara su nombre y luego llamaba a la persona en cuestión. Anunciaba la llamada y preguntaba si debía ser aceptada. En caso afirmativo, conectaría la llamada y la dejaría.

Podías decirle a Urgencias dónde estabas. Podías darle varios números de teléfono para que probara. Así, si estabas en la oficina de otra persona, Urgencias podría encontrarte. Si estaba en su casa, ER podría encontrarlo. Si estabas en otra ciudad, ER podría encontrarte. Y, al final, si ER no podía encontrarte, tomaba un mensaje. Ahí es donde entraba el buzón de voz.

Curiosamente, Teradyne no supo cómo vender ER. El proyecto se quedó sin presupuesto y se transformó en algo que sí sabíamos vender:

CDS, TheCraft Dispatch System, para enviar a los reparadores telefónicos a su próximo trabajo. Y Teradyne también abandonó la patente sin decírnoslo. (!) El actual titular de la patente la presentó tres meses después que nosotros. (!!)

Mucho después de la transformación de ER en CDS, pero mucho antes de que me enterara de que la patente había sido retirada. Esperé en un árbol al director general de la empresa. Teníamos un gran roble delante del edificio. Me subí a él y esperé a que llegara su Jaguar. Me encontré con él en la puerta y le pedí unos minutos. Me lo concedió.

Le dije que teníamos que volver a poner en marcha el proyecto de urgencias. Le dije que estaba seguro de que podría ganar dinero. Me sorprendió diciendo: "De acuerdo, Bob, elabora un plan. Muéstrame cómo puedo ganar dinero. Si lo haces, y me lo creo, volveré a poner en marcha ER".

No me lo esperaba. Esperaba que dijera: "Tienes razón, Bob. Voy a volver a poner en marcha ese proyecto y voy a averiguar cómo ganar dinero con él". Pero no. Puso la carga de nuevo en mí. Y era una carga que me resultaba ambivalente. Después de todo, yo era un tipo de software, no un tipo de dinero. Quería trabajar en el proyecto de ER, no ser responsable de los beneficios y las pérdidas.

Pero no quería mostrar mi ambivalencia. Así que le di las gracias y salí de su despacho con estas palabras:

"Gracias Russ. Estoy comprometido. . . supongo".

Con esto, permítanme presentarles a Roy Osherove, que les dirá lo patético de esa declaración.

Un lenguaje de compromiso

Por Roy

Osherove Say.

Significa. Hacer.

El compromiso consta de tres partes.

1. *Dices* que lo harás.
2. Lo *dices en serio*.
3. *Realmente* lo haces.

Pero, ¿cuántas veces nos encontramos con otras personas (no con nosotros mismos, por supuesto) que nunca llegan hasta el final de estas tres etapas?

- **Le preguntas al informático** por qué la red es tan lenta y te dice "Sí. Tenemos que conseguir nuevos routers". Y *sabes que* nunca pasará nada en esa categoría.

- **Le pides a un miembro del equipo** que realice algunas pruebas manuales antes de facturar el código fuente, y te responde: "Claro. Espero poder hacerlo al final del día". Y de alguna manera *sientes que tendrás* que preguntar mañana si realmente se realizaron pruebas antes del check-in.
- **Tu jefe** entra en la habitación y murmura: "tenemos que ir más rápido". Y tú *sabes* que en realidad se refiere a que TÚ tienes que ir más rápido. No va a hacer nada al respecto.

Hay muy pocas personas que, cuando dicen algo, lo dicen en serio y luego lo hacen. Hay algunos que dicen cosas y las dicen en serio, pero nunca las hacen. Y hay mucha más gente que promete cosas y ni siquiera *tiene la intención* de hacerlas. ¿Has oído alguna vez a alguien decir: "Tío, tengo que perder peso", y sabes que no va a hacer nada al respecto? Sucede todo el tiempo.

¿Por qué seguimos teniendo esa extraña sensación de que, la mayoría de las veces, la gente no está realmente comprometida con la consecución de algo?

Peor aún, a menudo nuestra intuición puede fallar. A veces *nos gustaría* creer que alguien quiere realmente lo que dice cuando en realidad no es así. *Nos* gustaría creer a un desarrollador cuando dice, acuciado por la esquina, que puede terminar esa tarea de dos semanas en una semana, pero no deberíamos hacerlo.

En lugar de confiar en nuestras tripas, podemos utilizar algunos trucos relacionados con el lenguaje para intentar averiguar si la gente quiere realmente lo que dice. Y al cambiar lo que decimos, podemos empezar a ocuparnos de los pasos 1 y 2 de la lista anterior por nuestra cuenta. Cuando *decimos que* nos vamos a comprometer con algo, tenemos que decirlo *en serio*.

Reconocer la falta de compromiso

Deberíamos fijarnos en el lenguaje que utilizamos cuando nos comprometemos a hacer algo, como señal reveladora de lo que está por

venir. En realidad, se trata más bien de buscar *palabras* específicas en lo que decimos. Si no se encuentran esas palabritas mágicas, lo más probable es que no queramos decir lo que decimos, o que no lo creamos factible.

Estos son algunos ejemplos de palabras y frases que deben buscarse y que son signos reveladores de falta de compromiso:

- **Necesidad.** "Necesitamos hacer esto". "Necesito perder peso". "Alguien debería hacerlo".
- **Hope\wish.** "Espero tener esto hecho para mañana". "Espero que podamos volver a vernos algún día." "Ojalá tuviera tiempo para

eso". "Ojalá este ordenador fuera más rápido."

- **Veamos.** (no seguido de "yo...") "Quedemos algún día".
"Terminemos esto".

Cuando empieces a buscar estas palabras, verás que empiezas a detectarlas en casi todas partes a tu alrededor, e incluso en las cosas que dices a los demás.

Verás que tendemos a estar muy ocupados y a no responsabilizarnos de las cosas.

Y eso *no está* bien cuando tú o alguien más depende de esas promesas como parte del trabajo. Sin embargo, ya has dado el primer paso: empieza a reconocer la falta de compromiso a tu alrededor y en ti.

Hemos oído cómo suena la falta de compromiso. ¿Cómo reconocemos el verdadero compromiso?

¿A qué suena el compromiso?

Lo que es común en las frases de la sección anterior es que o bien asumen que las cosas están fuera de "mis" manos o no asumen la responsabilidad personal. En cada uno de estos casos, las personas se comportan como si fueran *víctimas* de una situación en lugar de tener el control de la misma.

La verdad es que *tú, personalmente*, SIEMPRE tienes algo que está bajo *tu* control, por lo que siempre hay *algo* que puedes comprometerte plenamente a hacer.

El ingrediente secreto para reconocer el compromiso real es buscar frases que suenen así Voy a... para... (ejemplo: Terminaré esto para el martes).

¿Qué es lo importante de esta frase? *Estás afirmando un hecho sobre algo que TÚ harás con un tiempo final claro.* No estás hablando de nadie más que de ti mismo. Estás hablando de una *acción que harás* tú. No lo harás "*posiblemente*", o "*podrías llegar a hacerlo*"; lo lograrás.

No hay (técnicamente) ninguna salida a este compromiso verbal. Has dicho que lo harás y ahora sólo es posible un resultado binario: o lo haces,

o no lo haces. Si no lo haces, la gente puede exigirte que cumplas tus promesas. Te sentirás *mal* por no haberlo hecho. Te sentirás *incómodo por* contarle a alguien que no lo has hecho (si ese alguien te ha oído prometer que lo harás).

Da miedo, ¿verdad?

Estás asumiendo toda la responsabilidad de algo, frente a una audiencia de al menos una persona. No eres tú el que se pone delante del espejo o de la pantalla del ordenador. Eres tú, frente a otro ser humano, y diciendo que lo harás. Ese es el comienzo del compromiso. Ponerte en la situación que te obliga a hacer algo.

Has cambiado el lenguaje que utilizas por un lenguaje de compromiso, y

eso te ayudará a superar las dos etapas siguientes: el significado y el seguimiento.

He aquí una serie de razones por las que es posible que no *quieras*, o no sigas, con algunas soluciones.

No funcionaría porque dependo de la persona X para hacer esto

Sólo puedes comprometerte con cosas sobre las que tienes *pleno control*. Por ejemplo, si tu objetivo es terminar un módulo que también depende de otro equipo, no puedes comprometerte a terminar el módulo con plena integración con el otro equipo. Pero *puedes comprometerte a realizar acciones específicas que te lleven a tu objetivo*. Podrías:

- Siéntate durante una hora con Gary, del equipo de infraestructura, para entender tus dependencias.
- Crea una interfaz que abstraiga la dependencia de tu módulo de la infraestructura del otro equipo.
- Reúnete al menos tres veces esta semana con el responsable de la compilación para asegurarte de que tus cambios funcionan bien en el sistema de compilación de la empresa.
- Cree su propia compilación personal que ejecute sus pruebas de integración para el módulo.

¿Ves la diferencia?

Si el objetivo final depende de otra persona, debes comprometerte con acciones específicas que te acerquen al objetivo final.

No funcionaría porque realmente no sé si se puede hacer

Si no se puede hacer, aún puedes comprometerte con acciones que te acerquen al objetivo. Averiguar si se puede hacer puede ser una de las acciones a las que comprometerse.

En lugar de comprometerse a arreglar los 25 errores restantes antes del lanzamiento (lo que puede no ser posible), puede comprometerse a realizar estas acciones específicas que le acercan a ese objetivo:

- Revisa los 25 errores e intenta recrearlos.
- Siéntese con el QA que encontró cada error para ver una reproducción de ese error.
- Dedica todo el tiempo que tengas esta semana a intentar arreglar cada fallo.

No funcionaría porque a veces no lo consigo

Eso sucede. Puede ocurrir algo inesperado, y así es la vida. Pero tú sigues queriendo estar a la altura de las expectativas. En ese caso, es hora de cambiar las expectativas, lo antes posible.

Si no puedes cumplir tu compromiso, lo más importante es que avises

cuanto antes a quien te hayas comprometido.

Cuanto antes levante la bandera ante todas las partes interesadas, más probable será que haya tiempo para que el equipo se detenga, reevalúe las acciones que se están llevando a cabo y decida si se puede hacer o cambiar algo (en términos de prioridades, por ejemplo). Al hacer esto, el compromiso puede seguir cumpliéndose, o puede cambiarse a otro compromiso.

Algunos ejemplos son:

- Si fijas una reunión para el mediodía en una cafetería del centro con un colega y te quedas atascado en el tráfico, dudas de que puedas cumplir tu compromiso de llegar a tiempo. Puedes llamar a tu colega en cuanto te des cuenta de que puedes llegar tarde y avisarle. Tal vez puedas encontrar un lugar más cercano para quedar, o quizás posponer la reunión.
- Si te comprometiste a resolver un fallo que pensabas que era solucionable y te das cuenta en algún momento de que el fallo es mucho más horrible de lo que se pensaba, puedes levantar la bandera. El equipo puede entonces decidir un curso de acción para hacer ese compromiso (emparejamiento, picoteo en soluciones potenciales, lluvia de ideas) o cambiar la prioridad y pasarte a otro fallo más sencillo.

Un punto importante aquí es: Si no le dices a nadie sobre el problema potencial tan pronto como sea posible, no estás dando a nadie la oportunidad de ayudarte cumplir con su compromiso.

Resumen

Crear un lenguaje de compromiso puede sonar un poco aterrador, pero puede ayudar a resolver muchos de los problemas de comunicación a los que se enfrentan los programadores hoy en día: estimaciones, plazos y contratiempos de comunicación cara a cara. Se te tomará como un desarrollador serio que cumple con su palabra, y eso es una de las mejores cosas que puedes esperar en nuestra industria.

Aprender a decir "sí"

Le pedí a Roy que contribuyera con ese artículo porque me tocó la fibra sensible. Llevo algún tiempo predicando sobre cómo decir no. Pero es igual de importante aprender a decir que sí.

La otra cara del "intento"

Imaginemos que Pedro es responsable de algunas modificaciones en el motor de clasificación. Ha calculado en privado que estas modificaciones le llevarán cinco o seis días. También cree que redactar la documentación de las modificaciones le llevará unas cuantas horas. El lunes por la

mañana, su jefa, Marge, le pregunta por el estado de las cosas.

Marge: "Peter, ¿tendrás las modificaciones del motor de clasificación hechas para el viernes?" Peter: "Creo que es factible".

Marge: "¿Incluirá eso la documentación?"

Peter: "Intentaré hacer eso también".

Tal vez Marge no pueda oír los titubeos en las declaraciones de Peter, pero lo cierto es que él no se compromete mucho. Marge hace preguntas que exigen respuestas booleanas, pero las respuestas booleanas de Peter son difusas.

Fíjate en el abuso de la palabra intentar. En el último capítulo utilizamos la definición de "esfuerzo extra" de intentar. Aquí, Pedro está usando la definición de "tal vez, tal vez no".

Peter estaría mejor respondiendo así:

Marge: "Peter, ¿tendrás las modificaciones del motor de clasificación hechas para el viernes?" Peter: "Probablemente, pero podría ser el lunes".

Marge: "¿Incluirá eso la documentación?"

Pedro: "La documentación me llevará unas horas más, así que el lunes es posible, pero podría ser hasta el martes".

En este caso, el lenguaje de Peter es más honesto. Está describiendo su propia incertidumbre a Marge. Puede que Marge sea capaz de afrontar esa incertidumbre. Por otro lado, puede que no.

Comprometerse con la disciplina

Marge: "Peter, necesito un sí o un no definitivo. ¿Tendrás el motor de clasificación terminado y documentado para el viernes?"

Esta es una pregunta perfectamente justa para Marge. Ella tiene un horario que mantener, y necesita una respuesta binaria sobre el viernes. ¿Cómo debería responder Peter?

Peter: "En ese caso, Marge, tendré que decir que no. Lo más pronto que puedo ser *seguro* que terminaré con los mods y los docs es el martes".

Marge: "¿Te comprometes al martes?"

Pedro: "Sí, lo tendré todo listo el martes".

Pero, ¿y si Marge realmente necesita las modificaciones y la documentación para el viernes?

Marge: "Peter, el martes me da un verdadero problema. Willy, nuestro escritor técnico, estará disponible el lunes. Tiene cinco días para terminar la guía del usuario. Si no tengo los documentos del motor de

clasificación para el lunes por la mañana, no podrá terminar el manual a tiempo. ¿Puedes hacer los documentos primero?"

Pedro: "No, los mods tienen que ser lo primero, porque generamos los docs a partir de la salida de las pruebas".

Marge: "Bueno, ¿no hay alguna manera de terminar los mods y los documentos antes del lunes por la mañana?"

Ahora Peter tiene que tomar una decisión. Es muy probable que termine con las modificaciones del motor de la tasa el viernes, e incluso podría terminar los documentos antes de irse a casa el fin de semana. También *podría* hacer algunas horas de trabajo el sábado si las cosas tardan más de lo que espera. Entonces, ¿qué debería decirle a Marge?

Peter: "Mira Marge, es muy posible que pueda tener todo hecho para el lunes por la mañana si le dedico unas horas extra el sábado".

¿Resuelve eso el problema de Marge? No, simplemente cambia las probabilidades, y eso es lo que Peter tiene que decirle.

Marge: "¿Puedo contar el lunes por la mañana entonces?" Peter: "Probablemente, pero no definitivamente".

Eso podría no ser suficiente para Marge.

Marge: "Mira, Peter, realmente necesito una definición de esto. ¿Hay *alguna* manera de que te comprometas a tenerlo hecho antes del lunes por la mañana?"

Peter podría estar tentado a romper la disciplina en este punto. Podría terminar más rápido si no escribe sus pruebas. Podría terminar más rápido si no refactoriza. Podría terminar más rápido si no ejecuta la suite de regresión completa.

Aquí es donde el profesional traza la línea. En primer lugar, Peter se equivoca en sus suposiciones. *No va a terminar más rápido si no* escribe sus pruebas. *No* lo hará más rápido si no refactoriza. *No* lo hará más rápido si omite la suite de regresión completa. Años de experiencia nos han enseñado que romper las disciplinas sólo nos retrasa.

Pero en segundo lugar, como profesional tiene la responsabilidad de mantener ciertos estándares. Su código debe ser probado, y debe tener pruebas. Su código tiene que estar limpio. Y tiene que estar seguro de que no ha roto nada más en el sistema.

Peter, como profesional, ya se ha comprometido a mantener estas normas. Todos los demás compromisos que asuma deben estar subordinados a eso. Así que toda esta línea de razonamiento debe ser abortada.

Peter: "No, Marge, no hay forma de asegurar ninguna fecha antes del martes. Siento que eso te fastidie la agenda, pero es la realidad a la que

nos enfrentamos".

Marge: "Maldición. Realmente contaba con traer a este antes.
¿Estás segura?"

Pedro: "Estoy seguro de que podría ser tan tarde como el martes, sí."

Marge: "Bien, creo que iré a hablar con Willy para ver si puede reorganizar su horario".

En este caso, Marge aceptó la respuesta de Peter y empezó a buscar otras opciones. ¿Pero qué pasa si todas las opciones de Marge se han agotado?

¿Y si Peter fuera la última esperanza?

Marge: "Peter, mira, sé que esto es una gran imposición, pero realmente necesito que encuentres una manera de hacer todo esto para el lunes por la mañana. Es realmente crítico. ¿No hay algo que puedas hacer?"

Así que ahora Peter empieza a pensar en trabajar algunas horas extras importantes, y probablemente la mayor parte del fin de semana. Tiene que ser muy honesto consigo mismo sobre su resistencia y sus reservas. Es fácil *decir que vas* a hacer muchas cosas los fines de semana, pero es mucho más difícil reunir la energía suficiente para hacer un trabajo de calidad.

Los profesionales conocen sus límites. Saben cuántas horas extras pueden aplicar de forma efectiva, y saben cuál será el coste.

En este caso, Peter se siente bastante seguro de que unas cuantas horas extra durante la semana y algo de tiempo el fin de semana serán suficientes.

Peter: "Bien, Marge, te diré algo. Llamaré a casa y aclararé algunas horas extras con mi familia. Si les parece bien, terminaré esta tarea el lunes por la mañana. Incluso vendré el lunes por la mañana para asegurarme de que todo va bien con Willy. Pero luego me iré a casa y no volveré hasta el miércoles. ¿Trato?"

Esto es perfectamente justo. Peter sabe que puede hacer las modificaciones y los documentos si trabaja las horas extras. También sabe que será inútil durante un par de días después.

Conclusión:

Los profesionales no están obligados a decir que sí a todo lo que se les pide. Sin embargo, deben esforzarse por encontrar formas creativas de hacer posible el "sí". Cuando los profesionales dicen que sí, utilizan el lenguaje del compromiso para que no haya dudas sobre lo que han prometido.

4. Codificación



En un libro¹ anterior escribí mucho sobre la estructura y la naturaleza del *Código Limpio*. Este capítulo trata del *acto* de codificar y del contexto que rodea ese acto.

A los 18 años podía escribir razonablemente bien, pero tenía que mirar las teclas. No podía escribir a ciegas. Así que una tarde me pasé unas largas horas ante un teclado IBM 029 negándome a mirar los dedos mientras escribía un programa que había escrito en varios formularios de codificación. Examiné cada tarjeta después de teclearla y descarté las que estaban mal escritas.

Al principio escribí bastantes errores. Al final de la noche los escribí todos casi a la perfección. Me di cuenta, durante esa larga noche, de que escribir aciegas es cuestión de *confianza*. Mis dedos sabían dónde estaban las teclas, sólo tenía que ganar la confianza de que no me estaba equivocando. Una de las cosas que me ayudó a tener esa confianza es que podía *sentir* cuando cometía un error. Al final de la tarde, si cometía un error, lo sabía casi al instante y simplemente expulsaba la tarjeta sin mirarla.

Ser capaz de percibir tus errores es realmente importante. No sólo en la mecanografía, sino en todo. Tener sentido del error significa que se cierra rápidamente el bucle de retroalimentación y se aprende de los errores con mayor rapidez. He estudiado,

y dominado, varias disciplinas desde aquel día en el 029. He descubierto que, en cada caso, la clave de la maestría es la confianza y el sentido del error.

Este capítulo describe mi conjunto personal de reglas y principios para la codificación. Estas reglas y principios no se refieren a mi código en sí, sino a mi comportamiento, mi estado de ánimo y mi actitud al escribir código. Describen mi propio contexto mental, moral y emocional para

escribir código. Son las raíces de mi confianza y mi sentido del error.

Es probable que no estés de acuerdo con todo lo que digo aquí. Al fin y al cabo, se trata de algo muy personal. De hecho, es posible que no estés de acuerdo con algunas de mis actitudes y principios. No pasa nada: no pretenden ser verdades absolutas para nadie más que para mí. Lo que son es el enfoque de un hombre para ser un codificador profesional.

Tal vez, estudiando y contemplando mi propio medio de codificación personal, pueda aprender a arrebatarme la piedra de la mano.

Preparación

La codificación es una actividad intelectualmente desafiante y agotadora. Requiere un nivel de concentración y atención que pocas otras disciplinas exigen. Esto se debe a que la codificación requiere hacer malabares con muchos factores que compiten entre sí a la vez.

1. En primer lugar, su código debe funcionar. Debes entender qué problema estás resolviendo y comprender cómo se resuelve ese problema. Debe asegurarse de que el código que escribe es una representación fiel de esa solución. Debes gestionar cada detalle de esa solución sin dejar de ser coherente con el lenguaje, la plataforma, la arquitectura actual y todas las vergüenzas del sistema actual.
2. Su código debe resolver el problema que le ha planteado el cliente. A menudo, los requisitos del cliente no resuelven realmente sus problemas. Es usted quien debe ver esto y negociar con el cliente para asegurarse de que se satisfacen sus verdaderas necesidades.
3. Su código debe encajar bien en el sistema existente. No debe aumentar la rigidez, la fragilidad o la opacidad de ese sistema. Las dependencias deben estar bien gestionadas. En resumen, tu código debe seguir unos sólidos principios de ingeniería.²
4. Tu código debe ser legible para otros programadores. No se trata simplemente de escribir buenos comentarios. Más bien, requiere que elabores el código de tal manera que revele tu intención. Esto es difícil de hacer. De hecho, puede ser lo más difícil que un programador puede dominar.

Compaginar todas estas preocupaciones es difícil. Es fisiológicamente difícil mantener la concentración y el enfoque necesarios durante largos periodos de tiempo. A esto hay que añadir los problemas y las distracciones del trabajo en equipo, en una organización, y las preocupaciones de la vida cotidiana. La conclusión es que las posibilidades de distracción son elevadas.

Si no puedes concentrarte y enfocar lo suficiente, el código que escribas será

erróneo. Tendrá errores. Tendrá una estructura incorrecta. Será opaco y enrevesado. No resolverá los problemas reales de los clientes. En resumen, habrá que rehacerlo o rehacerlo. Trabajar con distracción genera residuos.

Si estás cansado o distraído, *no codifiques*. Sólo acabarás rehaciendo lo que has hecho. En lugar de eso, busca una forma de eliminar las distracciones y tranquilizar tu mente.

Código 3 AM

El peor código que he escrito fue a las 3 de la mañana. Era el año 1988 y trabajaba en una empresa de telecomunicaciones llamada Clear Communications. Todos trabajábamos muchas horas para construir un "sweat equity". Por supuesto, todos soñábamos con ser ricos.

Una noche muy tarde, o mejor dicho, una mañana muy temprano, para resolver un problema de tiempo, hice que mi código se enviara un mensaje a sí mismo a través del sistema de envío de eventos (lo llamábamos "enviar correo"). Era una solución *errónea*, pero a las 3 de la mañana parecía bastante buena. De hecho, después de 18 horas de codificación sólida (por no mencionar las 60-70 horas semanales) era *lo único* que se me ocurría.

Recuerdo que me sentía muy bien conmigo mismo por las largas horas que trabajaba. Recuerdo que me sentía *dedicado*. Recuerdo haber pensado que trabajar a las 3 de la mañana es lo que hacen los profesionales serios. ¡Qué equivocado estaba!

Ese código se volvió contra nosotros una y otra vez. Creó una estructura de diseño defectuosa que todo el mundo utilizaba, pero con la que había que trabajar constantemente. Provocaba todo tipo de errores de sincronización y extraños bucles de retroalimentación. Nos encontramos con

en bucles de correo infinitos, ya que un mensaje provocaba el envío de otro, y luego otro, infinitamente. Nunca tuvimos tiempo de reescribir este fajo (eso creíamos), pero siempre parecíamos tener tiempo de añadir otra verruga o parche para solucionarlo. La basura crecía y crecía, rodeando ese código de las 3 de la mañana con cada vez más equipaje y efectos secundarios. Años después se había convertido en una broma del equipo. Siempre que estaba cansado o frustrado me decían: "¡Cuidado! Bob está a punto de enviarse el correo a sí mismo".

La moraleja de esta historia es: No escribas código cuando estés cansado. La dedicación y la profesionalidad tienen que ver más con la disciplina que con las horas. Asegúrate de que tu sueño, tu salud y tu estilo de vida están ajustados para que puedas dedicar ocho *buenas* horas al día.

Código de la preocupación

¿Alguna vez te has metido en una gran pelea con tu cónyuge o amigo y luego has intentado codificarla? ¿Te has dado cuenta de que había un

proceso de fondo en tu mente tratando de resolver, o al menos revisar, la pelea? A veces puedes sentir el estrés de ese proceso de fondo en tu pecho, o en la boca del estómago. Puede hacer que te sientas ansioso, como cuando has tomado demasiado café o coca-cola light. Es una distracción.

Cuando estoy preocupado por una discusión con mi mujer, o por la crisis de un cliente, o por un hijo enfermo, no puedo mantener la concentración. Mi concentración flaquea. Me encuentro con los ojos en la pantalla y los dedos en el teclado, sin hacer nada. Catatónico.

Paralizado. A un millón de kilómetros de distancia trabajando en el problema en segundo plano en lugar de resolver realmente el problema de codificación que tengo delante.

A veces me obligo a *pensar* en el código. Puede que me obligue a escribir una o dos líneas. Puede que me obligue a pasar una o dos pruebas. Pero no puedo mantenerlo. Inevitablemente me encuentro descendiendo a una insensibilidad estupefacta, sin ver nada a través de mis ojos abiertos, revolviendo interiormente la preocupación de fondo.

He aprendido que no es momento de codificar. Cualquier código que produzca será basura. Así que en lugar de codificar, tengo que resolver la preocupación.

Por supuesto, hay muchas preocupaciones que simplemente no pueden resolverse en una o dos horas. Además, es probable que nuestros empleadores no toleren durante mucho tiempo nuestra incapacidad para trabajar mientras resolvemos nuestros problemas personales. El truco está en aprender a cerrar

reducir el proceso en segundo plano, o al menos reducir su prioridad para que no sea una distracción continua.

Para ello, divido mi tiempo. En lugar de obligarme a codificar mientras la preocupación de fondo me atormenta, dedico un bloque de tiempo, quizás una hora, a trabajar en el asunto que me genera la preocupación. Si mi hijo está enfermo, llamaré a casa para comprobarlo. Si he tenido una discusión con mi mujer, la llamaré y hablaré de los problemas. Si tengo problemas de dinero, dedicaré tiempo a pensar en cómo puedo resolver los problemas financieros. Sé que no es probable que resuelva los problemas en esta hora, pero es muy probable que pueda reducir la ansiedad y calmar el proceso de fondo.

Lo ideal sería que el tiempo dedicado a luchar contra los problemas personales fuera tiempo personal. Sería una pena pasar una hora en la oficina de esta manera.

Los desarrolladores profesionales asignan su tiempo personal para que el tiempo que pasan en la oficina sea lo más productivo posible. Eso significa que debes reservar específicamente tiempo en casa para calmar

tus ansiedades para no llevarlas a la oficina.

Por otro lado, si te encuentras en la oficina y las ansiedades de fondo están minando tu productividad, entonces es mejor dedicar una hora a acallarlas que utilizar la fuerza bruta para escribir un código que luego tendrás que desechar (o peor, vivir con él).

La zona de flujo

Se ha escrito mucho sobre el estado hiperproductivo conocido como "flujo". Algunos programadores lo llaman "la Zona". Se llame como se llame, probablemente estés familiarizado con él. Es el estado de conciencia altamente concentrado y con visión de túnel en el que pueden entrar los programadores mientras escriben código. En este estado se sienten *productivos*. En este estado se sienten *infalibles*. Y por eso desean alcanzar ese estado, y a menudo miden su autoestima por el tiempo que pueden pasar allí.

He aquí un pequeño consejo de alguien que ha estado allí y ha vuelto: *Evita la Zona*. Este estado de conciencia no es realmente hiperproductivo y ciertamente no es infalible. En realidad, no es más que un estado meditativo leve en el que ciertas facultades racionales disminuyen en favor de una sensación de velocidad.

Permítanme ser claro al respecto. Escribirás más código en la Zona. Si estás practicando TDD, darás más vueltas al bucle rojo/verde/refactor.

rápidamente. Y *sentirás* una leve euforia o una sensación de conquista. El problema es que pierdes parte del panorama general mientras estás en la Zona, por lo que es probable que tomes decisiones que luego tendrás que volver atrás y revertir. Puede que el código escrito en la Zona salga más rápido, pero tendrás que volver a visitarlo más veces.

Hoy en día, cuando siento que caigo en la Zona, me alejo unos minutos. Me despejo contestando algunos correos electrónicos o mirando algunos tweets. Si se acerca el mediodía, hago una pausa para comer. Si estoy trabajando en un equipo, busco un compañero.

Una de las grandes ventajas de la programación por parejas es que es prácticamente imposible que una pareja entre en la Zona. La Zona es un estado poco comunicativo, mientras que el emparejamiento requiere una comunicación intensa y constante. De hecho, una de las quejas que escucho a menudo sobre el emparejamiento es que bloquea la entrada en la Zona. ¡Bien! La Zona *no* es donde quieres estar.

Bueno, eso no es *del todo* cierto. Hay momentos en los que la Zona está exactamente donde quieres estar. Cuando estás *practicando*. Pero de eso hablaremos en otro capítulo.

Música

En Teradyne, a finales de los años 70, tenía una oficina privada. Era el administrador del sistema de nuestro PDP 11/60, y por lo tanto era uno de los pocos programadores a los que se les permitía tener un terminal privado. Ese terminal era un VT100 que funcionaba a 9600 baudios y estaba conectado al PDP 11 con 80 pies de cable RS232 que había tendido sobre las baldosas del techo desde mi oficina hasta la sala de ordenadores.

Tenía un equipo de música en mi oficina. Era un viejo tocadiscos, un amplificador y altavoces de suelo. Tenía una importante colección de vinilos, entre los que se encontraban Led Zeppelin, Pink Floyd y... bueno, ya te haces una idea.

Solía poner a tope el equipo de música y luego escribir código. Creía que me ayudaba a concentrarme. Pero estaba equivocado.

Un día volví a un módulo que había estado editando mientras escuchaba la secuencia inicial de *The Wall*. Los comentarios de ese código contenían letras de la pieza, y anotaciones editoriales sobre bombarderos en picado y bebés llorando.

Fue entonces cuando me di cuenta. Como lector del código, estaba aprendiendo más sobre la colección de música del autor (yo) que sobre el problema que el código intentaba resolver.

Me he dado cuenta de que simplemente no codifiqué bien mientras escucho música. La música no me ayuda a concentrarme. De hecho, el acto de escuchar música parece consumir algún recurso vital que mi mente necesita para escribir un código limpio y bien diseñado.

Tal vez no funcione así para ti. Quizá la música te *ayude a* escribir código. Conozco a mucha gente que codifica mientras lleva auriculares. Acepto que la música puede ayudarles, pero también sospecho que lo que realmente ocurre es que la música les ayuda a entrar en la Zona.

Interrupciones

Visualízate a ti mismo mientras codificas en tu puesto de trabajo. ¿Cómo respondes cuando alguien te hace una pregunta? ¿Les hablas con brusquedad? ¿Miras de reojo? ¿Tu lenguaje corporal les dice que se vayan porque estás ocupado? En resumen, ¿eres grosero?

¿O dejas lo que estás haciendo y ayudas amablemente a alguien que está atascado? ¿Lo tratas como te gustaría que te trataran a ti si estuvieras atascado?

La respuesta grosera suele provenir de la Zona. Puede que te moleste que te saquen de la Zona, o que te moleste que alguien interfiera en tu intento de entrar en la Zona. En cualquier caso, la grosería suele provenir de tu relación con la Zona.

Sin embargo, a veces no es la Zona la que tiene la culpa, sino que estás

tratando de entender algo complicado que requiere concentración. Hay varias soluciones para esto.

El emparejamiento puede ser muy útil para hacer frente a las interrupciones. Tu compañero de pareja puede mantener el contexto del problema en cuestión, mientras tú atiendes una llamada telefónica o una pregunta de un compañero de trabajo. Cuando vuelves a tu compañero de pareja, éste te ayuda rápidamente a reconstruir el contexto mental que tenías antes de la interrupción.

TDD es otra gran ayuda. Si tienes una prueba que falla, esa prueba mantiene el contexto en el que te encuentras. Puedes volver a ella después de una interrupción y seguir haciendo que esa prueba fallida pase.

Al final, por supuesto, *habrá interrupciones* que te distraigan y te hagan perder tiempo. Cuando ocurran, recuerde que la próxima vez puede ser el que tiene que interrumpir a otro. Así que la actitud profesional es unavoluntad educada de ser útil.

Bloqueo del escritor

A veces el código simplemente no viene. Me ha pasado a mí y he visto que le pasa a otros. Te sientas en tu puesto de trabajo y no pasa nada.

A menudo encontrarás otro trabajo que hacer. Leerás el correo electrónico. Leerás tweets. Mirarás libros, agendas o documentos. Convocarás reuniones. Iniciarás conversaciones con otros. Harás *cualquier cosa* para notener que enfrentarte a esa estación de trabajo y ver cómo el código se niegaa aparecer.

¿Qué causa estos bloqueos? Ya hemos hablado de muchos de los factores. Para mí, otro factor importante es el sueño. Si no duermo lo suficiente, simplemente no puedo codificar. Otros son la preocupación, el miedo y la depresión.

Curiosamente hay una solución muy sencilla. Funciona casi siempre. Es fácil de hacer, y puede proporcionarle el impulso para conseguir escribir mucho código.

La solución: Encontrar una pareja.

Es sorprendente lo bien que funciona. En cuanto te sientas junto a otra persona, los problemas que te bloqueaban se desvanecen. Hay un cambio *fisiológico que se produce* cuando trabajas con alguien. No sé lo que es, pero definitivamente puedo sentirlo. Hay algún tipo de cambio químico en mi cerebro o en mi cuerpo que me hace superar el bloqueo y me pone en marcha de nuevo.

No es una solución perfecta. A veces, el cambio dura una o dos horas, y a continuación se produce un agotamiento tan intenso que tengo que separarme de mi pareja y buscar algún hueco en el que recuperarme. A

veces, incluso cuando me siento con alguien, no puedo hacer más que estar de acuerdo con lo que hace esa persona. Pero para mí la reacción típica al emparejamiento es una recuperación de mi impulso.

Aportación creativa

Hay otras cosas que hago para evitar el bloqueo. Hace mucho tiempo aprendí que la producción creativa depende de la entrada creativa.

Leo mucho, y leo todo tipo de material. Leo material sobre software, política, biología, astronomía, física, química, matemáticas y muchomás.

más. Sin embargo, creo que lo que mejor prepara la bomba de laproducción creativa es la ciencia ficción.

Para ti, puede ser otra cosa. Quizás una buena novela de misterio, o poesía, o incluso una novela romántica. Creo que la verdadera cuestión es que la creatividad engendra creatividad. También hay un elemento de escapismo. Las horas que paso lejos de mis problemas habituales, mientras me estimulan activamente con ideas desafiantes y creativas, dan como resultado una presión casi irresistible para crear algo por mí mismo.

No todas las formas de aportación creativa me funcionan. Ver la televisión no suele ayudarme a crear. Ir al cine es mejor, pero sólo un poco. Escuchar música no me ayuda a crear código, pero sí a crear presentaciones, charlas y vídeos. De todas las formas de entrada creativa, nada me funciona mejor que la vieja ópera espacial.

Depuración

Una de las peores sesiones de depuración de mi carrera ocurrió en 1972. Los terminales conectados al sistema de contabilidad de los Teamsters solían congelarse una o dos veces al día. No había forma de forzar que esto sucediera. El error no prefería ningún terminal ni ninguna aplicación en particular. No importaba lo que el usuario hubiera estado haciendo antes de la congelación. Un minuto el terminal funcionaba bien y al siguiente se congelaba sin remedio.

Se tardó semanas en diagnosticar el problema. Mientras tanto, los "Teamsters" estaban cada vez más molestos. Cada vez que se producía una congelación, la persona de ese terminal tenía que dejar de trabajar y esperar hasta que pudieran coordinar a todos los demás usuarios para que terminaran sus tareas. Entonces nos llamaban y reiniciábamos. Era una pesadilla.

Las primeras dos semanas las dedicamos a recopilar datos entrevistando a las personas que habían sufrido los encierros. Les preguntábamos qué estaban haciendo en ese momento y qué habían hecho previamente.

Preguntamos a otros usuarios si habían notado algo en *sus* terminales en el momento del bloqueo. Todas estas entrevistas se hicieron por teléfono

porque los terminales estaban situados en el centro de Chicago, mientras que nosotros trabajábamos a 50 kilómetros al norte, en los campos de maíz.

No teníamos registros, ni contadores, ni depuradores. Nuestro único acceso al interior del sistema eran las luces y los interruptores de palanca del panel frontal. Podíamos parar el ordenador, y luego echar un vistazo a la memoria palabra por palabra. Pero

no pudimos hacer esto por más de cinco minutos porque los Teamsters necesitaban su sistema de respaldo.

Pasamos unos días escribiendo un sencillo inspector en tiempo real que podía ser operado desde el teletipo ASR-33 que nos servía de consola. Con él podíamos mirar y hurgar en la memoria mientras el sistema estaba en marcha. Añadimos mensajes de registro que se imprimían en el teletipo en momentos críticos. Creamos contadores en memoria que contaban los eventos y recordaban el historial de estados que podíamos inspeccionar con el inspector. Y, por supuesto, todo esto tuvo que escribirse desde cero en ensamblador y probarse por las noches cuando el sistema no estaba en uso.

Los terminales eran controlados por interrupciones. Los caracteres que se enviaban a los terminales se mantenían en búferes circulares. Cada vez que un puerto serie terminaba de enviar un carácter, se producía una interrupción y el siguiente carácter del buffer circular se preparaba para ser enviado.

Finalmente descubrimos que cuando un terminal se congelaba era porque las tres variables que gestionaban el buffer circular estaban desincronizadas. No teníamos ni idea de por qué ocurría esto, pero al menos era una pista. En algún lugar de los 5 KSLOC de código de supervisión había un error que manejaba mal uno de esos punteros.

Este nuevo conocimiento también nos permitió descongelar los terminales manualmente. Podíamos introducir valores por defecto en esas tres variables utilizando el inspector, y los terminales volvían a funcionar mágicamente. Con el tiempo, escribimos un pequeño truco que buscaba entre todos los contadores para ver si estaban desalineados y los reparaba. Al principio invocábamos ese hack pulsando un interruptor especial de interrupción del usuario en el panel frontal cada vez que los Teamsters llamaban para informar de una congelación. Más tarde, simplemente ejecutamos la utilidad de reparación una vez por segundo.

Un mes más tarde, el problema de la congelación estaba resuelto, en lo que respecta a los camioneros. De vez en cuando, uno de sus terminales se detenía durante medio segundo o algo así, pero a un ritmo básico de 30 caracteres por segundo, nadie parecía darse cuenta.

Pero, ¿por qué se desajustaban los contadores? Tenía diecinueve años y estaba decidido a averiguarlo.

El código de supervisión había sido escrito por Richard, que desde entonces se había ido a la universidad. Ninguno de los demás estaba familiarizado con ese código porque

Richard había sido muy posesivo con él. Ese código era *suyo* y no se nos permitía conocerlo. Pero ahora Richard se había ido, así que saqué el listado de centímetros de grosor y empecé a repasarlo página por página.

Las colas circulares de ese sistema eran simplemente estructuras de datos FIFO, es decir, colas. Los programas de aplicación empujaban los caracteres en un extremo de la cola hasta que ésta se llenaba. Los cabezales de interrupción sacaban los caracteres del otro extremo de la cola cuando la impresora estaba lista para ellos. Cuando la cola estaba vacía, la impresora se detenía. Nuestro error hacía que las aplicaciones pensaran que la cola estaba llena, pero hacía que los cabezales de interrupción pensaran que la cola estaba vacía.

Las cabezas de interrupción se ejecutan en un "hilo" diferente al resto del código. Así que los contadores y las variables que son manipulados tanto por las cabezas de interrupción como por el resto del código deben ser protegidos de la actualización concurrente. En nuestro caso, eso significaba desactivar las interrupciones alrededor de cualquier código que manipulara esas tres variables. Cuando me senté con ese código sabía que estaba buscando algún lugar en el código que tocara las variables pero que no desactivara las interrupciones primero.

Hoy en día, por supuesto, utilizaríamos la plétora de potentes herramientas a nuestra disposición para encontrar todos los lugares en los que el código tocó esas variables. En cuestión de segundos sabríamos cada línea de código que las tocó. En cuestión de minutos sabríamos cuáles son las que no desactivan las interrupciones. Pero esto era 1972, y yo no tenía ninguna herramienta como esa. Lo que tenía eran mis ojos.

Revisé minuciosamente cada página de ese código, buscando las variables. Por desgracia, las variables se utilizaban *en todas partes*. Casi todas las páginas las tocaban de una forma u otra. Muchas de esas referencias no desactivaban las interrupciones porque eran referencias de sólo lectura y, por tanto, inofensivas. El problema era que en ese ensamblador en particular no había una buena manera de saber si una referencia era de sólo lectura sin seguir la lógica del código. Cada vez que una variable era leída, podía ser actualizada y almacenada posteriormente.

Y si eso ocurriera mientras las interrupciones están activadas, las variables podrían corromperse.

Me llevó días de intenso estudio, pero al final lo encontré. Allí, en medio del código, había un lugar donde una de las tres variables se actualizaba mientras las interrupciones estaban activadas.

Hice los cálculos. La vulnerabilidad duraba unos dos microsegundos.

Había una docena de terminales que funcionaban a 30 cps, así que una interrupción cada 3 ms o

Así que. Dado el tamaño del supervisor, y la velocidad de reloj de la CPU, esperaríamos una congelación por esta vulnerabilidad una o dos veces al día. ¡Bingo!

Arreglé el problema, por supuesto, pero nunca tuve el valor de desactivar el hack automático que inspeccionaba y arreglaba los contadores. Hasta el día de hoy no estoy convencido de que no hubiera otro agujero.

Tiempo de depuración

Por alguna razón, los desarrolladores de software no consideran el tiempo de depuración como tiempo de codificación. Piensan en el tiempo de depuración como una llamada de la naturaleza, algo que simplemente *hay que* hacer. Pero el tiempo de depuración es tan caro para la empresa como el tiempo de codificación, y por lo tanto todo lo que podamos hacer para evitarlo o disminuirlo es bueno.

Hoy en día paso mucho menos tiempo depurando que hace diez años. No he medido la diferencia, pero creo que es un factor de diez. Logré esta reducción realmente radical en el tiempo de depuración adoptando la práctica del Desarrollo Dirigido por Pruebas (TDD), de la que hablaremos en otro capítulo.

Tanto si adoptas el TDD como cualquier otra disciplina de igual eficacia,³te corresponde como profesional reducir el tiempo de depuración lo más cerca posible de cero. Está claro que el cero es un objetivo asintótico, pero no por ello deja de ser el objetivo.

A los médicos no les gusta reabrir pacientes para arreglar algo que hicieron mal. A los abogados no les gusta reabrir casos en los que han metido la pata. Un médico o un abogado que hiciera eso con demasiada frecuencia no sería considerado profesional. Del mismo modo, un desarrollador de software que crea muchos errores está actuando de forma poco profesional.

Cómo controlar el ritmo de trabajo

El desarrollo de software es un maratón, no un sprint. No se puede ganar la carrera intentando correr lo más rápido posible desde el principio. Se gana conservando los recursos y moderando el ritmo. Un corredor de maratón cuida su cuerpo antes y *durante* la carrera. Los programadores profesionales conservan su energía y creatividad con el mismo cuidado.

Saber cuándo alejarse

¿No puedes irte a casa hasta que resuelvas este problema? Oh, sí que puedes, ¡y probablemente deberías! La creatividad y la inteligencia son estados mentales fugaces.

Cuando estás cansado, desaparecen. Si luego golpeas tu cerebro no funcional durante hora tras hora de madrugada intentando resolver un problema, simplemente te cansarás más y reducirás la posibilidad de que la ducha, o el coche, te ayuden a resolver el problema.

Cuando estés atascado, cuando estés cansado, desconéctate durante un tiempo. Deja que tu subconsciente creativo se ocupe del problema. Conseguirás hacer más cosas en menos tiempo y con menos esfuerzo si eres cuidadoso con tus recursos. Controla tu ritmo y el de tu equipo. Aprende tus patrones de creatividad y brillantez, y aprovéchalos en lugar de trabajar contra ellos.

Conduciendo a casa

Un lugar en el que he resuelto varios problemas es mi coche cuando vuelvo a casa del trabajo. Conducir requiere muchos recursos mentales no creativos. Hay que dedicar los ojos, las manos y parte de la mente a la tarea; por tanto, hay que desconectar de los problemas del trabajo. Hay algo en la *desconexión* que permite a tu mente buscar soluciones de una manera diferente y más creativa.

La ducha

He resuelto un número desmesurado de problemas en la ducha. Quizá ese chorro de agua a primera hora de la mañana me despierta y me hace repasar todas las soluciones que se le ocurren a mi cerebro mientras duermo.

Cuando estás trabajando en un problema, a veces te acercas tanto a él que no puedes ver todas las opciones. Te pierdes soluciones elegantes porque la parte creativa de tu mente se ve suprimida por la intensidad de tu enfoque. A veces, la mejor manera de resolver un problema es irse a casa, cenar, ver la televisión, acostarse y levantarse a la mañana siguiente y ducharse.

Llegar tarde

Llegarás tarde. Nos pasa a los mejores. Nos pasa a los más aplicados. A veces nos equivocamos y acabamos llegando tarde.

El truco para gestionar los retrasos es la detección temprana y la transparencia. Lo peor es que sigas diciéndole a todo el mundo, hasta el final, que vas a llegar a tiempo, y luego los decepciones. *No* lo hagas. En lugar de ello, mide *periódicamente* tu progreso con respecto a tu objetivo y

establece tres ⁴fechas finales basadas en hechos: el mejor caso, el casonominal y el peor caso.

Sea lo más honesto posible sobre las tres fechas. *No incorpore esperanzas en sus estimaciones.* Presenta las tres cifras a tu equipo y a las partes interesadas.

Actualice estas cifras diariamente.



¿Y si estos números indican que se *puede* perder un plazo? Por ejemplo, digamos que hay una feria en diez días y que necesitamos tener nuestro producto allí. Pero digamos también que tu estimación de tres números para la función en la que estás trabajando es 8/12/20.

No esperes que puedas hacerlo todo en diez días. La esperanza es el asesino de proyectos. La esperanza destruye los calendarios y arruina la reputación. La esperanza te meterá en serios problemas. Si la feria es dentro de diez días, y tu estimación nominal es de 12, no lo vas a conseguir. Asegúrate de que el equipo y las partes interesadas entienden la situación, y no dejes de hacerlo hasta que haya un plan de retirada. No dejes que nadie tenga esperanzas.

Apurando

¿Y si tu jefe te sienta y te pide que intentes cumplir el plazo? ¿Y si tu jefe insiste en que "hagas lo que sea necesario"? *Mantén tus estimaciones.* Tus estimaciones originales son más precisas que cualquier cambio que hagas mientras tu jefe se enfrenta a ti. Dile a tu jefe que ya has considerado las opciones (porque lo has hecho) y que la única forma de mejorar el calendario es reducir el alcance. *No caigas en la tentación de precipitarte.*

Pobre del pobre desarrollador que se doblegue ante la presión y acceda a *intentar* cumplir el plazo. Ese desarrollador empezará a tomar atajos y a trabajar horas extra con la vana esperanza de obrar un milagro. Esta es una receta para el desastre porque te da a ti, a tu equipo y a las partes interesadas falsas esperanzas. Permite que todo el mundo evite enfrentarse al problema y retrasa las necesarias decisiones difíciles.

No hay manera de apresurarse. No puedes obligarte a codificar más rápido. No puedes hacer que resuelvas los problemas más rápido. Si lo intentas, sólo te retrasarás a ti mismo y harás un lío que retrasará a todos los demás también.

Así que debe responder a su jefe, a su equipo y a sus interlocutores privándoles de esperanza.

Horas extras

Así que tu jefe dice: "¿Y si trabajas dos horas más al día? ¿Y si trabajas el sábado? Vamos, tiene que haber una forma de meter las horas suficientes para terminar el reportaje a tiempo".

Las horas extras pueden funcionar, y a veces son necesarias. A veces se

puede llegar a una fecha que de otro modo sería imposible si se hacen algunas jornadas de diez horas y uno o dos sábados. Pero esto es muy arriesgado. No es probable que consigas hacer un 20% más de trabajo trabajando un 20% más de horas. Es más, las horas extras fracasarán con *toda seguridad* si se prolongan más de dos o tres semanas.

Por lo tanto, *no debes* aceptar trabajar horas extras a menos que (1) te lo puedas permitir personalmente, (2) sean de corta duración, dos semanas o menos, y (3) *tu jefe tenga un plan de respaldo* en caso de que el esfuerzo de las horas extras fracase.

Este último criterio es el que rompe el trato. Si tu jefe no puede explicarte lo que va a hacer si el esfuerzo de las horas extraordinarias fracasa, entonces no deberías aceptar trabajar horas extraordinarias.

Falso envío

De todos los comportamientos poco profesionales que puede tener un programador, quizá el peor de todos sea decir que se ha terminado cuando se sabe que no es así. A veces se trata simplemente de una mentira abierta, y eso ya es bastante malo. Pero el caso mucho más insidioso es cuando conseguimos racionalizar una nueva definición de "terminado". Nos convencemos de que ya hemos terminado *lo suficiente* y pasamos a la siguiente tarea. Racionalizamos que cualquier trabajo que quede puede ser tratado más tarde, cuando tengamos más tiempo.

Esta es una práctica contagiosa. Si un programador lo hace, otros lo verán y seguirán su ejemplo. Uno de ellos estirará aún más la definición de "hecho", y todos los demás adoptarán la nueva definición. He visto esto llevado a extremos horribles. Uno de mis clientes definió "hecho" como "facturado". El código ni siquiera tenía que compilar. Es muy fácil estar "hecho" si nada tiene que funcionar.

Cuando un equipo cae en esta trampa, los directivos escuchan que todo va bien. Todos los informes de situación muestran que todo el mundo está a tiempo. Es como los ciegos que hacen un picnic en las vías del tren: Nadie ve el tren de carga de trabajo inacabado que se les viene encima hasta que es demasiado tarde.

Definir "Hecho"

Se evita el problema de la falsa entrega creando una definición independiente de "hecho". La mejor manera de hacerlo es hacer que sus analistas de negocio y probadores creen pruebas de aceptación automatizadas ⁵que deben pasar antes de poder decir que se ha terminado. Estas pruebas deben estar escritas en un lenguaje de pruebas como FITNESSE, Selenium, RobotFX, Cucumber, etc. Las pruebas deben ser comprensibles para las partes interesadas y la gente de negocios, y deben ejecutarse con frecuencia.

Ayuda

Programar es *difícil*. Cuanto más joven eres, menos crees esto. Después de todo, es sólo un montón de sentencias `if` y `while`. Pero a medida que adquieres experiencia empiezas a darte cuenta de que la forma en que combinas esas sentencias `if` y `while` es críticamente importante. No puedes simplemente juntarlas y esperar lo mejor. En lugar de eso, tienes que dividir cuidadosamente el sistema en pequeñas unidades comprensibles que tengan tan poco que ver entre sí como sea posible, y eso es difícil.

La programación es tan difícil, de hecho, que está más allá de la capacidad de una sola persona para hacerlo bien. No importa lo experto que seas, seguro que te beneficiarás de las ideas y pensamientos de otro programador.

Ayudar a los demás

Por ello, es responsabilidad de los programadores estar disponibles para ayudar a los demás. Es una violación de la ética profesional encerrarse en un cubículo u oficina y rechazar las consultas de los demás. Tu trabajo no es tan importante como para que no puedas prestar algo de tu tiempo para ayudar a los demás. De hecho, como profesional estás obligado a ofrecer esa ayuda siempre que se necesite.

Esto no significa que no necesites tiempo para estar solo. Por supuesto que sí. Pero tienes que ser justo y educado al respecto. Por ejemplo, puedes hacer saber que entre las 10 de la mañana y el mediodía no debes ser molestado, pero que de la 1 a las 3 de la tarde tu puerta está abierta.

Debes ser consciente del estado de tus compañeros de equipo. Si ves a alguien que parece estar en problemas, debes ofrecer tu ayuda. Probablemente te sorprenderá el profundo efecto que puede tener tu ayuda. No es que seas mucho más inteligente que la otra persona, sino que una nueva perspectiva puede ser un profundo catalizador para resolver problemas.

Cuando ayudes a alguien, sentaos a escribir código juntos. Planea pasar la mayor parte de una hora o más. Puede que te lleve menos que eso, pero no quieres que parezca que tienes prisa. Resígnate a la tarea y haz un esfuerzo sólido. Es probable que salgas de allí habiendo aprendido más de lo que has dado.

Ser ayudado

Cuando alguien se ofrezca a ayudarte, sé amable. Acepta la ayuda con gratitud y entrégate a ella. *No protejas tu territorio*. No rechaces la ayuda porque estés bajo presión. Dale treinta minutos más o menos. Si para entonces la persona no está ayudando mucho, discúlpate amablemente y termina la sesión dando las gracias. Recuerda que, al igual que estás obligado a ofrecer ayuda, estás obligado a aceptarla.

Aprende a *pedir* ayuda. Cuando estés atascado, o desconcertado, o simplemente no puedas entender un problema, pide ayuda a alguien. Si estás sentado en una sala de equipo, puedes sentarte y decir: "Necesito ayuda". De lo contrario, utiliza yammer, o twitter, o el correo electrónico, o el teléfono de tu mesa. Pide ayuda. De nuevo, esto es una cuestión de ética profesional. No es profesional quedarse atascado cuando la ayuda es fácilmente accesible.

A estas alturas, es posible que esperen que estalle en un coro de *Kumbaya* mientras los conejitos peludos saltan sobre las espaldas de los unicornios y todos volamos felizmente sobre el arco iris de la esperanza y el cambio. No, no es así. Verás, los programadores *tienden* a ser introvertidos arrogantes y ensimismados. No nos metimos en este negocio porque nos guste *la gente*. La mayoría de nosotros nos metimos en la programación porque preferimos centrarnos profundamente en minucias estériles, hacer malabarismos con muchos conceptos simultáneamente y, en general, demostrarnos a nosotros mismos que tenemos cerebros del tamaño de un planeta, todo ello sin tener que interactuar con las desordenadas complejidades de *otras personas*.

Sí, es un estereotipo. Sí, es una generalización con muchas excepciones. Pero la realidad es que los programadores no suelen ser colaboradores.⁶ Y, sin embargo, la colaboración es fundamental para una programación eficaz. Por lo tanto, como para muchos de nosotros la colaboración no es un instinto, necesitamos *disciplinas* que nos impulsen a colaborar.

Mentoring

Más adelante en el libro hay un capítulo entero sobre este tema. Por ahora permítanme decir simplemente que la formación de los programadores menos experimentados es la

responsabilidad de los que tienen más experiencia. Los cursos de formación no sirven. Los libros no sirven. Nada puede llevar a un joven desarrollador de software a un alto rendimiento más rápido que su propio impulso y la tutoría efectiva de sus superiores. Por lo tanto, una vez más, es una cuestión de ética profesional que los programadores veteranos dediquen tiempo a tomar a los programadores más jóvenes bajo su tutela ya orientarlos. Del mismo modo, los programadores más jóvenes tienen el deber profesional de buscar esa tutoría de sus superiores.

Bibliografía

[**Martin09**]: Robert C. Martin, *Clean Code*, Upper Saddle River, NJ: Prentice Hall, 2009.

[**Martin03**]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: PrenticeHall, 2003.

5. Desarrollo basado en pruebas



Han pasado más de diez años desde que el Desarrollo Dirigido por Pruebas (TDD) hizo su debut en la industria. Llegó como parte de la ola de Extreme Programming (XP), pero desde entonces ha sido adoptado por Scrum, y prácticamente por todos los demás métodos ágiles. Incluso los equipos no ágiles practican TDD.

Cuando, en 1998, oí hablar por primera vez de "Test First Programming", me mostré escéptico. ¿Quién no lo sería? ¿Escribir *primero* las pruebas unitarias? ¿Quién haría una tontería como esa?

Pero para entonces ya llevaba treinta años como programador profesional y había visto cómo iban y venían las cosas en el sector. Sabía que no debía descartar nada de buenas a primeras, sobre todo cuando lo dice alguien como Kent Beck.

Así que en 1999 viajé a Medford, Oregón, para reunirme con Kent y aprender de él la disciplina. La experiencia fue impactante.

Kent y yo nos sentamos en su despacho y empezamos a codificar un pequeño y sencillo problema en Java. Yo quería escribir la tontería sin más. Pero Kent se resistió y me llevó, paso a paso, a través del proceso. Primero escribió una pequeña parte de una prueba unitaria, apenas lo suficiente para calificarla como código. Luego escribió el código suficiente para que esa prueba compilara. Luego escribió un poco más de prueba, y luego más código.

El tiempo de ciclo estaba completamente fuera de mi experiencia. Estaba acostumbrado a escribir código durante la mayor parte de una hora antes de intentar compilarlo o ejecutarlo. Pero Kent ejecutaba literalmente su código cada treinta segundos, más o menos. Me quedé atónito.

Es más, ¡reconocí el tiempo de ciclo! Era el tipo de tiempo de ciclo que había utilizado años atrás cuando era un niño que ¹programaba juegos en

lenguajes interpretados como Basic o Logo. En esos lenguajes no hay tiempo de construcción, así que sólo tienes que añadir una línea de código y luego ejecutar. Se recorre el ciclo muy rápidamente. Y por eso, puedes ser *muy* productivo en esos lenguajes.

Pero en la programación *real* ese tipo de tiempo de ciclo era absurdo. En la programación *real* había que dedicar mucho tiempo a escribir el código, y luego mucho más tiempo a conseguir que se compilara. Y luego aún más tiempo depurándolo. *Yo era un programador de C++, ¡maldita sea!* Y en C++ teníamos tiempos de compilación y enlace que tardaban minutos, a veces horas. Los tiempos de ciclo de treinta segundos eran inimaginables.

Sin embargo, allí estaba Kent, cocinando este programa Java en ciclos de treinta segundos y sin ningún indicio de que fuera a bajar el ritmo pronto. Así que me di cuenta, mientras estaba sentado en la oficina de Kent, de que usando esta sencilla disciplina podía codificar en lenguajes reales con el tiempo de ciclo de Logo. Estaba enganchado.

El jurado está presente

Desde aquellos días he aprendido que TDD es mucho más que un simple truco para acortar mi tiempo de ciclo. La disciplina tiene todo un repertorio de beneficios que describiré en los siguientes párrafos.

Pero primero tengo que decir esto:

- El jurado está presente.
- Se acabó la polémica.
- `GOTO` es perjudicial.
- Y el TDD funciona.

Sí, se han escrito muchos blogs y artículos polémicos sobre TDD a lo largo de los años y todavía los hay. En los primeros tiempos eran intentos serios de crítica y comprensión. Hoy en día, sin embargo, sólo son

despotrica. La conclusión es que el TDD funciona, y todo el mundo tiene que superarlo.

Sé que esto suena estridente y unilateral, pero teniendo en cuenta los antecedentes no creo que los cirujanos deban defender el lavado de manos, y no creo que los programadores deban defender el TDD.

¿Cómo puedes considerarte un profesional si no *sabes* que todo tu código funciona? ¿Cómo puedes saber que todo tu código funciona si no lo pruebas cada vez que haces un cambio? ¿Cómo puedes probarlo cada vez que haces un cambio si no tienes pruebas unitarias automatizadas con una cobertura muy alta? ¿Cómo puedes conseguir pruebas unitarias automatizadas con una cobertura muy alta sin practicar TDD?

Esta última frase requiere cierta elaboración. ¿Qué es el TDD?

Las tres leyes del TDD

1. No se permite escribir ningún código de producción hasta que no se haya escrito primero una prueba unitaria que falle.
2. No está permitido escribir más de una prueba de unidad de lo que es suficiente para fallar
-y no compilar está fallando.
3. No se permite escribir más código de producción que sea suficiente para pasar la prueba de unidad que actualmente falla.

Estas tres leyes te encierran en un ciclo que dura, quizás, treinta segundos. Empiezas escribiendo una pequeña parte de una prueba unitaria. Pero a los pocos segundos debes mencionar el nombre de alguna clase o función que aún no has escrito, lo que hace que la prueba unitaria no compile. Así que debes escribir código de producción que haga que la prueba compile. Pero no puedes escribir más que eso, así que empiezas a escribir más código de prueba unitaria.

Dando vueltas y más vueltas al ciclo. Añadiendo un poco al código de prueba. Añadiendo un poco al código de producción. Los dos flujos de código crecen simultáneamente en componentes complementarios. Las pruebas se ajustan al código de producción como un anticuerpo se ajusta a un antígeno.

La letanía de los beneficios

Certeza

Si adoptas TDD como disciplina profesional, entonces escribirás docenas de pruebas cada día, cientos de pruebas cada semana y miles de pruebas cada año. Y mantendrás todas esas pruebas a mano y las ejecutarás cada vez que hagas algún cambio en el código.

Soy el principal autor y mantenedor de FITNESSE, ²una herramienta de pruebas de aceptación basada en Java. En el momento de escribir este artículo, FITNESSE tiene 64.000 líneas de código, de las cuales 28.000 están contenidas en algo más de 2.200 pruebas unitarias individuales. Estas pruebas cubren al menos el 90% del código de producción ³y tardan unos 90 segundos en ejecutarse.

Cada vez que hago un cambio en cualquier parte de FITNESSE, simplemente ejecuto las pruebas unitarias. Si pasan, estoy casi seguro de que el cambio que hice no rompió nada. ¿Cómo de seguro es "casi seguro"? Lo suficientemente seguro como para enviarlo.

El proceso de control de calidad para FITNESSE es el comando: `ant release`. Ese comando construye FITNESSE desde cero y luego ejecuta todas las pruebas unitarias y de aceptación. Si todas esas pruebas pasan, lo envío.

Tasa de inyección de defectos

Ahora bien, FITNESSE no es una aplicación de misión crítica. Si hay un error, nadie muere, y nadie pierde millones de dólares. Así que puedo permitirme el lujo de realizar envíos basados únicamente en la superación de pruebas. Por otra parte, FITNESSE tiene miles de usuarios, y a pesar de la adición de 20.000 nuevas líneas de código el año pasado, mi lista de errores sólo tiene 17 errores (muchos de los cuales son de naturaleza cosmética). Así que sé que mi tasa de inyección de defectos es muy baja.

No se trata de un efecto aislado. Hay varios informes ⁴y estudios⁵ que describen una reducción significativa de los defectos. Desde IBM hasta Microsoft, desde Sabre hasta Symantec, empresa tras empresa y equipo tras equipo han experimentado reducciones de defectos de 2X, 5X e incluso 10X. Son cifras que ningún profesional debería ignorar.

Valor

¿Por qué no arreglas el código malo cuando lo ves? Tu primera reacción al ver una función desordenada es "Esto es un desastre, hay que limpiarlo". Tu segunda reacción es "¡No lo voy a tocar!" ¿Por qué? Porque sabes que si lo tocas te arriesgas a romperlo; y si lo rompes, pasaa ser tuyo.

Pero, ¿y si pudiera estar *seguro* de que su limpieza no rompe nada? ¿Y situviera el tipo de certeza que acabo de mencionar? ¿Qué pasaría si usted

¿podría pulsar un botón y *saber* en 90 segundos que sus cambios nohan roto nada, y que *sólo han hecho el bien*?

Este es uno de los beneficios más poderosos de TDD. Cuando tienes un conjunto de pruebas en el que confías, entonces pierdes todo el miedo a hacer cambios. Cuando ves un código malo, simplemente lo limpias en el acto. El código se convierte en arcilla que puedes esculpir con seguridad en estructuras simples y agradables.

Cuando los programadores pierden el miedo a limpiar, ¡limpian! Y el código limpio es más fácil de entender, más fácil de cambiar y más fácil de ampliar. Los defectos son aún menos probables porque el código se simplifica. Y la base de código *mejora* constantemente en lugar de la putrefacción normal a la que se ha acostumbrado nuestra industria.

¿Qué programador profesional permitiría que la podredumbre continuara?

Documentación

¿Has utilizado alguna vez un framework de terceros? A menudo, el tercero le enviará un manual con un buen formato escrito por escritores técnicos. El típico manual emplea 27 fotografías brillantes de ocho por diez colores con círculos y flechas y un párrafo en el reverso de cada una de ellas explicando cómo configurar, desplegar, manipular y utilizar ese framework. Al final, en el apéndice, suele haber una pequeña y fea sección que contiene todos los

ejemplos de código.

¿Cuál es el primer lugar al que vas en ese manual? Si eres un programador, vas a los ejemplos de código. Vas al código porque sabes que el código te dirá la verdad. Las 27 fotografías brillantes de ocho por diez colores con círculos y flechas y un párrafo en la parte de atrás pueden ser bonitas, pero si quieres saber cómo usar el código necesitas leer el código.

Cada una de las pruebas unitarias que escribes cuando sigues las tres leyes es un ejemplo, escrito en código, que describe cómo debe usarse el sistema. Si sigues las tres leyes, habrá una prueba unitaria que describa cómo crear cada objeto del sistema, de todas las maneras en que esos objetos pueden ser creados. Habrá una prueba de unidad que describa cómo llamar a cada función del sistema de todas las maneras en que esas funciones pueden ser llamadas de manera significativa. Para cualquier cosa que necesites saber cómo hacer, habrá una prueba de unidad que lo describa en detalle.

Las pruebas unitarias son documentos. Describen el diseño de más bajo nivel del sistema. No son ambiguas, son precisas, están escritas en un

lenguaje que el público entiende y son tan formales que se ejecutan. Son las mejores

tipo de documentación de bajo nivel que puede existir. ¿Qué profesional no proporcionaría esa documentación?

Diseño

Cuando sigues las tres leyes y escribes tus pruebas primero, te enfrentas a un dilema. A menudo sabes exactamente qué código quieres escribir, pero las tres leyes te dicen que escribas una prueba unitaria que falla porque ese código no existe. Esto significa que tienes que probar el código que vas a escribir.

El problema de probar el código es que hay que aislarlo. A menudo es difícil probar una función si esa función llama a otras funciones. Para escribir esa prueba tienes que encontrar alguna manera de desacoplar la función de todas las demás. En otras palabras, la necesidad de probar primero te obliga a pensar en *un buen diseño*.

Si no escribes tus pruebas primero, no hay ninguna fuerza que te impida acoplar las funciones juntas en una masa no comprobable. Si escribes tus pruebas después, podrás probar las entradas y las salidas de la masa total, pero probablemente será bastante difícil probar las funciones individuales.

Por lo tanto, seguir las tres leyes, y escribir tus pruebas primero, crea una fuerza que te impulsa a un mejor diseño desacoplado. ¿Qué profesional no emplearía herramientas que le condujeran hacia mejores diseños?

"Pero puedo escribir mis pruebas después", dices. No, no puedes. En realidad no. Puedes escribir *algunas* pruebas más tarde. Incluso puedes acercarte a una alta cobertura más tarde si tienes cuidado de medirla. Pero

las pruebas que escribes después son la *defensa*.

Las pruebas que se escriben primero son *ofensivas*. Las pruebas a posteriori las escribe alguien que ya está metido en el código y ya sabe cómo se resolvió el problema. No hay manera de que esas pruebas puedan ser ni de lejos tan incisivas como las pruebas escritas primero.

La opción profesional

El resultado de todo esto es que TDD es la opción profesional. Es una disciplina que mejora la seguridad, el valor, la reducción de defectos, la documentación y el diseño. Con todo eso a favor, podría considerarse *poco profesional* no utilizarlo.

Lo que no es el TDD

A pesar de todas sus bondades, el TDD no es una religión ni una fórmulamágica. Seguir las tres leyes no garantiza ninguno de estos beneficios.

Puedes seguir escribiendo mal código incluso si escribes tus pruebas primero. De hecho, puedes escribir malas pruebas.

Del mismo modo, hay ocasiones en las que seguir las tres leyes es simplemente poco práctico o inapropiado. Estas situaciones son raras, pero existen. Ningún desarrollador profesional debería seguir una disciplina cuando ésta hace más daño que bien.

Bibliografía

[**Maximilien**]: E. Michael Maximilien, Laurie Williams, "Assessing Test-Driven Development at IBM",

http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF

[**George2003**]: B. George, y L. Williams, "An Initial Investigation of Test-Driven Development in Industry",

<http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf>

[**Janzen2005**]: D. Janzen y H. Saiedian, "Test-driven development concepts, taxonomy, and future direction" (Conceptos, taxonomía y dirección futura del desarrollo basado en pruebas), *IEEE Computer*, volumen 38, número 9, pp. 43-50.

[**Nagappan2008**]: Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat y Laurie Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams", Springer Science + Business Media, LLC 2008:

http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

6. Practicando



Todos los profesionales practican su arte realizando ejercicios de perfeccionamiento. Los músicos ensayan escalas. Los jugadores de fútbol corren a través de los neumáticos. Los médicos practican suturas y técnicas quirúrgicas. Los abogados practican sus argumentos. Los soldados ensayan misiones. Cuando el rendimiento es importante, los profesionales practican. Este capítulo trata de las formas en que los programadores pueden practicar su arte.

Algunos antecedentes de la práctica

Practicar no es un concepto nuevo en el desarrollo de software, pero no lo reconocimos como práctica hasta justo después del cambio de milenio. Quizás la primera instancia formal de un programa de práctica fue impresa en la página 6 de [\[K&R-C\]](#).

```
main()
{
    printf("hola, mundo\n");
}
```

¿Quién de nosotros no ha escrito ese programa de una forma u otra? Lo utilizamos como una forma de probar un nuevo entorno o un nuevo lenguaje. Escribir y ejecutar ese programa es la prueba de que podemos escribir y ejecutar *cualquier* programa.

Cuando era mucho más joven, uno de los primeros programas que escribía en un ordenador nuevo era `SQINT`, los cuadrados de los enteros. Lo escribí en ensamblador, BASIC, FORTRAN, COBOL y un trillón de otros lenguajes. De nuevo, era una forma de demostrar que podía hacer que el ordenador hiciera lo que yo quería.

A principios de los 80, los ordenadores personales empezaron a aparecer en los grandes almacenes. Cada vez que pasaba por delante de uno, como un VIC-20 o un Commodore-64, o un TRS-80, escribía un pequeño programa

que imprimía en la pantalla un flujo infinito de caracteres '\N y '/. Los patrones que este programa producía eran agradables a la vista y parecían mucho más complejos que el pequeño programa que los generaba.

Aunque estos pequeños programas eran ciertamente programas de práctica, los programadores en general no *practicaban*. Francamente, nunca se nos ocurrió esa idea. Estábamos demasiado ocupados escribiendo código como para pensar en practicar nuestras habilidades. Y además, ¿qué sentido tendría? En aquellos años, la programación no requería reacciones rápidas ni dedos ágiles. No utilizamos editores de pantalla hasta finales de los años 70. Pasábamos gran parte de nuestro tiempo esperando a que se compilara o depurando largos y horribles tramos de código. Todavía no habíamos inventado los ciclos cortos de TDD, así que no necesitábamos el ajuste que la práctica podía aportar.

Veintidós ceros

Pero las cosas han cambiado desde los primeros tiempos de la programación. Algunas cosas han cambiado *mucho*. Otras cosas no han cambiado mucho.

Una de las primeras máquinas para las que escribí programas fue una PDP-8/I. Esta máquina tenía un tiempo de ciclo de 1,5 microsegundos. Tenía 4.096 palabras de 12 bits en la memoria del núcleo. Tenía el tamaño de una nevera y consumía una cantidad significativa de energía eléctrica. Tenía una unidad de disco que podía almacenar 32K de palabras de 12 bits, y hablábamos con ella con un teletipo de 10 caracteres por segundo. Nos parecía una máquina *poderosa* y la utilizábamos para hacer milagros.

Acabo de comprar un nuevo portátil Macbook Pro. Tiene un procesador de doble núcleo a 2,8 GHz, 8 GB de RAM, una unidad SSD de 512 GB y una pantalla LED de 17 pulgadas y 1920 × 1200. Lo llevo en mi mochila. Se sienta en mi regazo. Consume menos de 85 vatios.

Mi portátil es ocho mil veces más rápido, tiene dos millones de veces más de memoria, tiene dieciséis millones de veces más de almacenamiento fuera de línea, requiere el 1% de la

potencia, ocupa el 1% del espacio y cuesta una vigésima parte del precio del PDP-8/I. Hagamos las cuentas:

$$8,000 \times 2,000,000 \times 16,000,000 \times 100 \times 100 \times 25 = 6.4 \times 10^{22}$$

Este número es *grande*. ¡Estamos hablando de 22 *órdenes de magnitud*! Ese es el número de angstroms que hay entre aquí y Alpha Centauri. Ese es el número de electrones que hay en un dólar de plata. Esa es la masa de la Tierra en unidades de Michael Moore. Este es un gran, gran, número. Y está sentado en mi regazo, ¡y probablemente en el tuyo también!

¿Y qué estoy haciendo con este aumento de potencia de 22 factores de diez? Estoy haciendo más o menos lo que hacía con el PDP-8/I. Estoy

escribiendo sentencias *if*, bucles *while* y *asignaciones*.

Oh, tengo mejores herramientas para escribir esas declaraciones. Y tengo mejores lenguajes para escribir esas declaraciones. Pero la naturaleza de las declaraciones no ha cambiado en todo ese tiempo. El código de 2010 sería reconocible para un programador de la década de 1960. La arcilla que manipulamos no ha cambiado mucho en esas cuatro décadas.

Tiempo de entrega

Pero la *forma* de trabajar ha cambiado drásticamente. En los años 60 podía esperar uno o dos días para ver los resultados de una compilación. A finales de los 70, un programa de 50.000 líneas podía tardar 45 minutos en compilarse. Incluso en los 90, los largos tiempos de compilación eran la norma.

Los programadores de hoy no esperan a que se compile.¹ Los programadores de hoy tienen un poder tan inmenso bajo sus dedos que pueden dar vueltas al bucle rojo-verde-refactor en segundos.

Por ejemplo, trabajo en un proyecto Java de 64.000 líneas llamado FITNESSE. Una compilación completa, incluyendo *todas las* pruebas unitarias y de integración, se ejecuta en menos de 4 minutos. Si esas pruebas se superan, estoy listo para enviar el producto. *Así que todo el proceso de control de calidad, desde el código fuente hasta el despliegue, requiere menos de 4 minutos.*

Las compilaciones no llevan casi ningún tiempo medible. Las pruebas parciales requieren *segundos*. Así que, literalmente, puedo dar vueltas al bucle de compilación/prueba *diez veces por minuto*.

No siempre es prudente ir tan rápido. A menudo es mejor ir más despacio y limitarse a *pensar*.² Pero hay otras veces en las que dar vueltas a ese bucle lo más rápido posible es *muy* productivo.

Hacer cualquier cosa rápidamente requiere práctica. Hacer girar el bucle de código/prueba rápidamente requiere tomar decisiones muy rápidas.

Tomar decisiones rápidamente significa ser capaz de reconocer un gran número de situaciones y problemas y *saber* simplemente qué hacer para resolverlos.

Piensa en dos artistas marciales en combate. Cada uno debe reconocer lo que intenta el otro y responder adecuadamente en milisegundos. En una situación de combate no puedes permitirte el lujo de congelar el tiempo, estudiar las posiciones y deliberar sobre la respuesta adecuada. En una situación de combate simplemente tienes que *reaccionar*. De hecho, es tu *cuerpo* el que reacciona mientras tu mente trabaja en una estrategia de nivel superior.

Cuando estás dando vueltas al bucle de código/prueba varias veces por minuto, es tu *cuerpo* el que sabe qué teclas pulsar. Una parte primordial de

tu mente reconoce la situación y reacciona en milisegundos con la solución adecuada, mientras tu mente queda libre para concentrarse en el problema de mayor nivel.

Tanto en el caso de las artes marciales como en el de la programación, la velocidad depende de la *práctica*. Y en ambos casos la práctica es similar. Elegimos un repertorio de pares problema/solución y los ejecutamos una y otra vez hasta que los conocemos en frío.

Piensa en un guitarrista como Carlos Santana. La música que tiene en la cabeza simplemente sale de sus dedos. No se centra en la posición de los dedos ni en la técnica de picado. Su mente es libre para planificar melodías y armonías de alto nivel mientras su cuerpo traduce esos planes en movimientos de dedos de bajo nivel.

Pero para conseguir esa facilidad de ejecución es necesario *practicar*. Los músicos practican las escalas, los estudios y los riffs una y otra vez hasta que se los saben de memoria.

El Dojo de Codificación

Desde 2001 realizo una demostración de TDD que llamo *El juego de los bolos*.³ Es un pequeño y encantador ejercicio que dura unos treinta minutos. Experimenta el conflicto en el diseño, construye un clímax, y termina con una sorpresa. Escribí un capítulo entero sobre este ejemplo en [\[PPP2003\]](#).

A lo largo de los años he realizado esta demostración cientos, quizás miles, de veces. Se me dio *muy* bien. Podía hacerla mientras dormía. Minimizaba las pulsaciones, afinaba los nombres de las variables y retocaba la estructura del algoritmo hasta que quedaba bien. Aunque no lo sabía en ese momento, ésta fue mi primera kata.

En 2005 asistí a la Conferencia XP2005 en Sheffield, Inglaterra. Asistí a una sesión con el nombre de *Coding Dojo* dirigida por Laurent Bossavit y Emmanuel Gaillot. Hicieron que todo el mundo abriera sus portátiles y codificara con ellos mientras utilizaban TDD para escribir el *Juego de la Vida* de Conway. Lo llamaron "Kata" y acreditaron a "Pragmatic" Dave Thomas⁴ con la idea original.⁵

Desde entonces, muchos programadores han adoptado la metáfora de las artes marciales para sus sesiones de práctica. El nombre de Coding Dojo⁶ parece haberse quedado.

A veces un grupo de programadores se reúne y practica juntos como los artistas marciales. Otras veces, los programadores practican en solitario, también como los artistas marciales.

Hace aproximadamente un año estaba enseñando a un grupo de desarrolladores en Omaha. Durante el almuerzo me invitaron a unirme a su Coding Dojo. Observé cómo veinte desarrolladores abrían sus portátiles y,

tecla a tecla, seguían al líder que estaba haciendo la Kata del *Juego de Bolos*.

Hay varios tipos de actividades que tienen lugar en un dojo. He aquí algunas:

Kata

En las artes marciales, una *kata* es un conjunto preciso de movimientos coreografiados que simulan una parte de un combate. El objetivo, al que se aproxima asintóticamente, es la perfección. El artista se esfuerza por enseñar a su cuerpo a realizar cada movimiento a la perfección y a ensamblar esos movimientos en una representación fluida. Los katas bien ejecutados son hermosos de ver.

Por muy bonitos que sean, el objetivo de aprender un kata no es representarlo en el escenario. El objetivo es entrenar la mente y el cuerpo para reaccionar en una situación de combate concreta. El objetivo es hacer que los movimientos perfeccionados sean automáticos e instintivos para que estén ahí cuando los necesites.

Una kata de programación es un conjunto preciso de pulsaciones de teclas y movimientos de ratón coreografiados que simulan la resolución de algún problema de programación. En realidad no estás resolviendo el problema porque ya conoces la solución. Más bien, estás practicando los movimientos y las decisiones que implica la resolución del problema.

La asíntota de la perfección es de nuevo el objetivo. Se repite el ejercicio una y otra vez para entrenar al cerebro y a los dedos a moverse y reaccionar.

A medida que vayas practicando, podrás descubrir sutiles mejoras y eficiencias en tus movimientos o en la propia solución.

Practicar un conjunto de katas es una buena manera de aprender las teclas de acceso rápido y los modismos de navegación. También es una buena manera de aprender disciplinas como TDD y CI. Pero lo más importante es que es una buena manera de introducir en tu subconsciente pares de problemas/soluciones comunes, para que simplemente sepas cómo resolverlos cuando te enfrentes a ellos en la programación real.

Como cualquier artista marcial, un programador debe conocer varios katas diferentes y practicarlos con regularidad para que no se desvanezcan de la memoria. Muchos katas están grabados en <http://katas.softwarecraftsmanship.org>. Otros se pueden encontrar en <http://codekata.pragprog.com>. Algunas de mis favoritas son:

- ***El juego de los bolos***: <http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>
- ***Factores primos***: <http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsK>

[ata](#)

- **Word Wrap:** <http://thecleancoder.blogspot.com/2010/10/craftsman-62-dark-path.html>

Para un verdadero reto, intenta aprender un kata tan bien que puedas ponerle música. Hacerlo bien es *difícil*. ⁷

Wasa

Cuando estudié jujitsu, gran parte de nuestro tiempo en el dojo lo pasábamos en parejas practicando nuestro *wasu*. El *wasu* se parece mucho a un kata de dos personas. Las rutinas se memorizan con precisión y se reproducen. Un compañero hace el papel de agresor y el otro de defensor. Los movimientos se repiten una y otra vez mientras los practicantes intercambian sus papeles.

Los programadores pueden practicar de forma similar utilizando un juego conocido como *ping-pong*. ⁸ Los dos compañeros eligen una kata, o un problema sencillo. Un programador escribe una prueba de unidad y el otro debe hacerla pasar. Luego se invierten los papeles.

Si los compañeros eligen una kata estándar, el resultado es conocido y

los programadores están practicando y criticando las técnicas de teclado y ratón de los demás, y lo bien que han memorizado la kata. En el

Por otro lado, si los compañeros eligen un nuevo problema para resolver, el juego puede volverse un poco más interesante. El programador que escribe una prueba tiene una cantidad desmesurada de control sobre cómo se resolverá el problema. También tiene un gran poder para establecer restricciones. Por ejemplo, si los programadores deciden implementar un algoritmo de ordenación, el escritor de la prueba puede poner fácilmente restricciones de velocidad y espacio de memoria que desafiarán a su compañero. Esto puede hacer que el juego sea bastante competitivo... y divertido.

Randori

El Randori es un combate de forma libre. En nuestro dojo de jujitsu, preparábamos una variedad de escenarios de combate y luego los representábamos. A veces se le decía a una persona que se defendiera, mientras cada uno de los demás le atacaba en secuencia.

A veces poníamos a dos o más atacantes contra un solo defensor (normalmente el sensei, que casi siempre ganaba). A veces hacíamos dos contra dos, y así sucesivamente.

El combate simulado no se adapta bien a la programación; sin embargo, hay un juego que se practica en muchos dojos de codificación llamado *randori*. Es muy parecido al *wasu* de dos hombres en el que los compañeros resuelven un problema. Sin embargo, se juega con muchas personas y las reglas tienen

un giro. Con la pantalla proyectada en la pared, una persona escribe una prueba y luego se sienta. La siguiente persona hace pasar la prueba y luego escribe la siguiente. Esto puede hacerse en secuencia alrededor de la mesa, o las personas pueden simplemente alinearse según sesientan movidas. En cualquier caso, estos ejercicios pueden ser *muy* divertidos.

Es sorprendente lo mucho que se puede aprender en estas sesiones. Se puede obtener una visión inmensa de la forma en que otras personas resuelven los problemas. Estos conocimientos sólo pueden servir para ampliar tu propio enfoque y mejorar tus habilidades.

Ampliar su experiencia

Los programadores profesionales suelen sufrir una falta de diversidad en los tipos de problemas que resuelven. Los empleadores suelen imponer un único lenguaje, plataforma y dominio en el que sus programadores deben trabajar. Sin una influencia amplia, esto puede llevar a un estrechamiento

muy poco saludable de su currículum y su mentalidad. No es raro que estos programadores no estén preparados para los cambios que periódicamente seproducen en el sector.

Código abierto

Una forma de mantenerse a la vanguardia es hacer lo que hacen los abogados y los médicos: aceptar algún trabajo gratuito contribuyendo a un proyecto de código abierto. Hay muchos, y probablemente no hay mejor manera de aumentar tu repertorio de habilidades que trabajar en algo que le interesa a otra persona.

Así que si eres un programador de Java, contribuye a un proyecto de Rails. Si escribes mucho C++ para tu empresa, busca un proyecto de Python y contribuye a él.

Ética en la práctica

Los programadores profesionales practican en su tiempo libre. No es el trabajo de su empleador ayudarle a mantener sus habilidades afinadas para usted. No es tarea de su empleador ayudarle a mantener afinado su currículum. Los pacientes no pagan a los médicos para que practiquen la sutura. Los aficionados al fútbol no pagan (normalmente) por ver a los jugadores correr entre los neumáticos. Los asistentes a los conciertos no pagan por escuchar a los músicos tocar escalas.

Y los empleadores de los programadores no tienen que pagarles por su tiempo de práctica.

Como el tiempo de práctica es tu propio tiempo, no tienes que usar los mismos idiomas o plataformas que usas con tu empleador. Escoge cualquier lenguaje que te guste y mantén tus habilidades de políglota a punto. Si trabajas en un taller de .NET, practica un poco de Java o Ruby

durante el almuerzo o en casa.

Conclusión:

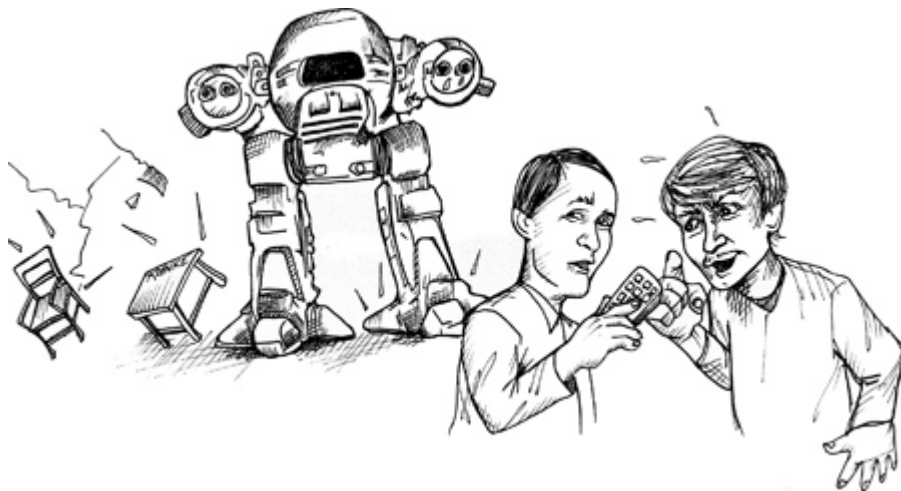
De un modo u otro, *todos los* profesionales ejercen. Lo hacen porque se preocupan por hacer el mejor trabajo posible. Además, practican en su tiempo libre porque son conscientes de que es su responsabilidad -y no la de su empleador- mantener sus habilidades a punto. Practicar es lo que haces cuando *no* te pagan. Lo haces para *que te paguen*, y para que te paguen *bien*.

Bibliografía

[K&R-C]: Brian W. Kernighan y Dennis M. Ritchie, *The C Programming Language*, Upper Saddle River, NJ: Prentice Hall, 1975.

[PPP2003]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

7. Pruebas de aceptación



El papel del programador profesional es un papel de comunicación además de un papel de desarrollo. Recuerde que el principio de "entrar y salir de la basura" también se aplica a los programadores, por lo que los programadores profesionales se aseguran de que su comunicación con los demás miembros del equipo, y con la empresa, sea precisa y saludable.

Comunicar los requisitos

Uno de los problemas de comunicación más comunes entre los programadores y la empresa son los requisitos. Los empresarios declaran lo que creen que necesitan, y luego los programadores construyen lo que creen que la empresa ha descrito. Al menos, así es como se supone que funciona. En realidad, la comunicación de los requisitos es extremadamente difícil, y el proceso está plagado de errores.

En 1979, mientras trabajaba en Teradyne, recibí la visita de Tom, el director de instalación y servicio de campo. Me pidió que le enseñara a utilizar el editor de texto ED-402 para crear un sencillo sistema de fichas de problemas.

El ED-402 era un editor propio escrito para el ordenador M365, que era el clon del PDP-8 de Teradyne. Como editor de texto era muy potente. Tenía un lenguaje de scripting incorporado que utilizábamos para todo tipo de aplicaciones de texto sencillas.

Tom no era programador. Pero la aplicación que tenía en mente era sencilla, así que pensó que yo podría enseñarle rápidamente y luego él podría escribir el

aplicación propia. En mi ingenuidad pensé lo mismo. Al fin y al cabo, el lenguaje de scripts era poco más que un lenguaje de macros para los comandos de edición, con construcciones de decisión y bucle muy rudimentarias.

Así que nos sentamos juntos y le pregunté qué quería que hiciera su aplicación. Comenzó con la pantalla de entrada inicial. Le mostré cómo crear un archivo de texto que contuviera las declaraciones del script y cómo escribir la representación simbólica de los comandos de edición en ese script. Pero cuando le miré a los ojos, no había nada que mirar. Mi explicación simplemente no tenía ningún sentido para él.

Era la primera vez que me encontraba con esto. Para mí era algo sencillo representar simbólicamente los comandos del editor. Por ejemplo, para representar un comando control-B (el comando que pone el cursor al principio de la línea actual) simplemente se escribía ^B en el archivo de script. Pero esto no tenía sentido para Tom. No podía dar el salto de editar un archivo a editar un archivo que editaba un archivo.

Tom no era tonto. Creo que simplemente se dio cuenta de que esto iba a ser mucho más complicado de lo que pensó inicialmente, y no quiso invertir el tiempo y la energía mental necesarios para aprender algo tan horriblemente enrevesado como usar un editor para comandar un editor.

Así que, poco a poco, me encontré implementando esta aplicación mientras él se sentaba a observar. En los primeros veinte minutos estaba claro que su énfasis había cambiado de aprender a hacerlo él mismo a asegurarse de que lo que *yo* hacía era lo que *él* quería.

Nos llevó un día entero. Él describía una característica y yo la implementaba mientras él miraba. El tiempo de ciclo era de cinco minutos o menos, así que no había razón para que se levantara a hacer otra cosa. Me pedía que hiciera X, y en cinco minutos tenía X funcionando.

A menudo dibujaba lo que quería en un trozo de papel. Algunas de las cosas que quería eran difíciles de hacer en el ED-402, así que yo proponía

otra cosa. Al final nos poníamos de acuerdo en algo que funcionara y yo lo hacía funcionar.

Pero luego lo probábamos y cambiaba de opinión. Decía algo así como: "Sí, eso no tiene la fluidez que estoy buscando. Vamos a intentarlo de otra manera".

Hora tras hora, hemos jugado con la aplicación y le hemos dado forma. Probamos una cosa, luego otra, y luego otra. Me quedó muy claro que *él* era el escultor y yo la herramienta que manejaba.

Al final, consiguió la aplicación que buscaba, pero no tenía ni idea de cómo construir la siguiente para él. Yo, por mi parte, aprendí una poderosa lección sobre cómo los clientes descubren realmente lo que necesitan.

Aprendí que su visión de las características no suele sobrevivir al contacto real con el ordenador.

Precisión prematura

Tanto las empresas como los programadores tienen la tentación de caer en la trampa de la precisión prematura. Los empresarios quieren saber exactamente lo que van a obtener antes de autorizar un proyecto. Los desarrolladores quieren saber exactamente lo que deben entregar antes de estimar el proyecto. Ambas partes quieren una precisión que sencillamente no se puede alcanzar, y a menudo están dispuestas a gastar una fortuna para conseguirla.

El principio de incertidumbre

El problema es que las cosas parecen diferentes sobre el papel que en un sistema en funcionamiento. Cuando la empresa ve realmente lo que ha especificado funcionando en un sistema, se da cuenta de que no era lo que quería en absoluto. Una vez que ven el requisito realmente funcionando, tienen una mejor idea de lo que realmente quieren, y normalmente no es lo que están viendo.

Hay una especie de efecto observador, o principio de incertidumbre, en juego. Cuando demuestras una función a la empresa, le das más información de la que tenía antes, y esa nueva información influye en cómo ve todo el sistema.

Al final, cuanto más precisos sean los requisitos, menos relevantes serán a medida que se implante el sistema.

Ansiedad por la estimación

Los desarrolladores también pueden caer en la trampa de la precisión. Saben que deben estimar el sistema y a menudo piensan que esto requiere precisión. No es así.

En primer lugar, incluso con información perfecta tus estimaciones tendrán una enorme varianza. En segundo lugar, el principio de incertidumbre hace

que la precisión inicial se convierta en una burla. Los requisitos cambiarán haciendo que esa precisión *sea* discutible.

Los desarrolladores profesionales entienden que las estimaciones pueden, y deben, hacerse sobre la base de requisitos de baja precisión, y reconocen que esas estimaciones *son estimaciones*. Para reforzar esto, los desarrolladores profesionales siempre incluyen barras de error con sus estimaciones para que la empresa entienda la incertidumbre. (Véase [el capítulo 10](#), "Estimación").

Ambigüedad tardía

La solución a la precisión prematura es aplazar la precisión todo lo posible. Los desarrolladores profesionales no concretan un requisito hasta que están a punto de desarrollarlo. Sin embargo, eso puede conducir a otro mal: la ambigüedad tardía.

A menudo las partes interesadas no están de acuerdo. Cuando lo hacen, les resulta más fácil sortear el desacuerdo que resolverlo. Encontrarán alguna forma de redactar el requisito con la que todos estén de acuerdo, sin resolver realmente la disputa. Una vez escuché a Tom DeMarco decir: "Una ambigüedad en un documento de requisitos representa una discusión entre los interesados". ¹

Por supuesto, no hace falta una discusión o un desacuerdo para crear ambigüedad. A veces los interesados simplemente asumen que sus lectores saben lo que quieren decir.

Puede ser perfectamente claro para ellos en su contexto, pero significar algo completamente diferente para el programador que lo lee. Este tipo de ambigüedad contextual también puede darse cuando clientes y programadores hablan cara a cara.

Sam (accionista): "Vale, ahora hay que hacer una copia de seguridad de estos archivos de registro". Paula: "Bien, ¿con qué frecuencia?"

Sam: "Diariamente".

Paula: "Claro. ¿Y dónde quieres que se guarde?"

Sam: "¿Qué quieres decir?"

Paula: "¿Quieres que lo guarde en un subdirectorío concreto?"

Sam: "Sí, eso estaría bien".

Paula: "¿Cómo lo llamamos?"

Sam: "¿Qué tal 'respaldo'?"

Paula: "Claro, eso estaría bien. Así que escribiremos el archivo de registro en el directorio de copias de seguridad todos los días. ¿A qué hora?"

Sam: "Todos los días".

Paula: "No, me refiero a qué hora del día quieres que se escriba". Sam: "A cualquier hora".

Paula: "¿Mediodía?"

Sam: "No, no durante el horario comercial. A medianoche sería mejor". Paula: "De acuerdo, a medianoche entonces".

Sam: "Genial, ¡gracias!"

Paula: "Siempre es un placer".

Más tarde, Paula le cuenta a su compañero Peter sobre la tarea.

Paula: "Bien, tenemos que copiar el archivo de registro en un subdirectorío llamado copia de seguridad cada noche a medianoche".

Peter: "Bien, ¿qué nombre de archivo debemos usar?" Paula: "log.backup debería servir".

Peter: "Lo tienes."

En otra oficina, Sam habla por teléfono con su cliente.

Sam: "Sí, sí, los archivos de registro se guardarán".

Carl: "De acuerdo, es vital que nunca perdamos ningún registro. Tenemos que revisar todos esos archivos de registro, incluso meses o años después, cada vez que haya una interrupción, un evento o una disputa."

Sam: "No te preocupes, acabo de hablar con Paula. Ella guardará los registros en un directorío llamado backup cada noche a medianoche".

Carl: "De acuerdo, eso suena bien".

Supongo que ha detectado la ambigüedad. El cliente espera que se guarden todos los archivos de registro, y Paula simplemente pensó que quería guardar el archivo de registro de la noche anterior. Cuando el cliente vaya a buscar las copias de seguridad de los archivos de registro de meses, sólo encontrará el de anoche.

En este caso, tanto Paula como Sam se equivocaron. Es responsabilidad de los desarrolladores profesionales (y de las partes interesadas) asegurarse de que se elimine toda ambigüedad de los requisitos.

Esto es *difícil*, y sólo hay una forma que conozco de hacerlo.

Pruebas de aceptación

El término *prueba de aceptación* está sobrecargado y se utiliza en exceso. Algunas personas asumen que son las pruebas que los usuarios ejecutan

antes de aceptar un lanzamiento. Otros piensan que son pruebas de control de calidad. En este capítulo definiremos las pruebas de aceptación como pruebas escritas por una colaboración de las partes interesadas y los programadores *con el fin de definir cuando un requisito está hecho*.

La definición de "hecho"

Una de las ambigüedades más comunes a las que nos enfrentamos como profesionales del software es la ambigüedad de "terminado". Cuando un desarrollador dice que ha terminado una tarea, ¿qué significa eso? ¿El desarrollador ha terminado en el sentido de que está listo para desplegar la función con plena confianza? ¿O quiere decir que está listo para el control de calidad?

O tal vez haya terminado de escribirlo y haya conseguido que funcione una vez, pero aún no lo ha probado realmente.

He trabajado con equipos que tenían una definición diferente para las palabras "hecho" y "completo". Un equipo en particular utilizaba los términos "hecho" y "terminado".

Los desarrolladores profesionales tienen una única definición de "hecho": Terminado significa *hecho*. Hecho significa que todo el código está escrito, todas las pruebas pasan, el control de calidad y las partes interesadas han aceptado. Hecho.

Pero, ¿cómo se puede conseguir este nivel de realización y seguir avanzando rápidamente de iteración en iteración? Creas un conjunto de pruebas automatizadas que, cuando pasan, cumplen todos los criterios anteriores. Cuando las pruebas de aceptación de su característica pasan, usted ha *terminado*.

Los desarrolladores profesionales dirigen la definición de sus requisitos hasta las pruebas de aceptación automatizadas. Trabajan con las partes interesadas y el departamento de control de calidad para garantizar que estas pruebas automatizadas sean una especificación completa de lo que se ha hecho.

Sam: "Bien, ahora hay que hacer una copia de seguridad de estos archivos de registro".

Paula: "Bien, ¿con qué frecuencia?"

Sam: "Diariamente".

Paula: "Claro. ¿Y dónde quieres que se guarde?"

Sam: "¿Qué quieres decir?"

Paula: "¿Quieres que lo guarde en un subdirectorío concreto?"

Sam: "Sí, eso estaría bien".

Paula: "¿Cómo lo llamamos?"

Sam: "¿Qué tal "respaldo"?"

Tom (probador): "Espera, copia de seguridad es un nombre demasiado común. ¿Qué estás almacenando realmente en este directorio?"

Sam: "Las copias de seguridad". Tom: "¿Las copias de seguridad de qué?" Sam: "Los archivos de registro".

Paula: "Pero sólo hay un archivo de registro".

Sam: "No, hay muchos. Uno para cada día".

Tom: "¿Quieres decir que hay un archivo de registro *activo* y muchas copias de seguridad de archivos de registro?"

Sam: "Por supuesto".

Paula: "¡Oh! Pensé que sólo querías una copia de seguridad temporal". Sam: "No, el cliente quiere conservarlos todos para siempre". Paula: "Eso es nuevo para mí. Vale, me alegro de que lo hayamos aclarado".

Tom: "Así que el nombre del subdirectorío debería decirnos exactamente qué hay en él".

Sam: "Tiene todos los viejos registros inactivos". Tom: "Así que llamémoslo

`old_inactive_logs`". Sam: "Genial".

Tom: "¿Y cuándo se crea este directorio?" Sam: "¿Eh?"

Paula: "Debemos crear el directorio cuando el sistema se inicie, pero sólo si el directorio no existe ya".

Tom: "Bien, ahí está nuestra primera prueba. Tendré que arrancar el sistema y ver si se crea el directorio `old_inactive_logs`. Entonces añadiré un archivo a ese directorio. Luego apagaré y volveré a arrancar, y me aseguraré de que tanto el directorio como el archivo siguen ahí".

Paula: "Esa prueba te va a llevar mucho tiempo de ejecución. La puesta en marcha del sistema ya dura 20 segundos, y sigue creciendo. Además, no quiero tener que construir todo el sistema cada vez que ejecute las pruebas de aceptación".

Tom: "¿Qué sugieres?"

Paula: "Crearemos una clase `SystemStarter`. El programa principal cargará este iniciador con un grupo de objetos `StartupCommand`, que seguirán

el patrón `COMMAND`. Entonces, durante el arranque del sistema, el `SystemStarter` simplemente le dirá a todos los objetos `StartupCommand` que se ejecuten. Uno de esos `StartupCommand` derivados creará el directorio `old_inactive_logs`, pero sólo si no existe ya".

Tom: "Oh, OK, entonces todo lo que necesito probar es ese derivado de `StartupCommand`. Puedo escribir una simple prueba `FITNESSE` para eso".

Tom va a la junta.

"La primera parte será algo así":

```
dado el comando LogFileDirectoryStartupCommand dado
que el directorio old_inactive_logs no existe
```

```
cuando se ejecuta el comando
entonces el directorio old_inactive_logs debería
existir y estar vacío
```

"La segunda parte se verá así":

```
dado el comando LogFileDirectoryStartupCommand dado
que el directorio old_inactive_logs existe y que
contiene un archivo llamado x
Cuando se ejecuta el comando
Entonces el directorio old_inactive_logs debería seguir
existiendo y debería contener un archivo llamado x
```

Paula: "Sí, eso debería cubrirlo".

Sam: "Vaya, ¿es realmente necesario todo eso?"

Paula: "Sam, ¿cuál de estas dos afirmaciones no es lo suficientemente importante como para especificarla?"

Sam: "Sólo quiero decir que parece mucho trabajo pensar y escribir todas estas pruebas".

Tom: "Lo es, pero no es más trabajo que escribir un plan de pruebas manual. Y es *mucho* más trabajo ejecutar repetidamente una prueba manual".

Comunicación

El objetivo de las pruebas de aceptación es la comunicación, la claridad y la precisión. Al acordarlas, los desarrolladores, las partes interesadas y los probadores entienden cuál es el plan de comportamiento del sistema.

Lograr este tipo de claridad es responsabilidad de todas las partes. Los desarrolladores profesionales hacen que

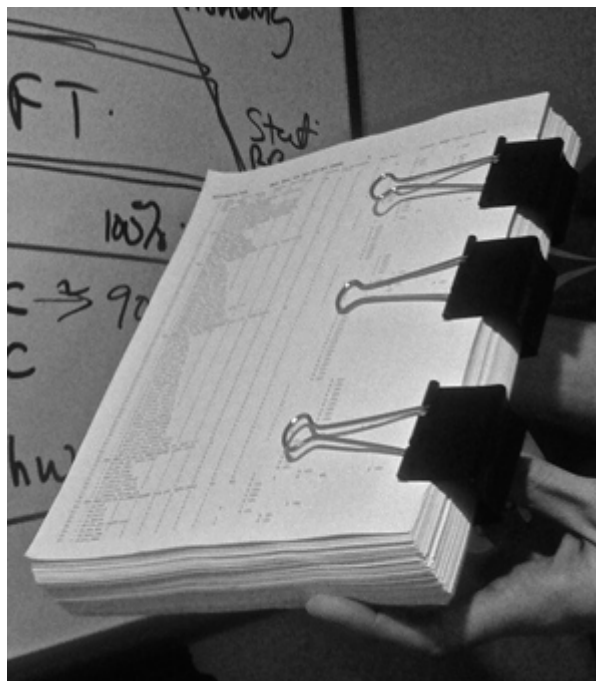
su responsabilidad de trabajar con las partes interesadas y los probadores para garantizar que todas las partes sepan lo que se va a construir.

Automatización

Las pruebas de aceptación deben ser *siempre* automatizadas. Hay un lugar para las pruebas manuales en otras partes del ciclo de vida del software, pero *este tipo de* pruebas nunca deben ser manuales. La razón es sencilla: el coste.

Considere la imagen de la [Figura 7-1](#). Las manos que se ven en ella pertenecen al director de control de calidad de una gran empresa de Internet. El documento que sostiene es el índice *de* su plan de pruebas *manuales*. Tiene un ejército de probadores manuales en ubicaciones externas que ejecutan este plan una vez cada seis semanas. Le cuesta más de un millón de dólares cada vez. Me lo cuenta porque acaba de volver de una reunión en la que su jefe le ha dicho que tienen que recortar su presupuesto en un 50%. Su pregunta es: "¿Qué mitad de estas pruebas no debo hacer?".

Figura 7-1. Plan de pruebas manual



Llamar a esto un desastre sería un gran eufemismo. El coste de ejecutar el plan de pruebas manual es tan enorme que han decidido sacrificarlo y simplemente vivir con el hecho de que *no sabrán si la mitad de su producto funciona*.

Los desarrolladores profesionales no permiten que se produzca este tipo de situación. El coste de la automatización de las pruebas de aceptación es tan pequeño en comparación con el coste de la ejecución de los planes de pruebas manuales que no tiene sentido económico escribir guiones para que los ejecuten los humanos. Los desarrolladores profesionales se responsabilizan de su parte para garantizar la automatización de las pruebas de aceptación.

Hay muchas herramientas comerciales y de código abierto que facilitan la

automatización de las pruebas de aceptación. FITNESSE, Cucumber, cuke4duke, robot framework y Selenium, por mencionar algunas. Todas estas herramientas permiten especificar pruebas automatizadas de una forma que los no programadores pueden leer, entender e incluso ser autores.

Trabajo extra

El punto de Sam sobre el trabajo es comprensible. Parece un montón de trabajo extra para escribir pruebas de aceptación como esta. Pero dada [la Figura7-1](#) podemos ver que no es realmente un trabajo extra en absoluto. Escribir estas pruebas es simplemente el trabajo de especificar el sistema. Especificar a este nivel de detalle es la única manera en que nosotros, como programadores, podemos saber qué significa "hecho". Especificar a este nivel de detalle es la única manera de que las partes interesadas puedan asegurarse de que el sistema por el que están pagando realmente hace lo que necesitan. Y especificar a este nivel de detalle es la única manera de automatizar con éxito las pruebas. Así que no veas estas pruebas como un trabajo extra. Considérelas como un gran ahorro de tiempo y dinero. Estas pruebas evitarán que implemente el sistema equivocado y le permitirán *saber* cuándo ha terminado.

¿Quién y cuándo escribe las pruebas de aceptación?

En un mundo ideal, las partes interesadas y el departamento de control de calidad colaborarían para escribir estas pruebas, y los desarrolladores las revisarían para comprobar su coherencia. En el mundo real, las partes interesadas rara vez tienen el tiempo o la inclinación para profundizar en el nivel de detalle necesario. Así que a menudo delegan la responsabilidad a los analistas de negocio, al departamento de control de calidad o incluso a los desarrolladores. Si resulta que los desarrolladores deben escribir estas pruebas, hay que tener cuidado de que el desarrollador que escribe la

prueba no sea el mismo que el que implementa la característica probada.

Normalmente, los analistas de negocio escriben las versiones del "camino feliz" de las pruebas, porque esas pruebas describen las características que tienen valor comercial. El control de calidad suele escribir las pruebas del "camino infeliz", las condiciones límite, las excepciones y los casos de esquina. Esto se debe a que el trabajo de QA es ayudar a pensar en lo que puede salir mal.

Siguiendo el principio de "precisión tardía", las pruebas de aceptación deben escribirse lo más tarde posible, normalmente unos días antes de que se implemente la característica. En los proyectos ágiles, las pruebas se escriben *después de que* las características hayan sido seleccionadas para la siguiente Iteración o Sprint.

Las primeras pruebas de aceptación deben estar listas el primer día de la

iteración. Cada día deberían completarse más hasta la mitad de la iteración, cuando todas deberían estar listas. Si todas las pruebas de aceptación no están listas a mitad de la iteración, algunos desarrolladores tendrán que colaborar para terminarlas. Si esto ocurre con frecuencia, habrá que añadir más BAs y/o QAs al equipo.

El papel del desarrollador

El trabajo de implementación de una función comienza cuando las pruebas de aceptación de esa función están listas. Los desarrolladores ejecutan las pruebas de aceptación de la nueva función y ven cómo fallan. A continuación, trabajan para conectar la prueba de aceptación con el sistema, y luego empiezan a hacer que la prueba pase implementando la característica deseada.

Paula: "Peter, ¿me echas una mano con esta historia?" Pedro:

"Claro, Paula, ¿qué pasa?"

Paula: "Aquí está la prueba de aceptación. Como puedes ver, está fallando".

```
dado el comando LogFileDirectoryStartupCommand
dado que el directorio old_inactive_logs no existe cuando
se ejecuta el comando
entonces el directorio old_inactive_logs debería
existir y estar vacío
```

Peter: "Sí, todo rojo. Ninguno de los escenarios está escrito. Déjame escribir el primero".

```
|scenario|dado el comando _|cmd|
|crear comando|@cmd|
```

Paula: "¿Tenemos ya una operación `createCommand`?"

Pedro: "Sí, está en el `CommandUtilitiesFixture` que escribí la semana pasada". Paula: "Vale, pues vamos a hacer la prueba ahora".

Pedro: (ejecuta la prueba). "Sí, la primera línea está en verde, pasemos a la siguiente".

No se preocupe demasiado por los escenarios y las instalaciones. Esos son sólo algunos de los elementos que tienes que escribir para conectar las pruebas con el sistema que se está probando.

Basta con decir que todas las herramientas proporcionan alguna manera de utilizar la concordancia de patrones para reconocer y analizar las declaraciones de la prueba, y luego llamar a las funciones que alimentan los datos de la prueba en el sistema que se está probando. El esfuerzo es pequeño, y los escenarios y los accesorios son reutilizables en muchas pruebas diferentes.

El punto de todo esto es que es el trabajo del desarrollador conectar las pruebas de aceptación con el sistema, y luego hacer que esas pruebas

pasen.

Prueba de negociación y agresión pasiva

Los autores de las pruebas son humanos y cometen errores. A veces las pruebas, tal y como están escritas, no tienen mucho sentido una vez que se empiezan a aplicar. Pueden ser demasiado complicadas. Pueden ser incómodas. Puede que contengan suposiciones tontas. O simplemente pueden ser erróneas. Esto puede ser muy frustrante si usted es el desarrollador que tiene que hacer que la prueba pase.

Como desarrollador profesional, tu trabajo es negociar con el autor del test para conseguir un mejor test. Lo que *nunca* debes hacer es tomar la opción pasivo-agresiva y decirte a ti mismo: "Bueno, eso es lo que dice el test, así que eso es lo que voy a hacer".

Recuerda que, como profesional, tu trabajo consiste en ayudar a tu equipo a crear el mejor software posible. Eso significa que todo el mundo tiene que estar atento a los errores y deslices, y trabajar juntos para corregirlos.

Paula: "Tom, esta prueba no está bien".

```
asegúrese de que la operación de posteo  $\epsilon$  termine en 2 segundos.
```

Tom: "A mí me parece bien. Nuestro requisito es que los usuarios no tengan que esperar más de dos segundos. ¿Cuál es el problema?"

Paula: "El problema es que sólo podemos dar esa garantía en sentido estadístico".

Tom: "¿Eh? Eso suena a palabras de comadreo. El requisito es dos segundos".

Paula: "Correcto, y podemos conseguirlo el 99,5% de las veces". Tom: "Paula, ese no es el requisito".

Paula: "Pero es la realidad. No hay manera de que pueda hacer la garantía de otra manera".

Tom: "Sam va a hacer un berrinche".

Paula: "No, la verdad es que ya he hablado con él al respecto. Le parece bien siempre que la experiencia *normal* del usuario sea de dos segundos o menos".

Tom: "Bien, ¿cómo escribo esta prueba? No puedo decir simplemente que la operación de correo *suele* terminar en dos segundos".

Paula: "Lo dices estadísticamente".

Tom: "¿Quieres decir que quieres que haga una operación de mil puestos y que me asegure de que no hay más de cinco que duren más de dos segundos? Eso es absurdo".

Paula: "No, eso llevaría la mayor parte de una hora de trabajo. ¿Qué tal esto?"

```
ejecute 15 transacciones postales y acumule los tiempos.  
asegúrese de que la puntuación Z fo 2 segundos sea al  
menos 2,57
```

Tom: "Vaya, ¿qué es una puntuación Z?"

Paula: "Sólo un poco de estadística. Toma, ¿qué te parece esto?"

```
ejecutar 15 operaciones de contabilización y acumular tiempos.  
aseguran que las probabilidades son del 99,5% de que el tiempo  
sea inferior a 2 segundos.
```

Tom: "Sí, es legible, más o menos, pero ¿puedo confiar en las matemáticas que hay detrás?"

Paula: "Me aseguraré de mostrar todos los cálculos intermedios en el informe de la prueba para que puedas comprobar las matemáticas si tienes alguna duda".

Tom: "Vale, eso me vale".

Pruebas de aceptación y pruebas unitarias

Las pruebas de aceptación no son pruebas *unitarias*. Las pruebas unitarias las *escriben* los programadores *para* los programadores. Son documentos formales de diseño que describen la estructura y el comportamiento de más bajo nivel del código. El público son los programadores, no las empresas.

Las pruebas de aceptación son escritas *por* el negocio *para* el negocio (incluso cuando tú, el desarrollador, acabas escribiéndolas). Son documentos de requisitos formales que especifican cómo debe comportarse el sistema desde el punto de vista de la empresa. La audiencia es la empresa y los programadores.

Puede ser tentador intentar eliminar el "trabajo extra" asumiendo que los dos tipos de pruebas son redundantes. Aunque es cierto que las pruebas unitarias y de aceptación suelen probar las mismas cosas, no son redundantes en absoluto.

En primer lugar, aunque pueden probar las mismas cosas, lo hacen a través de mecanismos y vías diferentes. Las pruebas unitarias se adentran en las tripas del sistema haciendo llamadas a métodos de clases concretas. Las pruebas de aceptación invocan el sistema mucho más allá, a nivel de la API o incluso de la interfaz de usuario. Por lo tanto, los caminos de ejecución que toman estas pruebas son muy diferentes.

Pero la verdadera razón por la que estas pruebas no son redundantes es que su función principal *no es probar*. El hecho de que sean pruebas es incidental. Las pruebas unitarias y las pruebas de aceptación son primero documentos y luego pruebas. Su propósito principal es documentar

formalmente el diseño, la estructura y el comportamiento del sistema. El hecho de que verifiquen automáticamente el diseño, la estructura y el comportamiento que especifican es tremendamente útil, pero la especificación es su verdadero propósito.

GUIs y otras complicaciones

Es difícil especificar las interfaces gráficas por adelantado. Se puede hacer, pero rara vez se hace bien. La razón es que la estética es subjetiva y, por tanto, volátil. La gente quiere *juguetear* con las interfaces gráficas de usuario. Quieren masajearlas y manipularlas. Quieren probar diferentes tipos de letra, colores, diseños de página y flujos de trabajo. Las interfaces gráficas están en constante cambio.

Esto hace que sea un reto escribir pruebas de aceptación para las interfaces gráficas de usuario. El truco está en diseñar el sistema de forma que se pueda tratar la GUI como si fuera una API en lugar de un conjunto de botones, deslizadores, rejillas y menús. Esto puede parecer extraño, pero en realidad se trata de un buen diseño.

Existe un principio de diseño llamado Principio de Responsabilidad Única (PRS). Este principio establece que se deben separar las cosas que cambian por diferentes razones, y agrupar las cosas que cambian por las mismas razones. Las interfaces gráficas no son una excepción.

El diseño, el formato y el flujo de trabajo de la interfaz gráfica cambiarán por razones estéticas y de eficiencia, pero la capacidad subyacente de la interfaz gráfica seguirá siendo la misma a pesar de estos cambios.

Por lo tanto, cuando se escriben pruebas de aceptación para una GUI se aprovechan las abstracciones subyacentes que no cambian con mucha frecuencia.

Por ejemplo, puede haber varios botones en una página. En lugar de crear pruebas que hagan clic en esos botones en función de su posición en la página, tal vez pueda hacer clic en ellos en función de sus nombres. Mejor aún, tal vez cada uno de ellos tenga un `ID único` que pueda utilizar. Es mucho mejor escribir una prueba que seleccione el botón cuyo `ID` es `ok_button` que seleccionar el botón de la columna 3 de la fila 4 de la cuadrícula de control.

Pruebas a través de la interfaz adecuada

Mejor aún es escribir pruebas que invoquen las características del sistema subyacente a través de una API real en lugar de a través de la GUI. Esta API debería ser la misma que utiliza la interfaz gráfica de usuario. Esto no es nada nuevo. Los expertos en diseño llevan décadas diciéndonos que separemos nuestras interfaces gráficas de usuario de nuestras reglas de negocio.

Probar a través de la GUI es siempre problemático, a menos que se esté probando *sólo* la GUI. La razón es que la GUI es probable que cambie,

haciendo que las pruebas sean muy frágiles. Cuando cada cambio de la GUI rompe mil pruebas, o bien vas a empezar a tirar las pruebas o vas a dejar de cambiar la GUI. Ninguna de las dos opciones es buena. Así que escribe tus pruebas de reglas de negocio para que pasen por una API justo debajo de la GUI.

Algunas pruebas de aceptación especifican el comportamiento de la propia interfaz gráfica de usuario. Estas pruebas *deben* pasar por la GUI. Sin embargo, estas pruebas no prueban las reglas de negocio y por lo tanto no requieren que las reglas de negocio estén conectadas a la GUI. Por lo tanto, es una buena idea desacoplar la GUI y las reglas de negocio y reemplazar las reglas de negocio con stubs mientras se prueba la propia GUI.

Mantenga las pruebas de la GUI al mínimo. Son frágiles, porque la GUI es volátil. Cuantas más pruebas de la GUI tenga, menos probabilidades tendrá de mantenerlas.

Integración continua

Asegúrese de que todas sus pruebas unitarias y de aceptación se ejecutan varias veces al día en un sistema de *integración continua*. Este sistema debería ser activado por tu sistema de control de código fuente. Cada vez que alguien envíe un módulo, el sistema de integración continua debería iniciar una compilación y, a continuación, ejecutar todas las pruebas del sistema. Los resultados de esa ejecución deberían enviarse por correo electrónico a todos los miembros del equipo.

Parar las prensas

Es muy importante mantener las pruebas CI en funcionamiento en todo momento. Nunca deben fallar. Si fallan, entonces todo el equipo debe dejar de hacer lo que está haciendo

y centrarse en conseguir que las pruebas rotas vuelvan a pasar. Una compilación rota en el sistema de CI debe ser vista como una emergencia, un evento de "parar las prensas".

He asesorado a equipos que no se tomaron en serio las pruebas rotas. Estaban "demasiado ocupados" para arreglar las pruebas rotas, así que las dejaron de lado, prometiendo arreglarlas más tarde. En un caso, el equipo sacó las pruebas rotas de la compilación porque era muy incómodo verlas fallar. Más tarde, tras la entrega al cliente, se dieron cuenta de que habían olvidado volver a incluir esas pruebas en la compilación. Lo supieron porque un cliente enfadado les llamó para informarles de los fallos.

Conclusión:

La comunicación sobre los detalles es difícil. Esto es especialmente cierto para los programadores y las partes interesadas que se comunican sobre los detalles de una aplicación. Es demasiado fácil que cada parte agite sus

manos y *asuma* que la otra parte lo entiende. Con demasiada frecuencia, ambas partes están de acuerdo en que entienden y se van con ideas completamente diferentes.

La única manera que conozco de eliminar eficazmente los errores de comunicación entre los programadores y las partes interesadas es escribir pruebas de aceptación automatizadas. Estas pruebas son tan formales que se ejecutan. No tienen ninguna ambigüedad y no pueden desviarse de la aplicación. Son el documento de requisitos perfecto.

8. Estrategias de ensayo



Los desarrolladores profesionales prueban su código. Pero las pruebas no consisten simplemente en escribir unas cuantas pruebas unitarias o de aceptación. Escribir estas pruebas es algo bueno, pero está lejos de ser suficiente. Lo que todo equipo de desarrollo profesional necesita es una buena *estrategia de pruebas*.

En 1989, trabajaba en Rational en la primera versión de Rose. Cada mes, más o menos, nuestro director de control de calidad convocaba un día de "caza de errores". Todos los miembros del equipo, desde los programadores hasta los gerentes, pasando por las secretarías y los administradores de bases de datos, se sentaban con Rose y trataban de hacer que fallara. Se otorgaban premios a los distintos tipos de fallos. La persona que encontrara un fallo que se estrellara podía ganar una cena parados. La persona que encontraba el mayor número de fallos podía ganar un fin de semana en Monterrey.

El control de calidad no debe encontrar nada

Lo he dicho antes y lo volveré a decir. A pesar de que su empresa puede tener un grupo de control de calidad separado para probar el software, el

objetivo del grupo de desarrollo debe ser que el control de calidad no encuentre nada malo.

Por supuesto, no es probable que este objetivo se consiga constantemente.

Al fin y al cabo, cuando hay un grupo de personas inteligentes decididas a encontrar todas las arrugas y deficiencias de un producto, es probable que encuentren algunas.

Aun así, cada vez que el departamento de control de calidad encuentra algo, el equipo de desarrollo debería reaccionar con horror. Deberían preguntarse cómo ha ocurrido y tomar medidas para evitarlo en el futuro.

El control de calidad es parte del equipo

La sección anterior podría haber hecho parecer que la GC y el desarrollo están enfrentados, que su relación es adversa. Esta no es la intención. Por el contrario, el control de calidad y el desarrollo deberían trabajar juntos para garantizar la calidad del sistema. El mejor papel para la parte de la GC del equipo es actuar como especificadores y caracterizadores.

La GC como especificadora

El papel de la garantía de calidad debería ser trabajar con la empresa para crear las pruebas de aceptación automatizadas que se convierten en el verdadero documento de especificaciones y requisitos del sistema.

Iteración tras iteración, recogen los requisitos de la empresa y los traducen en pruebas que describen a los desarrolladores cómo debe comportarse el sistema (véase [el capítulo 7](#), "Pruebas de aceptación"). En general, la empresa redacta las pruebas del camino feliz, mientras que el departamento de control de calidad redacta las pruebas de esquina, límite y camino infeliz.

La GC como caracterizadora

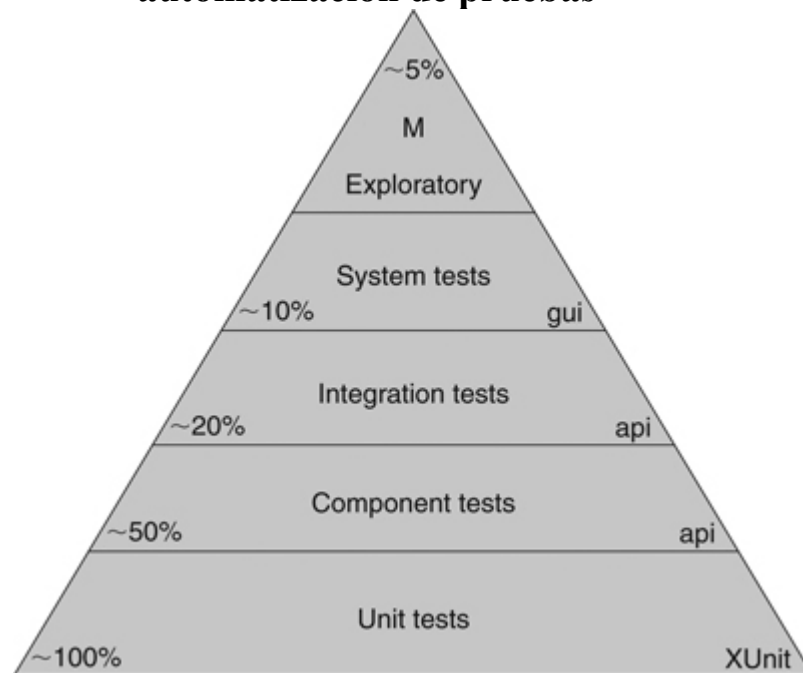
La otra función de la garantía de calidad es utilizar la disciplina de las pruebas exploratorias¹ para caracterizar el verdadero comportamiento del sistema en funcionamiento e informar de dicho comportamiento al desarrollo y a la empresa. En esta función, la GC *no* interpreta los requisitos. Más bien, identifica los comportamientos reales del sistema.

La pirámide de la automatización de pruebas

Los desarrolladores profesionales emplean la disciplina del Desarrollo Dirigido por Pruebas para crear pruebas unitarias. Los equipos de desarrollo profesionales utilizan pruebas de aceptación para especificar su sistema, y la integración continua ([capítulo 7](#), página [110](#)) para evitar la regresión. Pero estas pruebas son sólo una parte de la historia. Por muy bueno que sea tener un conjunto de pruebas unitarias y de aceptación, también necesitamos pruebas de nivel superior para asegurar que el control de calidad no encuentre nada. [La Figura 8-1](#) muestra la Pirámide de Automatización de Pruebas, ²una representación gráfica de los tipos de

pruebas que necesita una organización de desarrollo profesional.

Figura 8-1. La pirámide de automatización de pruebas



Pruebas unitarias

En la base de la pirámide están las pruebas unitarias. Estas pruebas están escritas por programadores, para programadores, en el lenguaje de programación del sistema. La intención de estas pruebas es especificar el sistema al nivel más bajo. Los desarrolladores escriben estas pruebas antes de escribir el código de producción como una forma de especificar lo que van a escribir. Se ejecutan como parte de la integración continua para garantizar que se mantiene la intención de los programadores.

Las pruebas unitarias proporcionan una cobertura tan cercana al 100% como sea posible. Por lo general, este número debería estar en algún punto de los 90. Y debe ser una cobertura *verdadera*, en contraposición a las pruebas falsas que ejecutan el código sin afirmar su comportamiento.

Pruebas de componentes

Son algunas de las pruebas de aceptación mencionadas en el capítulo anterior. Generalmente se escriben contra componentes individuales del sistema.

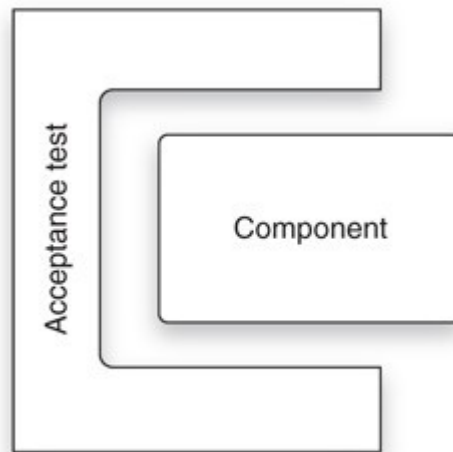
Los componentes del sistema encapsulan las reglas de negocio, por lo que las pruebas para esos componentes son las pruebas de aceptación para esas reglas de negocio

Como se muestra en [la Figura 8-2](#), una prueba de componente envuelve un componente. Pasa los datos de entrada al componente y recoge los

datos de salida del mismo. Comprueba que la salida coincide con la entrada. Cualquier otro componente del sistema se desacopla de la prueba

utilizando técnicas apropiadas de imitación y duplicación de pruebas.

Figura 8-2. Prueba de aceptación de componentes



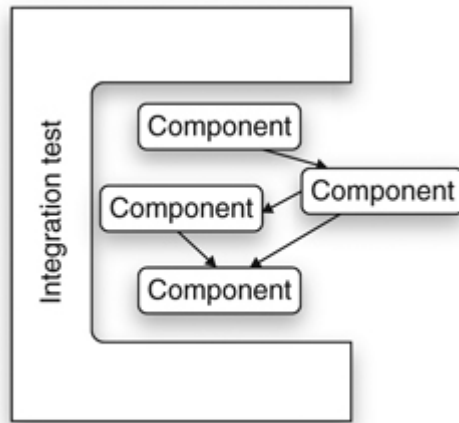
Las pruebas de componentes son escritas por QA y Business con la ayuda de desarrollo. Se componen en un entorno de pruebas de componentes como FITNESSE, JBehave o Cucumber. (Los componentes de la GUI se prueban con entornos de prueba de la GUI como Selenium o Watir). La intención es que el negocio sea capaz de leer e interpretar estas pruebas, si no es el autor de las mismas.

Las pruebas de componentes cubren aproximadamente la mitad del sistema. Se dirigen más a las situaciones de camino feliz y a los casos muy obvios de esquina, límite y camino alternativo. La inmensa mayoría de los casos de camino infeliz están cubiertos por las pruebas unitarias y no tienen sentido a nivel de pruebas de componentes.

Pruebas de integración

Estas pruebas sólo tienen sentido para los sistemas más grandes que tienen muchos componentes. Como se muestra en [la Figura 8-3](#), estas pruebas ensamblan grupos de componentes y prueban lo bien que se comunican entre sí. Los demás componentes del sistema se desacoplan como es habitual con los correspondientes mocks y test-doubles.

Figura 8-3. Prueba de integración



Las pruebas de integración son pruebas de *coreografía*. No prueban las reglas de negocio. Más bien, prueban lo bien que baila el conjunto de componentes. Son pruebas de *fontanería* que garantizan que los componentes están bien conectados y pueden comunicarse claramente entre sí.

Las pruebas de integración suelen ser escritas por los arquitectos del sistema, o los diseñadores principales, del sistema. Las pruebas garantizan que la estructura arquitectónica del sistema es sólida. Es en este nivel donde podemos ver las pruebas de rendimiento y de desempeño.

Las pruebas de integración suelen estar escritas en el mismo lenguaje y entorno que las pruebas de componentes. Normalmente *no se* ejecutan como parte del conjunto de integración continua, porque suelen tener tiempos de ejecución más largos.

En cambio, estas pruebas se ejecutan periódicamente (cada noche, cada semana, etc.) según lo consideren necesario sus autores.

Pruebas del sistema

Se trata de pruebas automatizadas que se ejecutan contra todo el sistema integrado. Son las pruebas de integración por excelencia. No prueban directamente las reglas de negocio. Más bien, comprueban que el sistema se ha conectado correctamente y que sus partes interoperan según lo previsto. Es de esperar que en este conjunto se realicen pruebas de rendimiento y desempeño.

Estas pruebas las escriben los arquitectos de sistemas y los responsables técnicos. Normalmente se escriben en el mismo lenguaje y entorno que las pruebas de integración de la interfaz de usuario. Se ejecutan con relativa poca frecuencia en función de su duración, pero cuanto más frecuente sea, mejor.

Las pruebas del sistema cubren quizás el 10% del sistema. Esto se debe a que su intención no es garantizar el comportamiento correcto del sistema, sino su correcta *construcción*. El comportamiento correcto del código y los componentes subyacentes ya se ha comprobado en las capas inferiores de la pirámide.

Pruebas exploratorias manuales

Aquí es donde los humanos ponen sus manos en los teclados y sus ojos en las pantallas. Estas pruebas no *están* automatizadas *ni tienen un guión*. La intención de estas pruebas es explorar el sistema en busca de comportamientos inesperados, al tiempo que se confirman los comportamientos esperados. Para ello necesitamos cerebros humanos, con creatividad humana, que trabajen para investigar y explorar el sistema. Crear un plan de pruebas por escrito para este tipo de pruebas es un fracaso.

Algunos equipos contarán con especialistas para realizar este trabajo. Otros equipos se limitarán a declarar uno o dos días de "caza de errores" en los que el mayor número posible de personas, incluidos gerentes, secretarios, programadores, probadores y redactores técnicos, "golpearán" el sistema para ver si pueden hacer que se rompa.

El objetivo no es la cobertura. No vamos a probar todas las reglas de negocio ni todas las vías de ejecución con estas pruebas. El objetivo es asegurar que el sistema se comporta bien bajo la operación humana y encontrar creativamente tantas "peculiaridades" como sea posible.

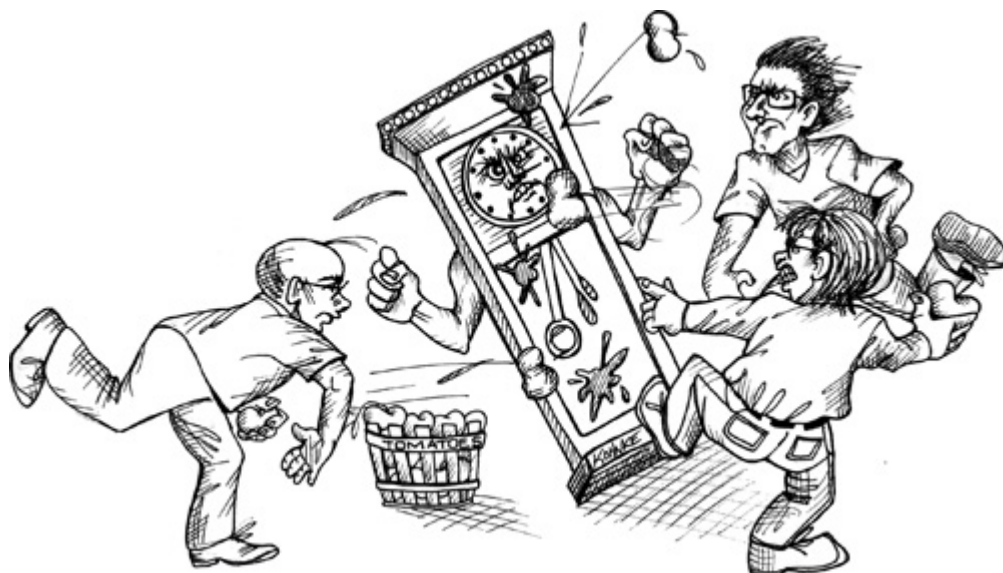
Conclusión:

TDD es una disciplina poderosa, y las pruebas de aceptación son formas valiosas de expresar y hacer cumplir los requisitos. Pero sólo son una parte de una estrategia de pruebas total. Para cumplir el objetivo de que "el control de calidad no encuentre nada", los equipos de desarrollo deben trabajar codo con codo con el control de calidad para crear una jerarquía de pruebas unitarias, de componentes, de integración, de sistema y exploratorias. Estas pruebas deben ejecutarse con la mayor frecuencia posible para proporcionar la máxima información y garantizar que el sistema permanezca continuamente limpio.

Bibliografía

[COHN09]: Mike Cohn, *Succeeding with Agile*, Boston, MA: Addison-Wesley, 2009.

9. Gestión del tiempo



Ocho horas es un periodo de tiempo extraordinariamente corto. Son sólo 480 minutos o 28.800 segundos. Como profesional, esperas utilizar esos pocos y preciosos segundos de la manera más eficiente y eficaz posible. ¿Qué estrategia puedes utilizar para asegurarte de que no pierdes el poco tiempo que tienes? ¿Cómo puedes gestionar eficazmente tu tiempo?

En 1986 vivía en Little Sandhurst, Surrey, Inglaterra. Dirigía un departamento de desarrollo de software de 15 personas para Teradyne en Bracknell. Mis días eran muy ajetreados, con llamadas telefónicas, reuniones improvisadas, problemas de servicio de campo e interrupciones. Así que, para poder hacer cualquier trabajo, tuve que adoptar algunas disciplinas de gestión del tiempo bastante drásticas.

$$2-\frac{1}{2}$$

- Al llegar, escribía un horario en mi pizarra. Dividía el tiempo en incrementos de 15 minutos y rellenaba la actividad en la que trabajaría durante ese bloque de tiempo.
- Llené completamente las primeras 3 horas de ese horario. A partir de las 9 de la mañana, empecé a dejar un hueco de 15 minutos por hora; de esta forma podía
empujar la mayoría de las interrupciones en una de esas ranuras abiertas y seguir trabajando.
- Dejé el tiempo después del almuerzo sin programar porque sabía que para entonces todo el infierno se habría desatado y tendría que estar en modo reactivo durante el resto del día. Durante esos raros periodos de la tarde en los que el caos no se entrometía, simplemente trabajaba en lo más importante hasta que lo hacía.

Este plan no siempre tuvo éxito. Levantarse a las 5 de la mañana no siempre

era factible, y a veces el caos se salía de todas mis cuidadosas estrategias y consumía mi día. Pero en la mayoría de los casos pude mantener la cabeza fuera del agua.

Reuniones

Las reuniones cuestan unos 200 dólares por hora y por asistente. Esto tiene en cuenta los salarios, las prestaciones, los costes de las instalaciones, etc. La próxima vez que esté en una reunión, calcule el coste. Puede que se sorprenda.

Hay dos verdades sobre la reunión.

1. Las reuniones son necesarias.
2. Las reuniones son una gran pérdida de tiempo.

A menudo estas dos verdades describen por igual la misma reunión. Algunos de los asistentes pueden encontrarlas valiosas; otros pueden considerarlas redundantes o inútiles.

Los profesionales son conscientes del alto coste de las reuniones. También son conscientes de que su propio tiempo es precioso; tienen códigos que escribir y horarios que cumplir. Por lo tanto, se resisten activamente a asistir a reuniones que no tengan un beneficio inmediato y significativo.

En declive

No tiene que asistir a todas las reuniones a las que le inviten. De hecho, es poco profesional acudir a demasiadas reuniones. Hay que utilizar el tiempo con prudencia. Por eso hay que tener mucho cuidado con las reuniones a las que se asiste y las que se rechazan educadamente.

La persona que te invita a una reunión no es responsable de gestionar tu tiempo. Sólo *tú* puedes hacerlo. Así que cuando recibas una invitación a una reunión, no

aceptar a menos que se trate de una reunión en la que su participación sea inmediata y significativamente necesaria para el trabajo que está realizando en ese momento.

A veces la reunión versará sobre algo que le interesa, pero que no es inmediatamente necesario. Tendrás que elegir si puedes permitirte ese tiempo. Ten cuidado: puede que haya más que suficientes reuniones de este tipo para consumir tus días.

A veces la reunión versará sobre algo a lo que puedes contribuir pero que no es inmediatamente significativo para lo que estás haciendo actualmente. Tendrás que elegir si la pérdida para tu proyecto vale la pena el beneficio para el suyo. Esto puede sonar cínico, pero tu responsabilidad es primero con *tus* proyectos. Aun así, a menudo es bueno que un equipo ayude a otro,

por lo que es posible que quieras discutir tu participación con tu equipo y tu director.

A veces su presencia en la reunión será solicitada por alguien con autoridad, como un ingeniero de alto nivel en otro proyecto o el director de otro proyecto. Tendrás que elegir si esa autoridad tiene más peso que tu horario de trabajo. De nuevo, tu equipo y tu supervisor pueden ayudarte a tomar esa decisión.

Uno de los deberes más importantes de tu jefe es mantenerte *alejado* de las reuniones. Un buen jefe estará más que dispuesto a defender tu decisión de rechazar la asistencia porque ese jefe está tan preocupado por tu tiempo como tú.

Saliendo de

Las reuniones no siempre salen como se planean. A veces te encuentras sentado en una reunión que habrías rechazado de haber sabido más.

A veces se añaden nuevos temas, o la manía de alguien domina la discusión. A lo largo de los años he desarrollado una regla sencilla: Cuando la reunión se vuelve aburrida, te vas.

De nuevo, tienes la obligación de gestionar bien tu tiempo. Si te encuentras atrapado en una reunión que no es un buen uso de tu tiempo, tienes que encontrar la manera de salir educadamente de esa reunión.

Está claro que no hay que salir furioso de una reunión exclamando "¡Esto es aburrido!". No es necesario ser grosero. Puedes simplemente preguntar, en el momento oportuno, si tu presencia sigue siendo necesaria. Puedes explicar que no puedes permitirte muchas

más tiempo, y pregunte si hay alguna forma de acelerar el debate o debarajar el orden del día.

Lo importante es darse cuenta de que permanecer en una reunión que se ha convertido en una pérdida de tiempo para ti, y a la que ya no puedes contribuir de forma significativa, no es profesional. Tienes la obligación de gastar sabiamente el tiempo y el dinero de tu empleador, así que no es poco profesional elegir un momento adecuado para negociar tu salida.

Tener una agenda y un objetivo

La razón por la que estamos dispuestos a soportar el coste de las reuniones es que a veces necesitamos que los participantes estén juntos en una sala para ayudar a conseguir un objetivo concreto. Para aprovechar bien el tiempo de los participantes, la reunión debe tener un orden del día claro, con tiempos para cada tema y un objetivo establecido.

Si te piden que asistas a una reunión, asegúrate de saber qué discusiones están sobre la mesa, cuánto tiempo se les ha asignado y qué objetivo se quiere alcanzar. Si no consigues una respuesta clara sobre estas cosas,

declina educadamente la asistencia.

Si acude a una reunión y se da cuenta de que el orden del día ha sido secuestrado o abandonado, debe pedir que se posponga el nuevo tema y se siga el orden del día. Si esto no sucede, debes marcharte educadamente cuando sea posible.

Reuniones de pie

Estas reuniones forman parte del canon ágil. Su nombre proviene del hecho de que se espera que los participantes estén de pie mientras se celebra la reunión.

Cada participante responde por turno a tres preguntas:

1. ¿Qué hice ayer?
2. ¿Qué voy a hacer hoy?
3. ¿Qué hay en mi camino?

Eso es todo. Cada pregunta *no* debería requerir más de veinte segundos, por lo que cada participante no debería necesitar más de un minuto. Incluso en un grupo de diez personas, esta reunión debería terminar mucho antes de que transcurran diez minutos.

Reuniones de planificación de iteraciones

Estas son las reuniones más difíciles de hacer bien en el canon ágil. Si se hacen mal, llevan demasiado tiempo. Se necesita habilidad para que estas reuniones salgan bien, una habilidad que vale la pena aprender.

Las reuniones de planificación de la iteración tienen por objeto seleccionar los elementos del backlog que se ejecutarán en la siguiente iteración. Las estimaciones deberían estar ya hechas para los elementos candidatos. La evaluación del valor de negocio ya debería estar hecha. En las organizaciones realmente buenas, las pruebas de aceptación/componentes ya estarán escritas, o al menos esbozadas.

La reunión debe transcurrir con rapidez, discutiendo brevemente cada uno de los temas pendientes y seleccionándolos o rechazándolos. No se deben dedicar más de cinco o diez minutos a un elemento determinado. Si es necesario un debate más largo, debe programarse para otro momento con un subconjunto del equipo.

Mi regla general es que la reunión no debe durar más del 5% del tiempo total de la iteración. Así, para una iteración de una semana (cuarenta horas), la reunión debería terminar en dos horas.

Iteración retrospectiva y demostración

Estas reuniones se celebran al final de cada iteración. Los miembros del equipo discuten lo que ha ido bien y lo que ha ido mal. Las partes interesadas ven una demostración de las nuevas características en

funcionamiento. Se puede abusar de estas reuniones, que pueden absorber mucho tiempo, así que prográmelas 45 minutos antes de la hora de salida del último día de la iteración. No dediques más de 20 minutos a la retrospectiva y 25 minutos a la demostración. Recuerda que sólo han pasado una o dos semanas, así que no debería haber mucho de qué hablar.

Argumentos/desacuerdos

Kent Beck me dijo una vez algo profundo: "Cualquier discusión que no pueda resolverse en cinco minutos no puede resolverse discutiendo". La razón por la que se prolonga tanto es que no hay pruebas claras que apoyen ninguna de las partes. El argumento es probablemente religioso, en contraposición a los hechos.

Los desacuerdos técnicos tienden a dispararse hacia la estratosfera. Cada parte tiene todo tipo de justificaciones para su posición, pero rara vez datos. Sin datos, cualquier argumento que no forje un acuerdo en unos pocos minutos (entre cinco y treinta) simplemente no forjará nunca un acuerdo.

Lo único que hay que hacer es ir a buscar datos.

Algunas personas intentan ganar una discusión por la fuerza de su carácter. Puede que te griten, te echen en cara o actúen con condescendencia. No importa; la fuerza de voluntad no resuelve los desacuerdos por mucho tiempo. Los datos lo hacen.

Algunas personas serán pasivo-agresivas. Estarán de acuerdo sólo para poner fin a la discusión, y luego sabotearán el resultado negándose a participar en la solución. Se dirán a sí mismos: "Así es como lo querían, y ahora van a conseguir lo que querían". Este es probablemente el peor tipo de comportamiento poco profesional que existe. Nunca, nunca hagas esto. Si estás de acuerdo, *debes* comprometerte.

¿Cómo se obtienen los datos necesarios para resolver un desacuerdo? A veces se pueden realizar experimentos, o hacer alguna simulación o modelización. Pero a veces la mejor alternativa es simplemente lanzar una moneda para elegir uno de los dos caminos en cuestión.

Si las cosas funcionan, entonces ese camino era viable. Si hay problemas, se puede dar marcha atrás y seguir el otro camino. Sería conveniente acordar un tiempo y una serie de criterios que ayuden a determinar cuándo se debe abandonar el camino elegido.

Ten cuidado con las reuniones que en realidad son sólo un lugar para desahogar un desacuerdo y recabar apoyos para una u otra parte. Y evita aquellas en las que solo se presenta uno de los contendientes.

Si una discusión debe ser realmente resuelta, pídale a cada uno de los argumentadores que presente su caso al equipo en cinco minutos o menos. A continuación, haz que el equipo vote. Toda la reunión durará menos de

quince minutos.

Focus-Manna

Perdóneme si esta sección parece oler a metafísica de la Nueva Era, o quizás a Dragones y Mazmorras. Es sólo que esta es la forma en que pienso sobre este tema.

La programación es un ejercicio intelectual que requiere largos periodos de concentración y atención. La concentración es un recurso escaso, como el maná. ¹Una vez agotado el maná de la concentración, hay que recargarlo realizando actividades no concentradas durante una hora o más.

No sé qué es este focus-manna, pero tengo la sensación de que es una sustancia física (o posiblemente su falta) que afecta al estado de alerta y a la atención. Sea lo que sea, puedes *sentir* cuando está ahí, y puedes sintiéndolo está

se ha ido. Los desarrolladores profesionales aprenden a gestionar su tiempo para aprovechar su focus-manna. Escribimos código cuando nuestro focus-manna está alto; y hacemos otras cosas menos productivas cuando no lo está.

Focus-manna también es un recurso en decadencia. Si no lo utilizas cuando está ahí, es probable que lo pierdas. Esa es una de las razones por las que las reuniones pueden ser tan devastadoras. Si gastas todo tu focus-manna en una reunión, no te quedará nada para la codificación.

La preocupación y las distracciones también consumen el focus-manna. La pelea que tuviste con tu cónyuge anoche, la abolladura que le hiciste a tu guardabarros esta mañana o la factura que olvidaste pagar la semana pasada te absorberán la concentración-maná rápidamente.

Dormir

No puedo insistir lo suficiente en esto. Tengo la mayor concentración-maná después de una buena noche de sueño. Siete horas de sueño a menudo me dan ocho horas completas de concentración-maná. Los desarrolladores profesionales gestionan su horario de sueño para asegurarse de que han completado su maná de concentración cuando llegan al trabajo por la mañana.

Cafeína

No hay duda de que algunos de nosotros podemos hacer un uso más eficiente de nuestro maná de concentración consumiendo cantidades moderadas de cafeína. Pero hay que tener cuidado. La cafeína también provoca un extraño "temblor" en tu concentración. Demasiada cafeína puede hacer que tu concentración se desvíe en direcciones muy extrañas. Un fuerte subidón de cafeína puede hacer que pierdas un día entero concentrándote en las cosas equivocadas.

El uso y la tolerancia a la cafeína es algo personal. Mi preferencia personales una sola taza de café fuerte por la mañana y una coca-cola light con el almuerzo. A veces duplico esta dosis, pero rara vez hago más que eso.

Recarga de

El maná de la concentración puede recargarse parcialmente si se desconcentra. Un buen paseo, una conversación con los amigos o un rato mirando por la ventana pueden ayudar a recargar el maná de concentración.

Algunas personas meditan. Otros se echan una siesta. Otros escuchan un podcast o hojean una revista.

He descubierto que una vez que el maná se ha ido, no puedes forzar el enfoque. Puedes seguir escribiendo código, pero es casi seguro que tendrás que reescribirlo al día siguiente, o vivir con una masa podrida durante semanas o meses. Así que es mejor tomarse treinta, o incluso sesenta minutos para desenfocarse.

Enfoque muscular

Hay algo peculiar en la práctica de disciplinas físicas como las artes marciales, el tai-chi o el yoga. Aunque estas actividades requieren una gran concentración, es un tipo de concentración diferente a la de la codificación. No es intelectual, es muscular.

Y de alguna manera la concentración muscular ayuda a recargar la concentración mental. Pero es más que una simple recarga. Me parece que un régimen regular de concentración muscular aumenta mi capacidad de concentración mental.

Mi forma elegida de concentración física es montar en bicicleta. Puedo pedalear durante una o dos horas, a veces cubriendo veinte o treinta millas. Voy por un sendero paralelo al río Des Plaines, así que no tengo que lidiar con los coches.

Mientras conduzco, escucho podcasts sobre astronomía o política. A veces simplemente escucho mi música favorita. Y a veces simplemente apago los auriculares y escucho la naturaleza.

Algunas personas se toman el tiempo de trabajar con sus manos. Quizá les guste la carpintería, la construcción de maquetas o la jardinería. Sea cual sea la actividad, hay algo en las actividades que se centran en los músculos que mejora la capacidad de trabajar con la mente.

Entradas y salidas

Otra cosa que considero esencial para concentrarme es equilibrar mi producción con una entrada adecuada. Escribir software es un ejercicio *creativo*. Me parece que soy más creativo cuando estoy expuesto a la creatividad de otras personas. Por eso leo mucha ciencia ficción. La creatividad de esos autores estimula de algún modo mis propios jugos

creativos para el software.

Boxeo de tiempo y tomates

Una forma muy eficaz que he utilizado para gestionar mi tiempo y concentrarme es utilizar la conocida Técnica Pomodoro, ²también conocida como *tomates*. La idea básica es muy sencilla. Se pone un temporizador decocina estándar (tradicionalmente con forma de tomate) durante 25 minutos. Mientras ese temporizador está en marcha, *no* dejas que *nada* interfiera en lo que estás haciendo. Si suena el teléfono, respondes

y pregunta amablemente si puedes devolver la llamada en 25 minutos. Si alguien se detiene para hacerte una pregunta, le preguntas amablemente si puedes devolverle la llamada en 25 minutos. Independientemente de la interrupción, simplemente la aplazas hasta que suene el temporizador. Al fin y al cabo, pocas interrupciones son tan urgentes que no puedan esperar 25 minutos.

Cuando suene el temporizador del tomate, deja de hacer lo que estás haciendo *inmediatamente*. Se ocupa de cualquier interrupción que se haya producido durante el tomate. A continuación, te tomas un descanso de unos cinco minutos. A continuación, se pone el temporizador en marcha durante otros 25 minutos y se empieza el siguiente tomate. Cada cuatro tomates te tomas un descanso más largo, de unos 30 minutos.

Se ha escrito bastante sobre esta técnica, y le insto a que la lea. Sin embargo, la descripción anterior debería proporcionarle lo esencial de la técnica.

Con esta técnica, tu tiempo se divide en tiempo de tomate y tiempo de no tomate. El tiempo de los tomates es productivo. Es dentro de los tomates donde se realiza el trabajo real. El tiempo fuera de los tomates son distracciones, reuniones, descansos u otro tipo de tiempo que no se dedica a las tareas.

¿Cuántos tomates puedes hacer en un día? En un buen día, puedes hacer 12 o incluso 14 tomates. En un mal día, puede que sólo consigas hacer dos o tres. Si los cuentas y los anotas, te harás una idea bastante rápida de cuánto tiempo del día dedicas a ser productivo y cuánto a ocuparte de "cosas".

Algunas personas se sienten tan cómodas con la técnica que calculan sus tareas en tomates y luego miden la velocidad semanal de los mismos. Pero esto no es más que la guinda del pastel. El verdadero beneficio de la Técnica Pomodoro es esa ventana de 25 minutos de tiempo productivo que defiendes agresivamente contra todas las interrupciones.

Evasión

A veces, tu corazón no está en tu trabajo. Puede ser que lo que hay que hacer dé miedo, sea incómodo o aburrido. Tal vez pienses que te va a

obligar a una confrontación o que te va a llevar a una ratonera ineludible. O puede que simplemente no quieras hacerlo.

Inversión de prioridades

Sea cual sea la razón, encuentras formas de evitar hacer el trabajo real. Te convences de que hay algo más urgente y lo haces en su lugar. Esto se llama *inversión de prioridades*. Aumentas la prioridad de una tarea para poder posponer la que tiene la verdadera prioridad. La inversión de prioridades es una mentira que nos contamos a nosotros mismos. No podemos afrontar lo que hay que hacer, así que nos convencemos de que otra tarea es más importante. Sabemos que no lo es, pero nos mentimos a nosotros mismos.

En realidad, no nos estamos mintiendo a nosotros mismos. Lo que realmente estamos haciendo es preparar la mentira que diremos cuando alguien nos pregunte qué hacemos y por qué lo hacemos. Estamos construyendo una defensa para protegernos del juicio de los demás.

Es evidente que se trata de un comportamiento poco profesional. Los profesionales evalúan la prioridad de cada tarea, sin tener en cuenta sus temores y deseos personales, y ejecutan esas tareas en orden de prioridad.

Callejones sin salida

Los callejones sin salida son una realidad para todos los artesanos del software. A veces, al tomar una decisión, se pierde en un camino técnico que no lleva a ninguna parte. Cuanto más comprometido esté con su decisión, más tiempo vagará por el desierto. Si has apostado por tu reputación profesional, vagarás para siempre.

La prudencia y la experiencia te ayudarán a evitar ciertos callejones sin salida, pero nunca los evitarás todos. Así que la verdadera habilidad que necesitas es darte cuenta rápidamente de cuándo estás en uno, y tener el valor de echarse atrás. Esto se denomina a veces "*la regla de los agujeros*": Cuando estés en uno, deja de cavar.

Los profesionales evitan comprometerse tanto con una idea que no puedan abandonarla y dar la vuelta. Mantienen la mente abierta a otras ideas para que, cuando lleguen a un callejón sin salida, sigan teniendo otras opciones.

Marismas, ciénagas, pantanos y otros líos

Peor que los callejones sin salida son los líos. Los líos te ralentizan, pero no te detienen. Los desórdenes te impiden avanzar, pero puedes hacerlo por la fuerza bruta. Los líos son peores que los callejones sin salida porque siempre puedes ver el camino hacia delante, y siempre parece más corto que el camino de vuelta (pero no lo es).

He visto productos arruinados y empresas destruidas por líos de

software. He visto cómo la productividad de los equipos disminuía desde el jitterbug hasta el dirge en tan sólo unos meses. Nada tiene un efecto negativo más profundo o duradero en la productividad de un equipo de software que un desorden. Nada.

El problema es que empezar un lío, como meterse en un callejón sin salida, es inevitable. La experiencia y la prudencia pueden ayudarte a evitarlos, pero en algún momento tomarás una decisión que te lleve a un lío.

La progresión de este desorden es insidiosa. Creas una solución a un problema simple, teniendo cuidado de mantener el código simple y limpio. A medida que el problema crece en alcance y complejidad, se amplía el código base, manteniéndolo tan limpio como sea posible. En algún momento te das cuenta de que tomaste una decisión de diseño equivocada cuando empezaste, y que tu código no se adapta bien a la dirección en la que se mueven los requisitos.

Este es el punto de inflexión. Todavía se puede volver atrás y arreglar el diseño. Pero también puedes seguir avanzando. Volver atrás parece costoso porque tendrás que rehacer el código existente, pero volver atrás *nunca* será más fácil que ahora. Si sigues adelante, llevarás el sistema a un pantano del que nunca podrá escapar.

Los profesionales temen los líos mucho más que los callejones sin salida. Siempre están atentos a los líos que empiezan a crecer sin límites, y gastarán todo el esfuerzo necesario para escapar de ellos lo antes y lo más rápido posible.

Avanzar a través de un pantano, cuando *sabes que es* un pantano, es la peor clase de inversión de prioridades. Al avanzar te mientes a ti mismo, mientes a tu equipo, mientes a tu empresa y mientes a tus clientes. Les estás diciendo que todo irá bien, cuando en realidad te diriges a una perdición compartida.

Conclusión:

Los profesionales del software son diligentes en la gestión de su tiempo y su concentración. Comprenden las tentaciones de la inversión de prioridades y la combaten como una cuestión de honor. Mantienen sus opciones abiertas, con una mentalidad abierta a las soluciones alternativas. Nunca se sienten tan atraídos por una solución que no puedan

abandonarla. Y siempre están atentos a los desórdenes crecientes, y los limpian en cuanto los reconocen. No hay

Una visión más triste que la de un equipo de desarrolladores de software arrastrándose infructuosamente por una ciénaga cada vez más profunda.

10. Estimación



La estimación es una de las actividades más sencillas, pero más temibles, a las que se enfrentan los profesionales del software. De ella depende gran parte del valor del negocio. Gran parte de nuestra reputación depende de ella. Es la causa de muchos de nuestros problemas y fracasos. Es la principal cuña que se ha abierto entre la gente de negocios y los desarrolladores. Es la fuente de casi toda la desconfianza que rige esa relación.

En 1978, fui el desarrollador principal de un programa Z-80 embebido de 32K escrito en lenguaje ensamblador. El programa se grabó en 32 chips EPROM de $1K \times 8$. Estos 32 chips se insertaron en tres placas, cada una de las cuales contenía 12 chips.

Teníamos cientos de dispositivos sobre el terreno, instalados en las centrales telefónicas de todo Estados Unidos. Cada vez que corregíamos un error o añadíamos una función, teníamos que enviar a los técnicos de servicio a cada una de esas unidades y hacer que sustituyeran los 32 chips.

Esto fue una pesadilla. Los chips y las placas eran frágiles. Las clavijas de los chips podían doblarse y romperse. La constante flexión de las placas podía dañar las juntas de soldadura. El riesgo de rotura y error era enorme. El coste para la empresa era demasiado alto.

Mi jefe, Ken Finder, vino a verme y me pidió que lo solucionara. Lo que él quería era una manera de hacer un cambio en un chip que no requiriera que todos los otros chips cambiaran. Si has leído mis libros, o escuchado mis charlas, sabes que despotrico mucho sobre la despleabilidad independiente. Aquí es donde aprendí por primera vez esa lección.

Nuestro problema era que el software era un único ejecutable enlazado. Si se

añadía una nueva línea de código al programa, todas las direcciones de las siguientes líneas de código cambiaban. Dado que cada chip simplemente contenía 1K del espacio de direcciones, el contenido de prácticamente todos los chips cambiaría.

La solución era bastante sencilla. Había que desacoplar cada chip de todos los demás. Cada uno tenía que convertirse en una unidad de compilación independiente que pudiera quemarse independientemente de todos los demás.

Así que medí los tamaños de todas las funciones de la aplicación y escribí un sencillo programa que las encajaba, como un rompecabezas, en cada uno de los chips, dejando 100 bytes de espacio más o menos para la expansión. Al principio de cada chip puse una tabla de punteros a todas las funciones de ese chip. Al arrancar, estos punteros se trasladaban a la RAM. Todo el código del sistema fue cambiado para que las funciones fueran llamadas sólo a través de estos vectores de RAM y nunca directamente.

Sí, lo tienes. Las fichas eran objetos, con vtables. Todas las funciones eran polimórficas. Y, sí, así es como aprendí algunos de los principios de OOD, mucho antes de saber lo que era un objeto.

Las ventajas eran enormes. No sólo podíamos desplegar chips individuales, sino que también podíamos hacer parches sobre el terreno trasladando funciones a la RAM y redirigiendo los vectores. Esto facilitó mucho la depuración sobre el terreno y la aplicación de parches en caliente.

Pero estoy divagando. Cuando Ken acudió a mí y me pidió que solucionara este problema, sugirió algo sobre punteros a funciones. Pasé un día o dos formalizando la idea y luego le presenté un plan detallado. Me preguntó cuánto tiempo me llevaría, y le respondí que me llevaría aproximadamente un mes.

Se tardó *tres* meses.

Sólo me he emborrachado dos veces en mi vida, y sólo una vez de *verdad*. Fue en la fiesta de Navidad de Teradyne en 1978. Tenía 26 años.

La fiesta se celebró en la oficina de Teradyne, que era en su mayor parte un espacio de laboratorio abierto. Todo el mundo llegó temprano, y luego hubo una enorme ventisca que impidió que la banda y el servicio de catering llegaran. Afortunadamente, había mucha bebida.

No recuerdo mucho de esa noche. Y lo que recuerdo desearía no hacerlo. Pero voy a compartir un momento conmovedor con ustedes.

Estaba sentada con las piernas cruzadas en el suelo con Ken (mi jefe, que entonces tenía 29 años y *no* estaba borracho) llorando por el tiempo que me estaba llevando el trabajo de vectorización. El alcohol había liberado mis miedos e inseguridades reprimidas sobre mi estimación. No *creo que* mi cabeza estuviera en su regazo, pero mi memoria no es muy clara en ese tipo

de detalles.

Recuerdo haberle preguntado si estaba enfadado conmigo y si creía que estaba tardando demasiado. Aunque la noche fue borrosa, su respuesta ha quedado clara a lo largo de las décadas siguientes. Me dijo: "Sí, creo que has tardado mucho, pero veo que estás trabajando duro en ello y haciendo buenos progresos. Es algo que realmente necesitamos. Así que, no, no estoy enfadado".

¿Qué es un presupuesto?

El problema es que vemos las estimaciones de diferentes maneras. A las empresas les gusta ver las estimaciones como compromisos. A los desarrolladores les gusta ver las estimaciones como conjeturas. La diferencia es profunda.

Un compromiso

Un compromiso es algo que debes cumplir. Si te comprometes a hacer algo en una fecha determinada, simplemente *tienes* que hacerlo en esa fecha. Si eso significa que tienes que trabajar 12 horas al día, los fines de semana, saltándote las vacaciones familiares, que así sea. Has asumido el compromiso y tienes que cumplirlo.

Los profesionales no se comprometen a menos que *sepan que pueden* cumplirlos. Es tan sencillo como eso. Si te piden que te comprometas a algo que no estás *seguro de* poder hacer, entonces tienes el honor de rechazarlo. Si le piden que se comprometa a una fecha que sabe que *puede cumplir*, pero que requeriría largas horas, fines de semana y la ausencia de vacaciones familiares, entonces la elección es suya; pero más vale que esté dispuesto a hacer lo que sea necesario.

El compromiso tiene que ver con la *certeza*. Otras personas van a aceptar tus compromisos y a hacer planes basados en ellos. El coste de no cumplirlos

El riesgo de los compromisos, para ellos y para su reputación, es enorme. Faltar a un compromiso es un acto de deshonestidad sólo un poco menos oneroso que una mentira manifiesta.

Una estimación

Una estimación es una suposición. No implica ningún compromiso. No se hace ninguna promesa. No hacer una estimación no es en absoluto deshonesto. La razón por la *que* hacemos estimaciones es porque *no sabemos* cuánto tiempo llevará algo.

Por desgracia, la mayoría de los desarrolladores de software son pésimos estimadores. Esto no se debe a que haya una habilidad secreta para estimar, no la hay. La razón por la que a menudo somos tan malos estimando es porque no entendemos la verdadera naturaleza de una estimación.

Una estimación no es un número. Una estimación es una *distribución*. Considera: Mike: "¿Cuál es tu estimación para completar la tarea de Frazzle?" Pedro: "Tres días".

¿Realmente Peter va a terminar en tres días? Es posible, pero ¿hasta qué punto es probable? La respuesta a eso es: No tenemos ni idea. ¿Qué quiso decir Peter, y qué ha aprendido Mike? Si Mike vuelve en tres días, ¿debería sorprenderse si Peter no ha terminado? ¿Por qué habría de hacerlo? Peter no se ha comprometido. Peter no le ha dicho qué probabilidad hay de que sean tres días frente a cuatro o cinco.

¿Qué habría pasado si Mike le hubiera preguntado a Peter qué probabilidad tenía su estimación de tres días?

MIKE: "¿Qué probabilidades hay de que termines en tres días? Pedro: "Bastante probable".

MIKE: "¿Puedes poner un número?"

Pedro: "Cincuenta o sesenta por ciento".

MIKE: "Así que es muy probable que te lleve cuatro días".

Pedro: "Sí, de hecho puede que incluso me lleve cinco o seis, aunque lo dudo". MIKE: "¿Cuánto lo dudas?"

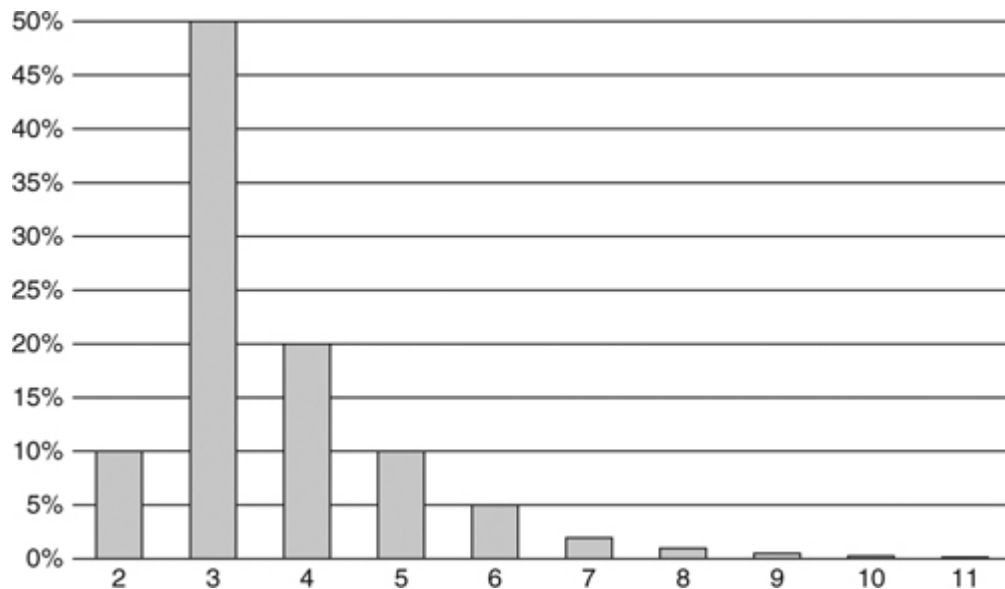
Pedro: "Oh, no sé... Estoy noventa y cinco por ciento seguro de que voy a terminar antes de que pasen seis días".

MIKE: "¿Quieres decir que pueden ser siete días?"

Pedro: "Bueno, sólo si todo sale mal. Diablos, si *todo* sale mal, podría llevarme diez o incluso once días. Pero no es muy probable que salgan tantas cosas mal".

Ahora estamos empezando a afinar la verdad. La estimación de Peter es una *distribución de probabilidad*. En su mente, Pedro ve la probabilidad de finalización como la que se muestra en la [Figura 10-1](#).

Figura 10-1. Distribución de probabilidades



Puedes ver por qué Peter dio la estimación original de tres días. Es la barra más alta del gráfico. Así que en la mente de Peter es la duración más probable para la tarea. Pero Mike ve las cosas de otra manera. Mira la cola de la derecha del gráfico y le preocupa que Peter pueda tardar realmente once días en terminar.

¿Debería Mike estar preocupado por esto? Por supuesto. Murphy¹ saldrá con la suya con Peter, así que algunas cosas probablemente saldrán mal.

Compromisos implícitos

Así que ahora Mike tiene un problema. No está seguro del tiempo que le llevará a Peter realizar la tarea. Para minimizar esa incertidumbre puede pedirle a Peter un compromiso. Esto es algo que Pedro no está en condiciones de dar.

MIKE: "Peter, ¿puedes darme una fecha concreta de cuándo vas a terminar?"

Peter: "No, Mike. Como he dicho, probablemente estará hecho entres, tal vez cuatro, días".

MIKE: "¿Podemos decir cuatro entonces?" Pedro: "No, *podrían* ser cinco o seis".

Hasta ahora, todo el mundo se está comportando de forma justa. Mike ha pedido un compromiso y Peter se ha negado cuidadosamente a dárselo. Así que Mike intenta una táctica diferente:

MIKE: "Vale, Peter, pero ¿puedes intentar que no sean más de seis días?"

La petición de Mike suena bastante inocente, y ciertamente Mike no tiene malas intenciones. Pero, ¿qué es exactamente lo que Mike le pide a Peter? ¿Qué significa "intentar"?

Ya hablamos de esto en el [capítulo 2](#). La palabra *intentar* es un término cargado. Si Pedro acepta "probar", entonces se está comprometiendo a seis días. No hay otra manera de interpretarlo. Estar de acuerdo en intentarlo es estar de acuerdo en tener éxito.

¿Qué otra interpretación podría haber? ¿Qué es, precisamente, lo que Pedro va a hacer para "probar"? ¿Va a trabajar más de ocho horas? Eso está claramente implícito. ¿Va a trabajar los fines de semana? Sí, eso también está implícito. ¿Se saltará las vacaciones familiares? Sí, eso también es parte de la implicación. Todas esas cosas son parte del "intento". Si Peter no hace esas cosas, entonces Mike podría acusarlo de no esforzarse lo suficiente.

Los profesionales distinguen claramente entre estimaciones y compromisos. No se comprometen a menos que sepan con certeza que lo van a conseguir. Tienen cuidado de no hacer ningún compromiso *implícito*. Comunican la distribución de probabilidad de sus estimaciones con la mayor claridad posible, para que los directivos puedan hacer planes adecuados.

PERT

En 1957 se creó la Técnica de Evaluación y Revisión de Programas (PERT) para apoyar el proyecto del submarino Polaris de la Marina estadounidense. Uno de los elementos del PERT es la forma de calcular las estimaciones. El esquema proporciona una forma muy sencilla, pero muy eficaz, de convertir las estimaciones en distribuciones de probabilidad adecuadas para los gestores.

Cuando se estima una tarea, se proporcionan tres números. Esto se llama *trivariante*:

- **O:** Estimación optimista. Esta cifra es *tremendamente* optimista. Sólo se podría realizar la tarea tan rápidamente si todo saliera bien. De hecho, para que las matemáticas funcionen este número debería tener mucho

menos de un 1% de probabilidad de ocurrencia. ²En el caso de Peter, esto sería 1 día, como se muestra en [la figura 10-1](#).

- **N: Estimación nominal.** Es la estimación con mayor probabilidad de éxito. Si se dibujara un gráfico de barras, sería la barra más alta, como se muestra en [la figura 10-1](#). Son 3 días.
- **P:** Estimación pesimista. Una vez más, esta estimación es *tremendamente* pesimista. Debería incluir todo excepto los huracanes, la guerra nuclear, los agujeros negros perdidos y otras catástrofes. De nuevo, las matemáticas sólo funcionan si este número tiene mucho menos de un 1% de posibilidades de éxito. En el caso de Peter este número está fuera de la tabla de la derecha. Así que 12 días.

Dadas estas tres estimaciones, podemos describir la distribución de

probabilidad de la siguiente manera:

$$\mu = \frac{O+4N+P}{6}$$

μ es la duración prevista de la tarea. En el caso de Peter es $(1+12+12)/6$, es decir, unos 4,2 días. Para la mayoría de las tareas, esta cifra será algo pesimista porque la cola derecha de la distribución es más larga que la cola izquierda.³

$$\sigma = \frac{P-O}{6}$$

σ es la desviación estándar⁴ de la distribución de probabilidad de la tarea. Es una medida de lo incierta que es la tarea. Cuando este número es grande, la incertidumbre también lo es. En el caso de Peter, este número es $(12 - 1)/6$, es decir, aproximadamente 1.8 días.

Teniendo en cuenta la estimación de Peter de 4,2/1,8, Mike entiende que esta tarea probablemente se realizará en cinco días, pero también podría tardar 6,0 incluso 9, días en completarse.

Pero Mike no se limita a gestionar una tarea. Está gestionando un proyecto de muchas tareas. Peter tiene tres de esas tareas en las que debe trabajar en secuencia. Peter ha estimado estas tareas como se muestra en [la Tabla 10-1](#).

Tabla 10-1. Tareas de Peter

Task	Optimistic	Nominal	Pessimistic	μ	σ
Alpha	1	3	12	4.2	1.8
Beta	1	1.5	14	3.5	2.2
Gamma	3	6.25	11	6.5	1.3

¿Qué pasa con esa tarea "beta"? Parece que Peter está bastante seguro de sí mismo, pero que podría salir algo mal que le hiciera descarrilar significativamente. ¿Cómo debería interpretarlo Mike? ¿Cuánto tiempo debe planear Mike para que Peter complete las tres tareas?

Resulta que, con unos simples cálculos, Mike puede combinar todas las tareas de Peter y obtener una distribución de probabilidad para todo el conjunto de tareas. Las matemáticas son bastante sencillas:

$$\mu_{\text{sequence}} = \sum \mu_{\text{task}}$$

Para cualquier secuencia de tareas, la duración prevista de esa secuencia es la simple suma de todas las duraciones previstas de las tareas de esa secuencia. Por lo tanto, si Pedro tiene que completar tres tareas y sus estimaciones son 4,2/1,8, 3,5/2,2 y 6,5/1,3, es probable que Pedro termine las tres en unos 14 días: $4.2 + 3.5 + 6.5$.

$$\sigma_{\text{sequence}} = \sqrt{\sum \sigma_{\text{task}}^2}$$

La desviación estándar de la secuencia es la raíz cuadrada de la suma

de los cuadrados de las desviaciones estándar de las tareas. Por lo tanto, la desviación estándar de las tres tareas de Pedro es aproximadamente 3.

$$\begin{aligned} & (1.8^2 + 2.2^2 + 1.3^2)^{1/2} = \\ & (3.24 + 2.48 + 1.69)^{1/2} = 9.77^{1/2} = \sim 3.13 \end{aligned}$$

Esto le dice a Mike que las tareas de Peter probablemente durarán 14 días, pero bien podrían durar 17 días (1σ) y posiblemente incluso 20 días (2σ). Incluso podría tardar más, pero es bastante improbable.

Vuelve a mirar la tabla de estimaciones. ¿Sientes la presión de realizar las tres tareas en cinco días? Después de todo, las estimaciones en el mejor de los casos son 1, 1 y

3. Incluso las estimaciones nominales sólo suman 10 días. ¿Cómo hemos llegado hasta los 14 días, con una posibilidad de 17 o 20? La respuesta es que la incertidumbre en esas tareas se acumula de forma que añade *realismo* al plan.

Si es usted un programador con más de unos años de experiencia, es probable que haya visto proyectos que se estimaron de forma optimista y que tardaron entre tres y cinco veces más de lo esperado. El sencillo

esquema PERT que acabamos de mostrar es una forma razonable de ayudara evitar el establecimiento de expectativas optimistas. Software

Los profesionales son muy cuidadosos a la hora de establecer unas expectativas razonables a pesar de la presión por intentar ir rápido.

Tareas de estimación

Mike y Peter estaban cometiendo un terrible error. Mike le preguntaba a Peter cuánto tiempo le llevarían sus tareas. Peter daba respuestas trivariantes honestas, pero ¿qué pasa con las opiniones de sus compañeros de equipo? ¿Podrían tener una idea diferente?

El recurso de estimación más importante que tienes son las personas que te rodean. Pueden ver cosas que tú no ves. Pueden ayudarte a estimar tus tareas con más precisión de la que tú puedes estimar por ti mismo.

Delphi de banda ancha

En los años 70, Barry Boehm nos presentó una técnica de estimación llamada "delphi de banda ancha". ⁵A lo largo de los años ha habido muchas variaciones. Algunas son formales, otras informales; pero todas tienen algo en común: el consenso.

La estrategia es sencilla. Un equipo de personas se reúne, discute una tarea, estima la tarea, e itera la discusión y la estimación hasta llegar a un acuerdo.

El enfoque original descrito por Boehm implicaba varias reuniones y documentos que suponen demasiada ceremonia y gastos generales para mi gusto. Yo prefiero enfoques sencillos y poco costosos como el siguiente.

Dedos voladores

Todos se sientan alrededor de una mesa. Las tareas se discuten de una en una. Para cada tarea se discute lo que implica la tarea, lo que podría confundirla o complicarla y cómo podría llevarse a cabo. A continuación, los participantes ponen las manos debajo de la mesa y levantan de 0 a 5 dedos en función del tiempo que creen que llevará la tarea. El moderador cuenta 1-2-3 y todos los participantes muestran sus manos a la vez.

Si todos están de acuerdo, pasan a la siguiente tarea. Si no, continúan la discusión para determinar por qué no están de acuerdo. Lo repiten hasta que se pongan de acuerdo.

El acuerdo no tiene por qué ser absoluto. Mientras las estimaciones sean cercanas, es suficiente. Así, por ejemplo, un par de 3s y 4s es un acuerdo.

Sin embargo, si todo el mundo levanta 4 dedos excepto una persona que levanta 1 dedo, entonces tienen algo de lo que hablar.

La escala de la estimación se decide al principio de la reunión. Puede ser el número de días para una tarea, o puede ser alguna escala más interesante como "dedos por tres" o "dedos al cuadrado".

La simultaneidad de la visualización de los dedos es importante. No queremos que la gente cambie sus estimaciones en función de lo que vean hacer a otras personas.

Planificación del póker

En 2002, James Grenning escribió un delicioso artículo en el ⁶que describía el "Planning Poker". Esta variación delphi de banda ancha se ha hecho tan popular que varias empresas diferentes han utilizado la idea para hacer regalos de marketing en forma de barajas de cartas de póquer de planificación. ⁷Hay incluso un sitio web llamado planningpoker.com que se puede utilizar para hacer póquer de planificación en la red con equipos distribuidos.

La idea es muy sencilla. Para cada miembro del equipo de estimación, se reparte una mano de cartas con diferentes números. Los números del 0 al 5 funcionan bien, y hacen que este sistema sea lógicamente equivalente a los *dedos voladores*.

Elige una tarea y discútela. En un momento dado, el moderador pide a todos que elijan una tarjeta. Los miembros del equipo sacan una tarjeta que coincida con su estimación y la sostienen con el reverso hacia afuera para que nadie más pueda ver el valor de la tarjeta. A continuación, el moderador dice a todos que muestren sus cartas.

El resto es como los dedos voladores. Si hay acuerdo, se acepta la

estimación. En caso contrario, las cartas vuelven a la mano y los jugadores continúan discutiendo la tarea.

Se ha dedicado mucha "ciencia" a elegir los valores correctos de las cartas para una mano. Algunos han llegado a utilizar cartas basadas en la serie de Fibonacci. Otros han incluido cartas para el infinito y el signo de interrogación.

Personalmente, creo que basta con cinco cartas etiquetadas como 0, 1, 3, 5, 10.

Estimación de la afinidad

Lowell Lindstrom me mostró hace varios años una variante especialmente singular delphi de banda ancha. He tenido bastante suerte con este enfoque con varios clientes y equipos.

Todas las tareas se escriben en tarjetas, sin que se vea ninguna estimación. El equipo de estimación se sitúa alrededor de una mesa o una pared con las tarjetas repartidas al azar. Los miembros del equipo no hablan, simplemente empiezan a ordenar las tarjetas entre sí. Las tareas que llevan más tiempo se desplazan a la derecha.

Las tareas más pequeñas se mueven a la izquierda.

Cualquier miembro del equipo puede mover cualquier carta en cualquier momento, incluso si ya ha sido movida por otro miembro. Cualquier carta movida más de n veces se aparta para ser discutida.

Al final, la clasificación silenciosa se acaba y se puede empezar a discutir. Se exploran los desacuerdos sobre el orden de las tarjetas. Puede haber algunas sesiones rápidas de diseño o algunos marcos de alambre dibujados a mano para ayudar a obtener el consenso.

El siguiente paso es trazar líneas entre las tarjetas que representen los tamaños de los cubos. Estos cubos pueden estar en días, semanas o puntos. Lo tradicional es que haya cinco cubos en una secuencia de Fibonacci (1, 2, 3, 5, 8).

Estimaciones trivariadas

Estas técnicas delphi de banda ancha son buenas para elegir una única estimación nominal para una tarea. Pero, como hemos dicho antes, la mayoría de las veces queremos tres estimaciones para poder crear una distribución de probabilidad. Los valores optimistas y pesimistas de cada tarea pueden generarse muy rápidamente utilizando cualquiera de las variantes delphi de banda ancha. Por ejemplo, si se utiliza el póquer de planificación, basta con pedir al equipo que levante las cartas para su estimación pesimista y luego tomar la más alta. Se hace lo mismo con la estimación optimista y se toma la más baja.

La ley de los grandes números

Las estimaciones están llenas de errores. Por eso se llaman estimaciones.

Una forma de gestionar el error es aprovechar la *Ley de los Grandes Números*.⁸ Una implicación de esta ley es que si se divide una tarea grande en muchas tareas más pequeñas y se estiman de forma independiente, la suma de las estimaciones de las tareas pequeñas será más precisa que una sola estimación de la tarea más grande. La razón de este aumento de la precisión es que los errores de las tareas pequeñas tienden a integrarse.

Francamente, esto es optimista. Los errores en las estimaciones tienden a la subestimación y no a la sobreestimación, por lo que la integración no es perfecta. Dicho esto,

Dividir las tareas grandes en pequeñas y estimar las pequeñas de forma independiente sigue siendo una buena técnica. Algunos de los errores *se integran*, y dividir las tareas es una buena manera de entender mejor estas tareas y descubrir las sorpresas.

Conclusión:

Los desarrolladores de software profesionales saben cómo proporcionar a la empresa estimaciones prácticas que ésta puede utilizar con fines de planificación. No hacen promesas que no puedan cumplir y no asumen compromisos que no estén seguros de poder cumplir.

Cuando los profesionales se comprometen, aportan cifras concretas y luego las cumplen. Sin embargo, en la mayoría de los casos los profesionales no asumen tales compromisos. En su lugar, proporcionan estimaciones probabilísticas que describen el tiempo de finalización previsto y la varianza probable.

Los desarrolladores profesionales trabajan con los demás miembros de su equipo para consensuar las estimaciones que se entregan a la dirección.

Las técnicas descritas en este capítulo son *ejemplos* de algunas de las diferentes formas en que los desarrolladores profesionales crean estimaciones prácticas. No son las únicas técnicas de este tipo y no son necesariamente las mejores. Son simplemente técnicas que me han funcionado bien.

Bibliografía

[**McConnell2006**]: Steve McConnell, *Software Estimation: Demystifying the Black Art*, Redmond, WA: Microsoft Press, 2006.

[**Boehm81**]: Barry W. Boehm, *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

[**Grenning2002**]: James Grenning, "Planning Poker or How to Avoid Analysis Paralysis while Release Planning", abril de 2002, <http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html>

11. Presión



Imagínese que está teniendo una experiencia extracorporal, observándose a sí mismo en una mesa de operaciones mientras un cirujano le realiza una operación a corazón abierto. Ese cirujano está intentando salvar su vida, pero el tiempo es limitado, por lo que está operando bajo un plazo, *literalmente un plazo*.

¿Cómo quiere que se comporte ese médico? ¿Quiere que parezca tranquilo y sereno? ¿Quiere que dé órdenes claras y precisas a su personal de apoyo? ¿Quiere que siga su formación y se atenga a sus disciplinas?

¿O quieres que sude y diga palabrotas? ¿Quiere que golpee y lance instrumentos? ¿Quieres que culpe a la dirección por sus expectativas poco realistas y que se queje continuamente del tiempo? ¿Quiere que se comporte como un profesional o como un desarrollador típico?

El desarrollador profesional es tranquilo y decisivo bajo presión. A medida que la presión aumenta, se atiene a su formación y a sus disciplinas, sabiendo que son la mejor manera de cumplir los plazos y los compromisos que le apremian.

En 1988 trabajaba en Clear Communications. Se trataba de una empresa emergente que nunca llegó a arrancar. Quemamos nuestra primera ronda de financiación y

luego tuvo que ir por un segundo, y luego un tercero.

La visión inicial del producto sonaba bien, pero la arquitectura del producto nunca pudo aterrizar. Al principio, el producto era tanto software como hardware. Luego pasó a ser sólo software. La plataforma de software cambió de PC a Sparcstations. Los clientes cambiaron de gama alta a gamabaja. Con el tiempo, incluso la intención original del producto se desvió, ya que la

empresa trató de encontrar algo que generara ingresos. En los casi cuatro años que pasé allí, no creo que la empresa viera un céntimo de ingresos.

No hace falta decir que los desarrolladores de software estábamos sometidos a una gran presión. Hubo bastantes noches muy largas, y fines de semana aún más largos pasados en la oficina en la terminal. Se escribían funciones en C *de 3.000 líneas*. Había discusiones con gritos e insultos.

Hubo intrigas y subterfugios. Había puñetazos en las paredes, bolígrafos lanzados con rabia contra las pizarras, caricaturas de colegas molestos grabadas en las paredes con las puntas de los lápices, y había un suministro interminable de ira y estrés.

Los plazos estaban condicionados por los eventos. Las funciones tenían que estar listas para las ferias comerciales o las demostraciones a los clientes. Cualquier cosa que pidiera un cliente, por tonta que fuera, la teníamos lista para la siguiente demostración. El tiempo era siempre demasiado corto. El trabajo siempre iba con retraso. Los calendarios eran siempre abrumadores.

Si trabajas 80 horas en una semana, puedes ser un héroe. Si te dedicas a hacer un desastre para una demostración a un cliente, puedes ser un héroe. Si lo hacías lo suficiente, podías ser ascendido. Si no lo hacías, podías ser despedido. Era una empresa emergente: todo se basaba en el "sudor". Y en 1988, con casi 20 años de experiencia a mis espaldas, me lo creí.

Yo era el director de desarrollo que decía a los programadores que trabajaban para mí que tenían que trabajar más y más rápido. Yo era uno de los que trabajaban 80 horas, escribiendo funciones en C de 3.000 líneas a las 2 de la madrugada mientras mis hijos dormían en casa sin su padre. Yo era el que lanzaba los bolígrafos y gritaba. Hacía que despidieran a la gente si no se ponía en forma. Era horrible. Yo era horrible.

Entonces llegó el día en que mi mujer me obligó a mirarme largamente en el espejo. No me gustó lo que vi. Me dijo que no era muy agradable estar con ella. Tuve que aceptar. Pero no me gustaba, así que salí de la casa entromba.

rabia y empecé a caminar sin rumbo. Caminé durante treinta minutos más o menos, hirviendo mientras caminaba; y entonces empezó a llover.

Y algo hizo clic dentro de mi cabeza. Empecé a reírme. Me reí de mi locura. Me reí de mi estrés. Me reí del hombre del espejo, el pobre imbécil que se había hecho la vida imposible a sí mismo y a los demás en nombre de... ¿qué?

Todo cambió ese día. Dejé los horarios locos. Dejé el estilo de vida de alto estrés. Dejé de lanzar bolígrafos y de escribir funciones C de 3.000 líneas. Decidí que iba a disfrutar de mi carrera haciéndola bien, no haciéndola estúpidamente.

Dejé ese trabajo tan profesionalmente como pude, y me convertí en consultor. Desde ese día no he vuelto a llamar a otra persona "jefe".

Evitar la presión

La mejor manera de mantener la calma bajo presión es evitar las situaciones que la provocan. Puede que esa evitación no elimine la presión por completo, pero puede contribuir en gran medida a minimizar y acortar los periodos de alta presión.

Compromisos

Como descubrimos en [el capítulo 10, es](#) importante evitar comprometerse con plazos que no estamos seguros de poder cumplir. La empresa siempre querrá estos compromisos porque quiere eliminar el riesgo. Lo que debemos hacer es asegurarnos de que el riesgo se cuantifica y se presenta a la empresa para que pueda gestionarlo adecuadamente. Aceptar compromisos poco realistas frustra este objetivo y hace un flaco favor tanto a la empresa como a nosotros mismos.

A veces los compromisos se hacen por nosotros. A veces nos encontramos con que nuestra empresa ha hecho promesas a los clientes sin consultarnos. Cuando esto ocurre, estamos obligados a ayudar a la empresa a encontrar la manera de cumplir esos compromisos. Sin embargo, *no estamos* obligados a *aceptar* los compromisos.

La diferencia es importante. Los profesionales siempre ayudarán a la empresa a encontrar la manera de alcanzar sus objetivos. Pero los profesionales no aceptan necesariamente los compromisos que la empresa les hace. Al final, si no podemos encontrar la manera de cumplir las promesas hechas por la empresa, entonces las personas que hicieron las promesas deben aceptar la responsabilidad.

Es fácil decirlo. Pero cuando tu negocio está fracasando y tu sueldo se retrasa por el incumplimiento de tus compromisos, es difícil no sentir la presión. Pero si te has comportado de forma profesional, al menos puedes mantener la cabeza alta mientras buscas un nuevo trabajo.

Mantenerse limpio

La forma de ir rápido, y de mantener los plazos a raya, es mantenerse limpio. Los profesionales no sucumben a la tentación de crear un desorden para avanzar rápidamente. Los profesionales saben que "rápido y sucio" es un oxímoron. Sucio significa siempre lento.

Podemos evitar la presión manteniendo nuestros sistemas, nuestro código y nuestro diseño lo más limpio posible. Esto no significa que pasemos interminables horas puliendo el código. Simplemente significa que no toleramos el desorden. Sabemos que los desórdenes nos retrasan, nos hacen perder fechas e incumplir compromisos. Así que hacemos el mejor trabajo

posible y mantenemos nuestra producción tan limpia como podemos.

Disciplina de crisis

Sabes lo que crees observándote a ti mismo en una crisis. Si en una crisis sigues tus disciplinas, entonces crees realmente en esas disciplinas. Por otro lado, si cambias tu comportamiento en una crisis, entonces no crees verdaderamente en tu comportamiento normal.

Si sigues la disciplina del Desarrollo Dirigido por Pruebas en tiempos sin crisis pero la abandonas durante una crisis, entonces no confías realmente en que TDD sea útil. Si mantienes tu código limpio en tiempos normales pero haces líos en una crisis, entonces no crees realmente que los líos te frenen. Si emparejas en una crisis pero no emparejas normalmente, entonces crees que emparejar es más eficiente que no emparejar.

Elige disciplinas con las que te sientas cómodo en una crisis. *Luego, síguelas siempre.* Seguir estas disciplinas es la mejor manera de evitar entrar en una crisis.

No cambie su comportamiento cuando llegue la crisis. Si tu disciplina es la mejor manera de trabajar, debes seguirla incluso en plena crisis.

Manejo de la presión

Prevenir, mitigar y eliminar la presión está muy bien, pero a veces la presión llega a pesar de las mejores intenciones y prevenciones. A veces el proyecto se alarga más de lo que se pensaba. A veces, el diseño inicial es erróneo y hay que rehacerlo.

A veces se pierde a un valioso miembro del equipo o a un cliente. A veces se hace un compromiso que no se puede cumplir. ¿Y entonces qué?

Que no cunda el pánico

Controla tu estrés. Las noches de insomnio no te ayudarán a terminar más rápido. Sentarse y preocuparse tampoco ayudará. Y lo peor que puedes hacer es precipitarte. Resiste esa tentación a toda costa. Las prisas sólo te harán caer más en el agujero.

En lugar de eso, ve más despacio. Piensa en el problema. Traza un curso hacia el mejor resultado posible y luego conduce hacia ese resultado a un ritmo razonable y constante.

Comunicar

Haz saber a tu equipo y a tus superiores que tienes problemas. Cuéntales tus mejores planes para salir del problema. Pídeles su opinión y orientación. Evite crear sorpresas. Nada hace que la gente se enfade más y sea menos racional que las sorpresas. Las sorpresas multiplican la presión por diez.

Apóyate en tus disciplinas

Cuando las cosas se ponen difíciles, *confía en tus disciplinas*. La razón por la que *tienes disciplinas* es para guiarte en momentos de gran presión. Estos son los momentos en los que debes prestar especial atención a todas tus disciplinas. *No* es el momento de cuestionarlas o abandonarlas.

En lugar de buscar con pánico algo, cualquier cosa, que te ayude a terminar más rápido, sé más deliberado y dedícate a seguir tus disciplinas elegidas. Si sigues TDD, entonces escribe aún más pruebas de lo habitual. Si eres un refactorizador despiadado, entonces refactoriza aún más. Si mantienes tus funciones pequeñas, entonces mantenlas aún más pequeñas. La única manera de superar la olla a presión es confiar en lo que ya sabes que funciona: tus disciplinas.

Obtener ayuda

Emparejamiento Cuando el calor apriete, busque un asociado que esté dispuesto a programar en pareja con usted. Acabarás más rápido y con menos defectos. Tu compañero de pareja te ayudará a mantener tus disciplinas y evitará que entres en pánico.

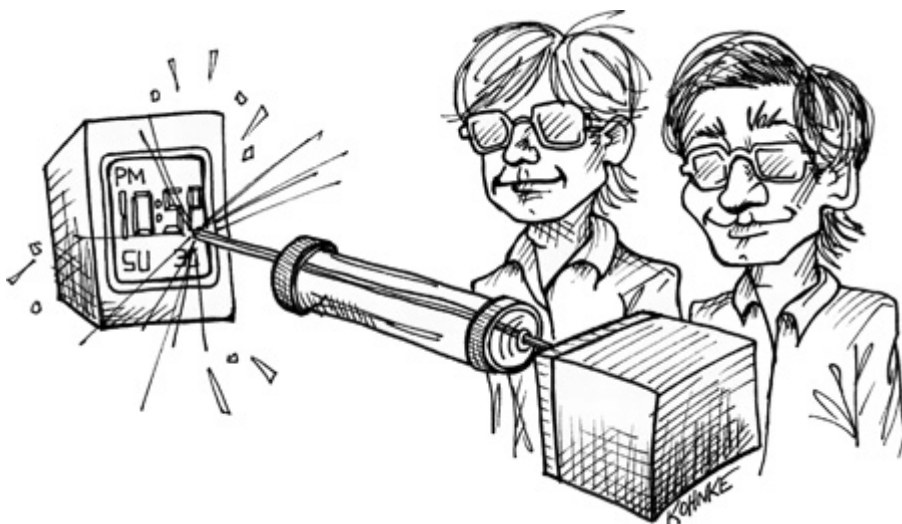
Tu compañero detectará cosas que a ti se te escapan, tendrá ideas útiles y te ayudará cuando pierdas la concentración.

Del mismo modo, cuando veas a otra persona que está bajo presión, ofrécete a formar pareja con ella. Ayúdala a salir del agujero en el que se encuentra.

Conclusión:

El truco para manejar la presión es evitarla cuando se puede, y capearla cuando no se puede. La evitas gestionando los compromisos, siguiendo tus disciplinas y manteniéndote limpio. La aguantas manteniendo la calma, comunicándote, siguiendo tu disciplina y buscando ayuda.

12. Colaboración



La mayor parte del software se crea en equipo. Los equipos son más eficaces cuando sus miembros colaboran profesionalmente. No es profesional ser un solitario o un recluso en un equipo.

En 1974 tenía 22 años. Mi matrimonio con mi maravillosa esposa, Ann Marie, apenas tenía seis meses. Todavía faltaba un año para el nacimiento de mi primera hija, Angela. Y yo trabajaba en una división de Teradyne conocida como Chicago Laser Systems.

A mi lado trabajaba mi compañero de instituto, Tim Conrad. Tim y yo habíamos hecho bastantes milagros en nuestra época. Construimos ordenadores juntos en su sótano. Construimos escaleras de Jacob en el mío. Nos enseñamos mutuamente a programar los PDP-8 y a conectar los circuitos integrados y los transistores en calculadoras funcionales.

Éramos programadores que trabajaban en un sistema que utilizaba el láser para recortar componentes electrónicos como resistencias y condensadores con una precisión extrema. Por ejemplo, recortamos el cristal del primer reloj digital, el Motorola Pulsar.

El ordenador que programábamos era el M365, un clon del PDP-8 de Teradyne. Escribíamos en lenguaje ensamblador y nuestros archivos fuente se guardaban en cartuchos de cinta magnética. Aunque podíamos editar en una pantalla, el proceso era bastante complicado, por lo que utilizábamos listados impresos para la mayor parte de nuestra lectura de código y la edición preliminar.

No teníamos ninguna facilidad para buscar en la base de código. No había forma de encontrar todos los lugares en los que se llamaba a una determinada función o se utilizaba una determinada constante. Como puedes imaginar, esto era un gran obstáculo.

Así que un día Tim y yo decidimos escribir un generador de referencias cruzadas. Este programa leería nuestras cintas de código fuente e imprimiría un listado de cada símbolo, junto con los números de archivo y de línea en los que se utilizaba ese símbolo.

El programa inicial era bastante sencillo de escribir. Simplemente leía la cinta fuente, analizaba la sintaxis del ensamblador, creaba una tabla de símbolos y añadía referencias a las entradas. Funcionaba muy bien, pero era terriblemente lento. Tardaba más de una hora en procesar nuestro Programa Operativo Maestro (el MOP).

La razón por la que era tan lento era que manteníamos la creciente tabla de símbolos en un único búfer de memoria. Cada vez que encontrábamos una nueva referencia la insertábamos en el búfer, moviendo el resto del búfer unos pocos bytes hacia abajo para hacer espacio.

Tim y yo no éramos expertos en estructuras de datos y algoritmos. Nunca habíamos oído hablar de las tablas hash o de las búsquedas binarias. No teníamos ni idea de cómo hacer un algoritmo rápido. Sólo sabíamos que lo

que estábamos haciendo era demasiado lento.

Así que probamos una cosa tras otra. Probamos a poner las referencias en listas enlazadas. Intentamos dejar huecos en el array y sólo hacer crecer el buffer cuando los huecos se llenaban. Intentamos crear listas enlazadas de huecos. Probamos todo tipo de ideas locas.

Nos poníamos ante la pizarra de nuestra oficina y dibujábamos diagramas de nuestras estructuras de datos y realizábamos cálculos para predecir el rendimiento. Cada día llegábamos a la oficina con otra idea nueva. Colaborábamos como locos.

Algunas de las cosas que probamos aumentaron el rendimiento. Otras lo ralentizaban. Era una locura. Fue entonces cuando descubrí lo difícil que es optimizar el software y lo poco intuitivo que es el proceso.

Al final conseguimos bajar el tiempo a menos de 15 minutos, lo que se acercaba mucho a lo que se tardaba simplemente en leer la cinta de origen. Así que estábamos satisfechos.

Los programadores frente a las personas

No nos convertimos en programadores porque nos guste trabajar con la gente. Por regla general, las relaciones interpersonales nos parecen desordenadas e imprevisibles. Nos gusta el comportamiento limpio y predecible de las máquinas que programamos. Somos

somos más felices cuando estamos solos en una habitación durante horas centrándonos profundamente en algún problema realmente interesante.

Vale, eso es una gran generalización y hay muchas excepciones. Hay muchos programadores a los que se les da bien trabajar con la gente y disfrutan con el reto. Pero la media del grupo sigue tendiendo en la dirección que he indicado. Nosotros, los programadores, disfrutamos de la leve privación sensorial y de la inmersión en el capullo de la *concentración*.

Los programadores frente a los empresarios

En los años setenta y ochenta, mientras trabajaba como programador en Teradyne, aprendí a ser *realmente* bueno depurando. Me encantaba el reto y me lanzaba a los problemas con vigor y entusiasmo. ¡Ningún error podía esconderse mucho tiempo de mí!

Cuando resolvía un fallo, era como ganar una victoria o matar al Jabberwock. Iba a ver a mi jefe, Ken Finder, con la hoja de Vorpal en la mano, y le describía apasionadamente lo *interesante que era el fallo*. Un día Ken finalmente estalló de frustración: "Los bichos no son interesantes. Los bichos sólo hay que arreglarlos".

Ese día aprendí algo. Es bueno sentir pasión por lo que hacemos. Pero

también es bueno no perder de vista los objetivos de la gente que te paga.

La primera responsabilidad del programador profesional es satisfacer las necesidades de su empleador. Eso significa colaborar con sus gerentes, analistas de negocio, probadores y otros miembros del equipo para *entendera fondo* los objetivos del negocio. Esto no significa que tenga que convertirse en un experto en negocios. Lo que *sí significa es que tienes que* entender por qué estás escribiendo el código que estás escribiendo, y cómo la empresa que te emplea se beneficiará de ello.

Lo peor que puede hacer un programador profesional es enterrarse felizmente en una tumba de tecnología mientras el negocio se estrellay arde a su alrededor. Su *trabajo* es mantener el negocio a flote.

Por eso, los programadores profesionales se toman el tiempo necesario para entender el negocio. Hablan con los usuarios sobre el software que utilizan. Hablan con el personal de ventas y marketing sobre los problemas que tienen. Hablan con sus directivos para entender los objetivos a corto y largo plazo del equipo.

En resumen, prestan atención al barco en el que navegan.

La única vez que me despidieron de un trabajo de programación fue en 1976. Por aquel entonces trabajaba para Outboard Marine Corp. Ayudabaa escribir un sistema de automatización de fábrica que utilizaba IBM System/7 para supervisar docenas de máquinas de fundición de aluminio en la planta.

Desde el punto de vista técnico, fue un trabajo desafiante y gratificante. La arquitectura del System/7 era fascinante, y el propio sistema de automatización de la fábrica era realmente interesante.

También teníamos un buen equipo. El jefe de equipo, John, era competente y estaba motivado. Mis dos compañeros de programación eran agradables y serviciales. Teníamos un laboratorio dedicado a nuestro proyecto y todos trabajamos en él. El socio comercial estaba comprometido y estaba en el laboratorio con nosotros. Nuestro director, Ralph, era competente, centrado y estaba al mando.

Todo debería haber ido bien. El problema era yo. Estaba bastante entusiasmado con el proyecto y la tecnología, pero a la edad de 24 años no podía preocuparme por el negocio ni por su estructura política interna.

Mi primer error fue el primer día. Me presenté sin corbata. Había llevado una en mi entrevista, y había visto que todos los demás llevaban corbata, pero no logré hacer la conexión. Así que en mi primer día, Ralph se dirigió a mí y me dijo claramente: "Aquí llevamos corbata".

No puedo decir lo mucho que me molestó eso. Me molestaba a un nivel profundo. Llevaba la corbata todos los días y la odiaba. ¿Pero por qué? Sabía en qué me metía. Sabía las convenciones que habían adoptado.

¿Por qué me molestaba tanto? Porque era un pequeño egoísta y narcisista. Simplemente no podía llegar al trabajo a tiempo. Y pensé que no importaba. Después de todo, estaba haciendo "un buen trabajo". Y era cierto, estaba haciendo un muy buen trabajo escribiendo mis programas. Era fácilmente el mejor programador técnico del equipo. Podía escribir código más rápido y mejor que los demás. Podía diagnosticar y resolver los problemas más rápidamente. *Sabía que* era valioso. Así que los tiempos y las fechas no me importaban mucho.

La decisión de despedirme se tomó un día cuando no me presenté a tiempo para un hito. Al parecer, John nos había dicho a todos que quería una demostración de las funciones en funcionamiento el próximo lunes. Estoy

seguro de que lo sabía, pero las fechas y las horas simplemente no eran importantes para mí.

Estábamos en desarrollo activo. El sistema no estaba en producción. No había ninguna razón para dejar el sistema funcionando cuando no había nadie en el laboratorio. Debí de ser el último en marcharme ese viernes y, al parecer, dejé el sistema sin funcionar. El hecho de que el lunes era importante simplemente no se me había metido en la cabeza.

Aquel lunes llegué con una hora de retraso y vi a todos reunidos con desgana en torno a un sistema que no funcionaba. John me preguntó: "¿Por qué no funciona el sistema hoy, Bob?". Mi respuesta: "No lo sé". Y me senté a depurarlo. Seguía sin tener ni idea de la demostración del lunes, pero me di cuenta por el lenguaje corporal de los demás de que algo iba mal. Entonces John se acercó y me susurró al oído: "¿Y si Stenberg hubiera decidido visitarnos?". Luego se alejó disgustado.

Stenberg era el vicepresidente encargado de la automatización. Hoy en día lo llamaríamos CIO. La pregunta no tenía sentido para mí. "¿Y qué?" pensé. "El sistema no está en producción, ¿cuál es el problema?".

Ese mismo día recibí mi primera carta de advertencia. Me decía que tenía que cambiar mi actitud inmediatamente o "el resultado será un despido rápido". Me quedé horrorizada.

Me tomé un tiempo para analizar mi comportamiento y empecé a darme cuenta de lo que había estado haciendo mal. Hablé con John y Ralph sobre ello. Me propuse dar un giro a mi persona y a mi trabajo.

¡Y lo hice! Dejé de llegar tarde. Empecé a prestar atención a la política interna. Empecé a entender por qué John estaba preocupado por Stenberg. Empecé a ver la mala situación en la que lo había puesto al no tener el sistema funcionando el lunes.

Pero era demasiado poco y demasiado tarde. La suerte estaba echada. Un mes después recibí una segunda carta de advertencia por un error trivial que cometí. Debería haberme dado cuenta en ese momento de que las

cartas eran una formalidad y que la decisión de despedirme ya estaba tomada. Pero estaba decidido a salvar la situación. Así que me esforcé aún más.

La reunión de despido llegó unas semanas después.

Ese día volví a casa con mi mujer embarazada de 22 años y tuve que decirle que me habían despedido. No es una experiencia que quiera repetir nunca.

Programadores contra programadores

Los programadores suelen tener dificultades para trabajar en estrecha colaboración con otros programadores. Esto conduce a algunos problemas realmente terribles.

Código de propiedad

Uno de los peores síntomas de un equipo disfuncional es cuando cada programador construye un muro alrededor de *su* código y se niega a que los demás programadores lo toquen. He estado en lugares donde los programadores ni siquiera dejaban que otros programadores *vieran* su código. Esta es una receta para el desastre.

Una vez fui consultor de una empresa que construía impresoras de alta gama. Estas máquinas tienen muchos componentes diferentes, como alimentadores, impresoras, apiladoras, grapadoras, cortadoras, etc. La empresa valoraba cada uno de estos dispositivos de forma diferente.

Los alimentadores eran más importantes que los apiladores, y nada era más importante que la impresora.

Cada programador trabajaba en *su* dispositivo. Uno escribía el código del alimentador y otro el de la grapadora. Cada uno de ellos guardaba su tecnología para sí mismo e impedía que nadie más tocara su código. La influencia política que ejercían estos programadores estaba directamente relacionada con el valor que la empresa daba al dispositivo. El programador que trabajaba en la impresora era inexpugnable.

Esto fue un desastre para la tecnología. Como consultor, pude ver que había una duplicación masiva en el código y que las interfaces entre los módulos estaban completamente sesgadas. Pero ningún argumento por mi parte pudo convencer a los programadores (o a la empresa) de que cambiaran su forma de actuar. Al fin y al cabo, sus revisiones salariales estaban ligadas a la importancia de los dispositivos que mantenían.

Propiedad colectiva

Es mucho mejor derribar todos los muros de propiedad del código y hacer que el equipo sea dueño de todo el código. Prefiero equipos en los que cualquier miembro del equipo pueda revisar cualquier módulo y hacer los cambios que considere oportunos. Quiero que el equipo sea el dueño del código, no los individuos.

Los desarrolladores profesionales no impiden que otros trabajen en el código. No construyen muros de propiedad alrededor del código. Más bien, trabajan con los demás en la mayor parte del sistema que pueden.

Aprenden de los demás trabajando con ellos en otras partes del sistema.

Emparejamiento

A muchos programadores no les gusta la idea de la programación por parejas. Me parece extraño, ya que la mayoría de los programadores *se* emparejan en caso de emergencia. ¿Por qué? Porque es claramente la forma más eficiente de resolver el problema. Se trata de volver al viejo adagio: dos cabezas piensan mejor que una. Pero si el emparejamiento es la forma más eficiente de resolver un problema en una emergencia, ¿por qué no es la forma más eficiente de resolver un problema en general?

No voy a citarte estudios, aunque hay algunos que se podrían citar. No te voy a contar anécdotas, aunque hay muchas que podría contar. Ni te voy a decir cuánto debes emparejar. Lo único que te voy a decir es que *los profesionales emparejan*. ¿Por qué? Porque al menos para algunos problemas es la forma más eficiente de resolverlos. Pero esa no es la única razón.

Los profesionales también se emparejan porque es la mejor manera de compartir conocimientos entre ellos. Los profesionales no crean silos de conocimiento. Más bien, aprenden las diferentes partes del sistema y del negocio emparejándose entre sí. Reconocen que, aunque todos los miembros del equipo tienen una posición que desempeñar, todos los miembros del equipo también deberían ser capaces de desempeñar otra posición en caso de necesidad.

Los profesionales se emparejan porque es la mejor manera de revisar el código. Ningún sistema debería estar formado por código que no haya sido revisado por otros programadores. Hay muchas formas de realizar revisiones de código; la mayoría de ellas son terriblemente ineficientes. La forma más eficiente y eficaz de revisar el código es colaborar en su escritura.

Cerebelo

Una mañana del año 2000, en pleno auge de las punto com, tomé el tren para llegar a Chicago. Al bajar del tren al andén me asaltó un enorme cartel publicitario que colgaba sobre las puertas de salida. El cartel era de una conocida empresa de software que estaba contratando programadores. Decía: *Ven a frotarte los cerebros con los mejores*.

Inmediatamente me sorprendió la gran estupidez de un cartel como ese. Estos pobres publicistas despistados intentaban atraer a una población altamente técnica, inteligente y con conocimientos de programación. Este es el tipo de gente que no sufre la estupidez particularmente bien. Los

publicistas intentaban evocar la imagen de compartir conocimientos con otras personas muy inteligentes. Por desgracia, se referían a una parte del cerebro, el

cerebelo, que se ocupa del control muscular fino, no de la inteligencia. Así que las mismas personas a las que trataban de atraer se burlaban de un errortan tonto.

Pero hay algo más que me intrigó de ese cartel. Me hizo pensar en un grupo de personas tratando de frotar cerebelos. Como el cerebelo está en la parte posterior del cerebro, la mejor manera de frotar cerebelos es de espaldas a los demás. Me imaginé a un equipo de programadores en cubículos, sentados en rincones de espaldas los unos a los otros, mirando las pantallas mientras llevaban auriculares.

Así *es* como se frotran los cerebros. Eso tampoco es un equipo.

Los profesionales trabajan *juntos*. No podéis trabajar juntos mientras estéis sentados en rincones con auriculares. Así que quiero que os sentéis alrededor de las mesas uno *frente* al otro. Quiero que podáis oler el miedo de los demás. Quiero que podáis escuchar los murmullos frustrados de alguien. Quiero una comunicación serendípica, tanto verbal como corporal. Quiero que os comunicuéis como una unidad.

Tal vez crea que trabaja mejor cuando trabaja solo. Eso puede ser cierto, pero no significa que el *equipo trabaje mejor cuando* usted trabaja solo. Y, de hecho, es muy poco probable que trabaje mejor cuando trabaja solo.

Hay ocasiones en las que trabajar solo es lo más adecuado. Hay veces en las que simplemente hay que reflexionar mucho sobre un problema. Hay veces en que la tarea es tan trivial que sería un desperdicio tener a otra persona trabajando contigo. Pero, en general, lo mejor es colaborar estrechamente con otras personas y emparejarse con ellas una gran parte del tiempo.

Conclusión:

Quizá no nos metimos en la programación para trabajar con la gente. Mala suerte para nosotros. La programación *consiste en trabajar con la gente*. Tenemos que trabajar con nuestra empresa, y tenemos que trabajar con los demás.

Lo sé, lo sé. ¿No sería estupendo que nos encerraran en una sala con seis pantallas enormes, una tubería T3, un conjunto paralelo de procesadores superrápidos, memoria RAM y disco ilimitados, y un suministro interminable de cola dietética y patatas fritas de maíz picantes? Por desgracia, no será así. Si realmente queremos pasar nuestros días programando, vamos a tener que aprender a hablar con la gente. ¹

13. Equipos y proyectos



¿Qué pasa si tienes muchos proyectos pequeños por hacer? ¿Cómo debe asignar esos proyectos a los programadores? ¿Y si tiene que realizar un proyecto realmente grande?

¿Se mezcla?

A lo largo de los años he sido consultor de varios bancos y compañías de seguros. Una cosa que parecen tener en común es la extraña forma de dividir los proyectos.

A menudo, un proyecto en un banco será un trabajo relativamente pequeño que requiere uno o dos programadores durante unas semanas. A menudo, este proyecto contará con un gestor de proyectos, que también está gestionando otros proyectos. Contará con un analista de negocio, que también está proporcionando los requisitos para otros proyectos. Contará con algunos programadores que también están trabajando en otros proyectos. Se asignará un probador o dos, que también estarán trabajando en otros proyectos.

¿Ves el patrón? El proyecto es tan pequeño que ninguna persona puede ser asignada a él a tiempo completo. Todo el mundo trabaja en el proyecto al 50%, o incluso al 25%.

Aquí hay una regla: No existe la mitad de una persona.

No tiene sentido decirle a un programador que dedique la mitad de su tiempo al proyecto A y el resto al proyecto B, especialmente cuando los dos

proyectos tienen dos directores de proyecto diferentes, analistas de negocio diferentes, programadores diferentes y probadores diferentes. ¿Cómo se puede llamar equipo a una monstruosidad como esa? Eso no es un equipo, es algo que salió de una batidora Waring.

El equipo de Gelled

Un equipo tarda en formarse. Los miembros del equipo empiezan a establecer relaciones. Aprenden a colaborar entre sí. Aprenden las peculiaridades, los puntos fuertes y los puntos débiles de cada uno. Con el tiempo, el equipo empieza a *cuajar*.

Hay algo verdaderamente mágico en un equipo compenetrado. Pueden hacer milagros. Se anticipan los unos a los otros, se cubren los unos a los otros, se apoyan los unos a los otros y se exigen lo mejor. Hacen que las cosas sucedan.

Un equipo bien formado suele estar formado por una docena de personas. Pueden ser hasta veinte o tan solo tres, pero el mejor número es probablemente alrededor de doce. El equipo debe estar compuesto por programadores, probadores y analistas. Y debe tener un director de proyecto.

La proporción entre programadores, probadores y analistas puede variar mucho, pero 2:1 es una buena cifra. Así, un equipo de doce personas bien compenetrado podría tener siete programadores, dos probadores, dos analistas y un director de proyecto.

Los analistas desarrollan los requisitos y escriben pruebas de aceptación automatizadas para ellos. Los probadores también escriben pruebas de aceptación automatizadas. La diferencia entre ambos es la perspectiva. Ambos escriben requisitos. Pero los analistas se centran en el valor del negocio; los probadores se centran en la corrección. Los analistas escriben los casos de camino feliz; los probadores se preocupan por lo que puede salir mal y escriben los casos de fallo y límite.

El director del proyecto hace un seguimiento del progreso del equipo y se asegura de que éste entienda los calendarios y las prioridades.

Uno de los miembros del equipo puede desempeñar un papel a tiempo parcial de entrenador, o maestro, con la responsabilidad de defender el proceso y las disciplinas del equipo. Actúa como conciencia del equipocuando éste tiene la tentación de salirse del proceso debido a la presión del calendario.

Fermentación

A un equipo como éste le lleva tiempo resolver sus diferencias, llegar a un acuerdo con los demás, y realmente compenetrarse. Puede llevar seis meses. Incluso puede llevar un año. Pero una vez que ocurre, es mágico. Un equipo bien compenetrado planifica juntos, resuelve los problemas juntos, se enfrenta a ellos juntos y *consigue hacer las cosas*.

Una vez que esto sucede, es absurdo separarlo sólo porque un proyecto llega a su fin. Lo mejor es mantener ese equipo unido y seguir alimentándolo con proyectos.

¿Qué fue primero, el equipo o el proyecto?

Los bancos y las compañías de seguros intentaron formar equipos en torno a los proyectos. Se trata de un enfoque insensato. Los equipos sencillamente no pueden cuajar. Los individuos sólo están en el proyecto durante un corto periodo de tiempo, y sólo durante un porcentaje de su tiempo, y por lo tanto nunca aprenden a tratar con los demás.

Las organizaciones de desarrollo profesional asignan los proyectos a los equipos ya formados, no forman equipos en torno a los proyectos. Un equipo cohesionado puede aceptar muchos proyectos simultáneamente y se repartirá el trabajo según sus propias opiniones, habilidades y capacidades. El equipo cohesionado conseguirá que los proyectos se lleven a cabo.

Pero, ¿cómo se gestiona eso?

Los equipos tienen velocidades.¹ La velocidad de un equipo es simplemente la cantidad de trabajo que puede realizar en un periodo de tiempo determinado. Algunos equipos miden su velocidad en *puntos* por semana, donde los puntos son una unidad de complejidad. Desglosan las características de cada proyecto en el que trabajan y las calculan en puntos. Luego miden cuántos puntos consiguen hacer por semana.

La velocidad es una medida estadística. Un equipo puede conseguir 38 puntos una semana, 42 la siguiente y 25 la siguiente. Con el tiempo, esto se compensará.

La dirección puede establecer objetivos para cada proyecto asignado a un equipo. Por ejemplo, si la velocidad media de un equipo es de 50 y tienen tres proyectos en los que están trabajando, la dirección puede pedir al equipo que divida su esfuerzo en 15, 15 y 20.

Aparte de tener un equipo engranado trabajando en sus proyectos, la ventaja de este esquema es que en caso de emergencia la empresa puede decir: "El proyecto B está en crisis; pon el 100% de tu esfuerzo en ese proyecto durante las próximas tres semanas".

Reasignar las prioridades con tanta rapidez es prácticamente imposible con los equipos que salieron de la batidora, pero los equipos unidos que trabajan en dos o tres proyectos simultáneamente pueden girar en un instante.

El dilema del propietario del proyecto

Una de las objeciones al enfoque que defiende es que los propietarios de los proyectos pierden cierta seguridad y poder. Los propietarios de proyectos que tienen un equipo dedicado a su proyecto pueden contar con

el esfuerzo de ese equipo. Saben que, dado que formar y disolver un equipo es una operación costosa, la empresa no se lo quitará por razones de corto plazo.

Por otro lado, si los proyectos se entregan a equipos agrupados, y si esos equipos se encargan de varios proyectos al mismo tiempo, la empresa es libre de cambiar las prioridades a su antojo. Esto puede hacer que el propietario del proyecto se sienta inseguro sobre el futuro. Los recursos de los que depende el propietario del proyecto pueden desaparecer de repente.

Francamente, prefiero esta última situación. La empresa no debería tener las manos atadas por la dificultad artificial de formar y disolver equipos. Si la empresa decide que un proyecto es más prioritario que otro, debería poder reasignar los recursos rápidamente. El propietario del proyecto tiene la responsabilidad de defender su proyecto.

Conclusión:

Los equipos son más difíciles de crear que los proyectos. Por lo tanto, es mejor formar equipos persistentes que pasen juntos de un proyecto a otro y que puedan encargarse de más de un proyecto a la vez. El objetivo al formar un equipo es darle el tiempo suficiente para que se consolide y luego mantenerlo unido como motor para realizar muchos proyectos.

Bibliografía

[RCM2003]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

[COHN2006]: Mike Cohn, *Agile Estimating and Planning*, Upper Saddle River, NJ: Prentice Hall, 2006.

14. Tutoría, aprendizaje y artesanía



Siempre me ha decepcionado la calidad de los licenciados en

informática. No es que los graduados no sean brillantes o no tengan talento, sino que no se les ha enseñado en qué consiste realmente la programación.

Grados de fracaso

Una vez entrevisté a una joven que estaba haciendo un máster en informática en una gran universidad. Solicitaba un puesto de becaria de verano. Le pedí que escribiera algo de código conmigo, y me dijo: "En realidad no escribo código".

Por favor, vuelva a leer el párrafo anterior, y luego pase de éste al siguiente.

Le pregunté qué cursos de programación había tomado para obtener su maestría. Me dijo que no había hecho ninguno.

Quizá quieras empezar por el principio del capítulo para asegurarte de que no has caído en un universo alternativo o acabas de despertar de un mal sueño.

Llegados a este punto, es posible que te preguntes cómo un estudiante de un programa de maestría en CS puede evitar un curso de programación. Yo me pregunté lo mismo en su momento. Todavía me lo pregunto hoy.

Por supuesto, esa es la más extrema de una serie de decepciones que he tenido al entrevistar a graduados. No todos los licenciados en informática son decepcionantes, ni mucho menos. Sin embargo, me he dado cuenta de que los que no lo son tienen algo en común: casi todos ellos *aprendieron a programar por su cuenta* antes de entrar en la universidad y siguieron aprendiendo por su cuenta a pesar de la universidad.

No me malinterpreten. Creo que es posible obtener una educación excelente en una universidad. Pero también creo que es posible pasar por el sistema y salir con un diploma y poco más.

Y hay otro problema. Incluso los mejores programas de licenciatura en ciencias de la computación no suelen preparar a los jóvenes graduados para lo que encontrarán en la industria. Esto no es una acusación a los programas de grado, sino que es la realidad de casi todas las disciplinas. Lo que se aprende en la escuela y lo que se encuentra en el trabajo son a menudo cosas muy diferentes.

Mentoring

¿Cómo aprendemos a programar? Permítanme contarles mi historia sobre la tutoría.

Digi-Comp I, mi primer ordenador

En 1964, mi madre me regaló un pequeño ordenador de plástico por mi duodécimo cumpleaños. Se llamaba Digi-Comp I.¹ Tenía tres flip-flops de

plástico y seis *puertas and de plástico*. Se podían conectar las salidas de los flip-flops a las entradas de las *compuertas*. También se podía conectar la salida de las *puertas and a las entradas de los* flip-flops. En resumen, esto le permitió crear una máquina de estado finito de tres bits.

El kit venía con un manual que te daba varios programas para ejecutar. Para programar la máquina, había que introducir pequeños tubos (segmentos cortos de pajitas de refresco) en unas clavijas que sobresalían de las chancas. El manual te decía exactamente dónde poner cada tubo, pero no lo que *hacían* los tubos. Esto me resultó muy frustrante.

Me quedé mirando la máquina durante horas y determiné cómo funcionaba en el nivel más bajo; pero no pude, por mi vida, averiguar cómo hacer que hiciera lo que yo quería. La última página del manual me decía que enviara un dólar y que ellos me devolverían un manual con la programación de la máquina. ²

Envié mi dólar y esperé con la impaciencia de un niño de doce años. El día que llegó el manual lo devoré. Era un sencillo tratado de álgebra booleana que abarcaba la factorización básica de las ecuaciones booleanas, las leyes asociativas y distributivas y el teorema de DeMorgan. El manual mostraba cómo expresar un problema en términos de una secuencia de ecuaciones booleanas. También describía cómo reducir esas ecuaciones para que cupieran en 6 *puertas y*.

Concebí mi primer programa. Todavía recuerdo el nombre: La Puerta Computarizada del Sr. Patternson. Escribí las ecuaciones, las reduje y las mapeé en los tubos y clavijas de la máquina. *¡Y funcionó!*

Al escribir esas tres palabras me han entrado escalofríos. Los mismos escalofríos que recorrieron a aquel niño de doce años hace casi medio siglo. Estaba enganchado. Mi vida ya no sería la misma.

¿Recuerda el momento en que su primer programa funcionó? ¿Cambió tu vida o te puso en un camino que no podías abandonar?

No lo descubrí todo por mí mismo. Fui *asesorado*. Algunas personas muy amables y muy hábiles (con las que tengo una enorme deuda de gratitud) se tomaron el tiempo de escribir un tratado sobre álgebra booleana que fuera accesible para un niño de doce años. Conectaron la teoría matemática con la pragmática del pequeño ordenador de plástico y me permitieron hacer que ese ordenador hiciera lo que yo quería.

Acabo de sacar mi copia de ese fatídico manual. Lo guardo en una bolsa con cierre. Sin embargo, los años han pasado factura amarilleando las páginas y volviéndolas quebradizas. Aun así, el poder de las palabras brilla en ellas. La elegancia de su descripción del álgebra booleana consumía tres escasas páginas. Su recorrido paso a paso por las ecuaciones de cada uno de los programas originales sigue siendo convincente. Fue una obra de maestría. Una obra que cambió la vida de al menos un joven. Sin embargo, dudo que

nunca conozca los nombres de los autores.

El ECP-18 en la escuela secundaria

A los quince años, cuando estaba en el primer año de instituto, me gustaba pasar el rato en el departamento de matemáticas. (Un día trajeron una máquina del tamaño de una sierra de mesa. Era un ordenador educativo hecho para los institutos, llamado ECP-18. Nuestra escuela recibió una demostración de dos semanas.

Me quedé en un segundo plano mientras los profesores y los técnicos hablaban. Esta máquina tenía una palabra de 15 bits (*¿qué es una palabra?*) y una memoria de tambor de 1024 palabras. (Para entonces ya sabía lo que era la memoria de tambor, pero sólo en concepto).

Cuando lo pusieron en marcha, emitió un sonido que recordaba al de un avión a reacción al despegar. Supuse que era el tambor que giraba. Una vez que se puso en marcha, fue relativamente silencioso.

La máquina era *preciosa*. Era esencialmente un escritorio de oficina con un maravilloso panel de control que sobresalía de la parte superior como el puente de un acorazado. El panel de control estaba adornado con filas de luces que también eran botones. Sentarse en ese escritorio era como sentarse en la silla del capitán Kirk.

Mientras veía a los técnicos pulsar esos botones, observé que se iluminaban al pulsarlos y que se podían volver a pulsar para apagarlos. También observé que había otros botones que pulsaban; botones con nombres como *depositar* y *ejecutar*.

Los botones de cada fila estaban agrupados en cinco grupos de tres. Mi Digi-Comp también era de tres bits, por lo que podía leer un dígito octal expresado en binario. No fue un gran salto darse cuenta de que se trataba de cinco dígitos octales.

Mientras los técnicos pulsaban los botones, podía oírles murmurar para sí mismos. Pulsaban 1, 5, 2, 0, 4, en la fila del *buffer de memoria* mientras se decían a sí mismos, "almacenar en 204". Pulsaban 1, 0, 2, 1, 3 y murmuraban: "cargar 213 en el *acumulador*". ¡Había una fila de botones llamada acumulador!

Diez minutos de eso y estaba bastante claro para mi mente de quince años que el 15 significaba *almacenar* y el 10 significaba *cargar*, que el acumulador era lo que se estaba almacenando o cargando, y que los otros números eran los números de una de las 1024 palabras del tambor. (¡Así que eso es una palabra!)

Poco a poco (no es un juego de palabras), mi ávida mente captó más y más códigos de instrucciones y conceptos. Cuando los técnicos se marcharon, ya sabía lo básico de cómo funcionaba aquella máquina.

Aquella tarde, durante una sala de estudio, me colé en el laboratorio de matemáticas y empecé a toquetear el ordenador. Hacía tiempo que había aprendido que es mejor pedir perdón que permiso. Introduje un pequeño programa que multiplicaba el acumulador por dos y sumaba uno. Introduje un 5 en el acumulador, ejecuté el programa y vi 13₈ en el acumulador. Tenía ¡funcionó!

Introduje varios otros programas sencillos como ese y todos funcionaron como estaba previsto. ¡Era el amo del universo!

Días después me di cuenta de lo estúpido, y afortunado, que había sido. Encontré una hoja de instrucciones por ahí en el laboratorio de matemáticas. En ella se mostraban todas las instrucciones y los op-codes, incluidos muchos que no había aprendido viendo a los técnicos. Me alegré de haber interpretado correctamente las que conocía y me emocioné con las demás. Sin embargo, una de las nuevas instrucciones era HLT. Sucedió que la instrucción de *parada* era una palabra de todos los ceros. Y resulta que yo había puesto una palabra de todos los ceros al final de cada uno de mis programas para poder cargarla en el acumulador y borrarla. El concepto de parada simplemente no se me había ocurrido. Me imaginé que el programa se detendría cuando terminara.

Recuerdo que en un momento dado me senté en el laboratorio de matemáticas para ver cómo uno de los profesores se esforzaba por hacer funcionar un programa. Intentaba escribir dos números en decimal en el teletipo adjunto y luego imprimir la suma. Cualquiera que haya intentado escribir un programa como éste en lenguaje de máquina en un miniordenador sabe que no es nada trivial. Hay que leer los caracteres, convertirlos en dígitos, luego en binario, sumarlos, volver a convertirlos en decimal y codificarlos de nuevo en caracteres. Y, créeme, ¡es *mucho* peor cuando estás introduciendo el programa en binario a través del panel frontal!

Observé cómo ponía un punto de interrupción en su programa y luego lo ejecutaba hasta que se detenía. (Este punto de interrupción primitivo le permitía examinar el contenido de los registros para ver qué había hecho su programa. Le recuerdo murmurando: "¡Vaya, qué rápido!". ¡Vaya, tengo noticias para él!

No tenía ni idea de cuál era su algoritmo. Ese tipo de programación era todavía mágica para mí. Y nunca me habló mientras yo observaba por encima de su hombro. De hecho, *nadie me hablaba* de este ordenador. Creo que me consideraban una molestia que había que ignorar, revoloteando por el laboratorio de matemáticas como una polilla.

Basta con decir que ni el alumno ni los profesores habían desarrollado un alto grado de habilidad social.

Al final consiguió que su programa funcionara. Era increíble verlo.

Tecleaba lentamente los dos números porque, a pesar de sus protestas anteriores, ese ordenador *no* era rápido (piense en leer palabras consecutivas de un tambor giratorio en 1967). Cuando pulsaba "return" después del segundo número, el ordenador parpadeaba ferozmente durante un rato y luego empezaba a imprimir el resultado. Tardó alrededor de un segundo por dígito. Imprimió todos los dígitos menos el último, parpadeó aún más ferozmente durante cinco segundos, y luego imprimió el último dígito y se detuvo.

¿Por qué esa pausa antes del último dígito? Nunca lo descubrí. Pero me hizo darme cuenta de que el planteamiento de un problema puede tener un efecto profundo en el usuario. Aunque el programa diera la respuesta correcta, *todavía* había algo que no funcionaba.

Esto era una tutoría. Ciertamente, no era el tipo de tutoría que yo podía esperar. Habría estado bien que uno de esos profesores me hubiera tomado bajo su ala y hubiera trabajado conmigo. Pero no importaba, porque yo los *observaba* y aprendía a un ritmo vertiginoso.

Tutoría no convencional

Les he contado estas dos historias porque describen dos tipos de tutoría muy diferentes, ninguno de los cuales es el que la palabra suele implicar. En el primer caso, aprendí de los autores de un manual muy bien escrito. En el segundo caso, aprendí observando a personas que intentaban activamente ignorarme. En ambos casos, los conocimientos adquiridos fueron profundos y fundamentales.

Por supuesto, también tuve otro tipo de mentores. Estaba el amable vecino que trabajaba en Teletype y que me trajo a casa una caja con 30 relés telefónicos para que jugara con ellos. Déjenme decirles que si le dan a un muchacho algunos relés y un transformador de tren eléctrico, ¡puede conquistar el mundo!

Hubo un vecino muy amable que era operador de radioaficionado y me enseñó a utilizar un multímetro (que enseguida rompí). El propietario de una tienda de material de oficina me permitió entrar y "jugar" con su carísima calculadora programable. La oficina de ventas de Digital Equipment Corporation me permitió entrar y "jugar" con su PDP-8 y PDP-10.

Luego estaba el gran Jim Carlin, un programador de BAL que me salvó de ser despedido de mi primer trabajo de programación ayudándome a depurar un programa Cobol que estaba muy por encima de mis posibilidades. Me enseñó a leer los volcados del núcleo y a formatear mi código con líneas en blanco, filas de estrellas y comentarios adecuados. Medio mi primer empujón hacia la artesanía. Lamento no haber podido devolverle el favor cuando el disgusto del jefe cayó sobre él un año

después.

Pero, francamente, eso es todo. No había muchos programadores senior a principios de los setenta. En todos los demás lugares donde trabajé, yo *era senior*. No había nadie que me ayudara a entender lo que era la verdadera programación profesional. No había ningún modelo que me enseñara cómo comportarme o qué valorar. Esas cosas tuve que aprenderlas por mí mismo, y no fue nada fácil.

Hard Knocks

Como ya he dicho antes, me despidieron de ese trabajo de automatización de la fábrica en 1976. Aunque era técnicamente muy competente, no había aprendido a prestar atención al negocio ni a los objetivos empresariales. Las fechas y los plazos no significaban nada para mí. Me olvidé de una gran demostración el lunes por la mañana, dejé el sistema estropeado el viernes y me presenté tarde el lunes con todo el mundo mirándome con enfado.

Mi jefe me envió una carta en la que me advertía de que tenía que hacer cambios inmediatamente o sería despedido. Esta fue una importante llamada de atención para mí. Reevalué mi vida y mi carrera y empecé a hacer algunos cambios significativos en mi comportamiento, algunos de los cuales has leído en este libro. Pero era demasiado poco y demasiado tarde. El impulso iba en la dirección equivocada y pequeñas cosas que antes no habrían importado se convirtieron en algo significativo. Así que, aunque me esforcé mucho, al final me acompañaron fuera del edificio.

No hace falta decir que no es divertido llevar ese tipo de noticias a casa con una esposa embarazada y una hija de dos años. Pero me levanté y me llevé algunas poderosas lecciones de vida a mi siguiente trabajo, que mantuve durante quince años y que constituyó la verdadera base de mi carrera actual.

Al final, sobreviví y prosperé. Pero tiene que haber una forma mejor. Habría sido mucho mejor para mí si hubiera tenido un verdadero mentor, alguien que me enseñara los entresijos. Alguien a quien hubiera podido observar mientras le ayudaba con pequeñas tareas, y que revisara y guiara mis primeros trabajos.

Alguien que me sirva de modelo y me enseñe los valores y los reflejos adecuados. Un sensei. Un maestro. Un mentor.

Aprendizaje

¿Qué hacen los médicos? ¿Cree que los hospitales contratan a licenciados en medicina y los meten en los quirófanos para hacer operaciones decorazón en su primer día de trabajo? Por supuesto que no.

La profesión médica ha desarrollado una disciplina de intensa tutoría envuelta en el ritual y lubricada con la tradición. La profesión médica

supervisa las universidades y se asegura de que los graduados tengan la mejor educación. Esa educación implica una *cantidad* aproximadamente *igual* de estudio en el aula y de actividad clínica en los hospitales trabajando con profesionales.

Al graduarse, y antes de poder obtener la licencia, los médicos recién acñados deben pasar un año de prácticas supervisadas y de formación llamado internado.

Se trata de una intensa formación en el puesto de trabajo. El becario está rodeado de modelos de conducta y profesores.

Una vez completado el internado, cada una de las especialidades médicas requiere de tres a cinco años más de práctica y formación supervisada, lo que se conoce como residencia. El residente adquiere confianza al asumir responsabilidades cada vez mayores sin dejar de estar rodeado y supervisado por médicos veteranos.

Muchas especialidades requieren de uno a tres años más de beca, en los que el estudiante continúa su formación especializada y su práctica supervisada.

Y *entonces podrán* presentarse a los exámenes y obtener la certificación del consejo.

Esta descripción de la profesión médica era un tanto idealizada, y probablemente muy inexacta. Pero el hecho es que cuando hay mucho en juego, no enviamos a los licenciados a una habitación, echamos carne de vez en cuando y esperamos que salgan cosas buenas. Entonces, ¿por qué hacemos esto en el software?

Es cierto que hay relativamente pocas muertes causadas por errores de software. Pero *hay* pérdidas monetarias significativas. Las empresas pierden enormes cantidades de dinero debido a la inadecuada formación de sus desarrolladores de software.

De alguna manera, la industria del desarrollo de software ha adquirido la idea de que los programadores son programadores, y que una vez que te has graduado puedes codificar. De hecho, no es nada raro que las empresas contraten a chavales recién salidos de la escuela, los formen en "equipos" y les pidan que construyan los sistemas más críticos. Es una locura.

Los pintores no lo hacen. Los fontaneros no lo hacen. Los electricistas no lo hacen. Diablos, ¡ni siquiera creo que los cocineros de comida rápida se comporten así! Me parece que las empresas que contratan a licenciados en Ciencias de la Computación deberían invertir más en su formación de lo que McDonalds invierte en sus camareros.

No nos engañemos, esto no importa. Hay mucho en juego. Nuestra civilización funciona con software. Es el software el que mueve y manipula

la información que impregna nuestra vida cotidiana. El software controla los motores, las transmisiones y los frenos de nuestros automóviles.

Mantiene nuestros saldos bancarios, nos envía nuestras facturas y acepta nuestros pagos. El software lava nuestra ropa y nos dice la hora. Pone imágenes en la televisión, envía nuestros mensajes de texto, hace nuestras llamadas telefónicas y nos entretiene cuando estamos aburridos. Está en todas partes.

Dado que confiamos a los desarrolladores de software todos los aspectos de nuestra vida, desde los más insignificantes hasta los más trascendentales, sugiero que un periodo razonable de formación y práctica supervisada no es inadecuado.

Aprendizaje de software

Entonces, ¿cómo *debe* la profesión del software inducir a los jóvenes graduados a las filas del profesionalismo? ¿Qué pasos deben seguir? ¿Qué retos deben afrontar? ¿Qué objetivos deben alcanzar? Vamos a trabajar hacia atrás.

Masters

Se trata de programadores que han liderado más de un proyecto de software importante. Normalmente tienen más de diez años de experiencia y han trabajado en varios tipos de sistemas, lenguajes y sistemas operativos. Saben cómo dirigir y coordinar varios equipos, son diseñadores y arquitectos competentes y pueden codificar en círculos alrededor de todos los demás sin sudar. Se les han ofrecido puestos de gestión, pero los han rechazado, han huido después de aceptarlos o los han integrado con su función principalmente técnica. Mantienen ese papel técnico leyendo, estudiando, practicando, haciendo y *enseñando*. Es a un maestro a quien la empresa asignará la responsabilidad técnica de un proyecto. Piensa en "Scotty".

Journeyman

Se trata de programadores formados, competentes y enérgicos. Durante este periodo de su carrera aprenderán a trabajar bien en equipo y a convertirse en líderes de equipo. Tienen conocimientos sobre la tecnología actual, pero suelen carecer de experiencia con muchos sistemas diversos. Suelen conocer un lenguaje, un sistema, una plataforma; pero están aprendiendo más.

Los niveles de experiencia varían mucho entre sus filas, pero la media es de unos

cinco años. En el lado más alejado de esa media tenemos a los maestros en ciernes; en el lado más cercano, a los aprendices recientes.

Los oficiales son supervisados por los maestros o por otros oficiales más veteranos. Los jóvenes oficiales rara vez tienen autonomía. Su trabajo está estrechamente supervisado. Su código es examinado. A medida que

adquieren experiencia, la autonomía aumenta. La supervisión se vuelve menos directa y más matizada. Con el tiempo, se convierte en una revisión por pares.

Aprendices/pasantes

Los licenciados comienzan su carrera como aprendices. Los aprendices no tienen autonomía. Son supervisados muy de cerca por los oficiales. Al principio no asumen ninguna tarea, sino que se limitan a prestar asistencia a los oficiales. Esta debe ser una época de programación por parejas muy intensa. Es cuando se aprenden y refuerzan las disciplinas. Es cuando se crean los cimientos de los valores.

Los oficiales son los maestros. Se aseguran de que los aprendices conozcan los principios de diseño, los patrones de diseño, las disciplinas y los rituales. Los maestros enseñan TDD, refactorización, estimación, etc. Asignan lecturas, ejercicios y prácticas a los aprendices; revisan su progreso.

El aprendizaje debe durar un año. En ese momento, si los oficiales están dispuestos a aceptar al aprendiz en sus filas, harán una recomendación a los maestros. Los maestros deben examinar al aprendiz tanto mediante una entrevista como revisando sus logros. Si los maestros están de acuerdo, el aprendiz se convierte en oficial.

La realidad

De nuevo, todo esto es idealizado e hipotético. Sin embargo, si cambias los nombres y entrecierras las palabras te darás cuenta de que no es tan diferente de cómo *esperamos* que funcionen las cosas ahora. Los graduados son supervisados por jóvenes jefes de equipo, que son supervisados por jefes de proyecto, y así sucesivamente. El problema es que, en la mayoría de los casos, esta supervisión *no es técnica*. En la mayoría de las empresas no hay ningún tipo de supervisión técnica. Los programadores obtienen aumentos y eventuales ascensos porque, bueno, eso es lo que se hace con los programadores.

La diferencia entre lo que hacemos hoy en día y mi programa idealizado de aprendizaje es el enfoque en la enseñanza técnica, la formación, la supervisión y la revisión.

La diferencia es la propia noción de que los valores profesionales y la pericia técnica deben ser enseñados, alimentados, mimados y cultivados. Lo que falta en nuestro actual enfoque estéril es la responsabilidad de los mayores de enseñar a los jóvenes.

Artesanía

Así que ahora estamos en condiciones de definir esta palabra: *artesanía*. ¿Qué es? Para entenderlo, veamos la palabra *artesano*. Esta palabra evoca

habilidad y calidad. Evoca la experiencia y la competencia. Un artesano es alguien que trabaja con rapidez, pero sin prisas, que ofrece presupuestos razonables y cumple los compromisos. Un artesano sabe cuándo decir que no, pero se esfuerza por decir que sí. Un artesano es un profesional.

La artesanía es la *mentalidad que* tienen los artesanos. La artesanía es un meme que contiene valores, disciplinas, técnicas, actitudes y respuestas.

Pero, ¿cómo adoptan los crápulas este meme? ¿Cómo consiguen esta mentalidad?

El meme de la artesanía se transmite de una persona a otra. Lo enseñan los mayores a los jóvenes. Se intercambia entre compañeros. Se observa y se reaprende, ya que los mayores observan a los jóvenes. La artesanía es un contagio, una especie de virus mental. Se contagia observando a los demás y dejando que el meme se arraigue.

Convencer a la gente

No se puede convencer a la gente de que sea artesana. No se les puede convencer de que acepten el meme de la artesanía. Los argumentos son ineficaces. Los datos son intrascendentes. Los estudios de casos no significan nada. La aceptación de un meme no es tanto una decisión racional como emocional. Esto es algo muy *humano*.

Entonces, ¿cómo conseguir que la gente adopte el meme de la artesanía? Recuerda que un meme es contagioso, pero sólo si se puede observar. Así que haz que el meme sea *observable*. Actúa como un modelo de conducta. Primero te conviertes en un artesano y dejas que se vea tu artesanía. Luego deja que el meme haga el resto del trabajo.

Conclusión:

La escuela puede enseñar la teoría de la programación informática. Pero la escuela no enseña ni puede enseñar la disciplina, la práctica y la habilidad de ser un artesano.

Esas cosas se adquieren a través de años de tutela personal y de mentores. Es hora de que los que estamos en la industria del software afrontemos el hecho de que guiar a la próxima hornada de desarrolladores de software hasta la madurez nos corresponderá a nosotros, no a las universidades. Es hora de que adoptemos un programa de aprendizaje, prácticas y orientación a largo plazo.

A. Herramientas



En 1978, trabajaba en Teradyne en el sistema de pruebas telefónicas que he descrito anteriormente. El sistema tenía unos 80KSLOC de ensamblador M365. Guardamos el código fuente en cintas.

Las cintas eran similares a esos cartuchos de cinta estéreo de 8 pistas que eran tan populares en los años 70. La cinta era un bucle sin fin y la unidad de cinta sólo podía moverse en una dirección. Los cartuchos venían en longitudes de 10', 25', 50' y 100'. Cuanto más larga era la cinta, más tiempo se tardaba en "rebobinar", ya que la unidad de cinta tenía que limitarse a moverla hacia delante hasta encontrar el "punto de carga". Unacinta de 100' tardaba cinco minutos en llegar al punto de carga, así que elegíamos las longitudes de nuestras cintas con criterio. ¹

Lógicamente, las cintas se subdividían en archivos. En una cinta podían caber tantos archivos como fuera posible. Para encontrar un archivo, se cargaba la cinta y se saltaba de uno en uno hasta encontrar el que se quería. Teníamos un listado del directorio del código fuente en la pared para saber cuántos archivos debíamos saltar antes de llegar al que queríamos.

Había una copia maestra de 100' de la cinta de código fuente en un estante del laboratorio. Estaba etiquetada como MASTER. Cuando queríamos editar un archivo, cargábamos la cinta de código fuente MASTER en una unidad y una 10' en blanco en otra. Pasábamos por la MASTER hasta que llegábamos al archivo que necesitábamos. Entonces copiamos ese archivo en la cinta de scratch. A continuación, "rebobinamos" ambas cintas y volvemos a colocar la MASTER en la estantería.

Había un listado especial de directorios del MASTER en un tablón de anuncios en el laboratorio. Una vez hechas las copias de los archivos que necesitábamos editar, poníamos una chincheta de color en el tablón junto al nombre de ese archivo. Así es como comprobábamos los archivos.

Editábamos las cintas en una pantalla. Nuestro editor de texto, el ED-402, era realmente muy bueno. Era muy parecido a vi. Leíamos una "página" de la cinta, editábamos el contenido y luego escribíamos esa página y leíamos la siguiente. Una página era normalmente 50 líneas de código. No podías

mirar adelante en la cinta para ver las páginas que venían, y no podías mirar atrás en la cinta para ver las páginas que habías editado. Así que usamos listados.

De hecho, marcábamos nuestros listados con todos los cambios que queríamos hacer, y *luego* editábamos los archivos de acuerdo con nuestras marcas. *Nadie* escribía ni modificaba código en el terminal. Eso era un suicidio.

Una vez realizados los cambios en todos los archivos que necesitábamos editar, los fusionábamos con el maestro para crear una cinta de trabajo. Esta es la cinta que usaríamos para ejecutar nuestras compilaciones y pruebas.

Una vez que hayamos terminado de probar y estemos seguros de que nuestros cambios han funcionado, miraremos la placa. Si no había nuevos pines en la placa, simplemente reetiquetábamos nuestra cinta de trabajo como MASTER y sacábamos nuestros pines de la placa. Si *había* nuevos pines en la placa, retirábamos nuestros pines y entregábamos nuestra cinta de trabajo a la persona cuyos pines seguían en la placa. Ellos tendrían que hacer la fusión.

Éramos tres, y cada uno tenía su propio color de pin, por lo que nos resultaba fácil saber quién tenía qué expedientes revisados. Y como todos trabajábamos en el mismo laboratorio y hablábamos con los demás todo el tiempo, teníamos el estado del tablero en la cabeza. Así que, por lo general, el tablero era redundante y a menudo no lo utilizábamos.

Herramientas

Hoy en día, los desarrolladores de software tienen una amplia gama de herramientas para elegir. La mayoría no merece la pena, pero hay algunas con las que todo desarrollador de software debe estar familiarizado. Este capítulo describe mi actual conjunto de herramientas personales. No he

hecho un estudio completo de todas las demás herramientas que existen, por lo que no debe considerarse una revisión exhaustiva. Esto es sólo lo que yo uso.

Control del código fuente

Cuando se trata del control del código fuente, las herramientas de códigoabierto suelen ser su mejor opción. ¿Por qué? Porque están escritas por desarrolladores, para desarrolladores. Las herramientas de código abiertoson las que los desarrolladores escriben para sí mismos cuando necesitan algo que funcione.

Hay bastantes sistemas de control de versiones "empresariales", comerciales y caros. Me parece que no se venden tanto a los desarrolladores como a los gestores, ejecutivos y "grupos de herramientas". Su lista de características es

impresionante y convincente. Por desgracia, a menudo no tienen las características que los desarrolladores realmente necesitan. La principal es la *velocidad*.

Un sistema de control de fuentes "empresarial"

Puede que su empresa haya invertido una pequeña fortuna en un sistema de control de código fuente "empresarial". Si es así, mis condolencias. Probablemente sea políticamente inapropiado que vayas por ahí diciendo a todo el mundo: "El tío Bob dice que no lo uses". Sin embargo, hay una solución fácil.

Puedes registrar tu código fuente en el sistema de la "empresa" al final de cada iteración (una vez cada dos semanas más o menos) y utilizar uno de los sistemas de código abierto en medio de cada iteración. Esto mantiene a todos contentos, no viola ninguna norma corporativa y mantiene tu productividad alta.

Bloqueo pesimista frente a optimista

El bloqueo pesimista parecía una buena idea en los años 80. Después de todo, la forma más sencilla de gestionar los problemas de actualización concurrente es serializarlos. Así que si *yo estoy* editando un archivo, mejor que no. De hecho, el sistema de pines de colores que usaba a finales de los 70 era una forma de bloqueo pesimista. Si había un pin en un archivo, no editabas ese archivo.

Por supuesto, el bloqueo pesimista tiene sus problemas. Si bloqueo un archivo y luego me voy de vacaciones, todos los demás que quieran editar ese archivo están atascados. De hecho, incluso si mantengo el archivo bloqueado durante uno o dos días, puedo retrasar a otros que necesiten hacer cambios.

Nuestras herramientas han mejorado mucho a la hora de fusionar archivos fuente que han sido editados simultáneamente. En realidad, es bastante

sorprendente cuando se piensa en ello. Las herramientas miran los dos archivos diferentes y el ancestro de esos dos archivos, y luego aplican múltiples estrategias para averiguar cómo integrar los cambios concurrentes. Y hacen un trabajo bastante bueno.

Así que la era del bloqueo pesimista ha terminado. Ya no necesitamos bloquear los archivos cuando los sacamos. De hecho, no nos molestamos en comprobar los archivos individuales en absoluto. Simplemente revisamos todo el sistema y editamos los archivos que necesitamos.

Cuando estemos listos para registrar nuestros cambios, realizaremos una operación de "actualización". Esta operación nos indica si alguien ha registrado el código antes que nosotros, fusiona automáticamente la mayoría de los cambios, encuentra conflictos y nos ayuda a realizar las fusiones restantes. A continuación, confirmamos el código fusionado.

Tendré mucho que decir sobre el papel que juegan las pruebas automatizadas y la integración continua con respecto a este proceso más adelante en este capítulo. Por el momento, digamos que *nunca* registramos el código que no pasa todas las pruebas. *Nunca jamás*.

CVS/SVN

El viejo sistema de control de código fuente es CVS. Era bueno para su época, pero se ha quedado un poco anticuado para los proyectos actuales. Aunque es muy bueno para manejar archivos y directorios individuales, no es muy bueno para.....renombrar archivos o borrar directorios. Y el áticobueno, cuanto menos se diga sobre eso, mejor.

Subversion, en cambio, funciona muy bien. Le permite revisar todo el sistema en una sola operación. Puedes actualizar, fusionar y confirmar fácilmente. Mientras no te metas en la ramificación, los sistemas SVN son bastante sencillos de gestionar.

Ramificación

Hasta 2008 evitaba todas las formas de bifurcación, excepto las más sencillas. Si un desarrollador creaba una rama, esa rama tenía que volver a la línea principal antes del final de la iteración. De hecho, era tan austero con las bifurcaciones que rara vez se hacían en los proyectos en los que participaba.

Si se utiliza SVN, sigo pensando que es una buena política. Sin embargo, hay algunas herramientas nuevas que cambian el juego por completo. Son los sistemas de control de fuentes *distribuidas*. `git` es mi favorito de los sistemas de control de fuentes distribuidas. Permíteme que te hable de él.

`git`

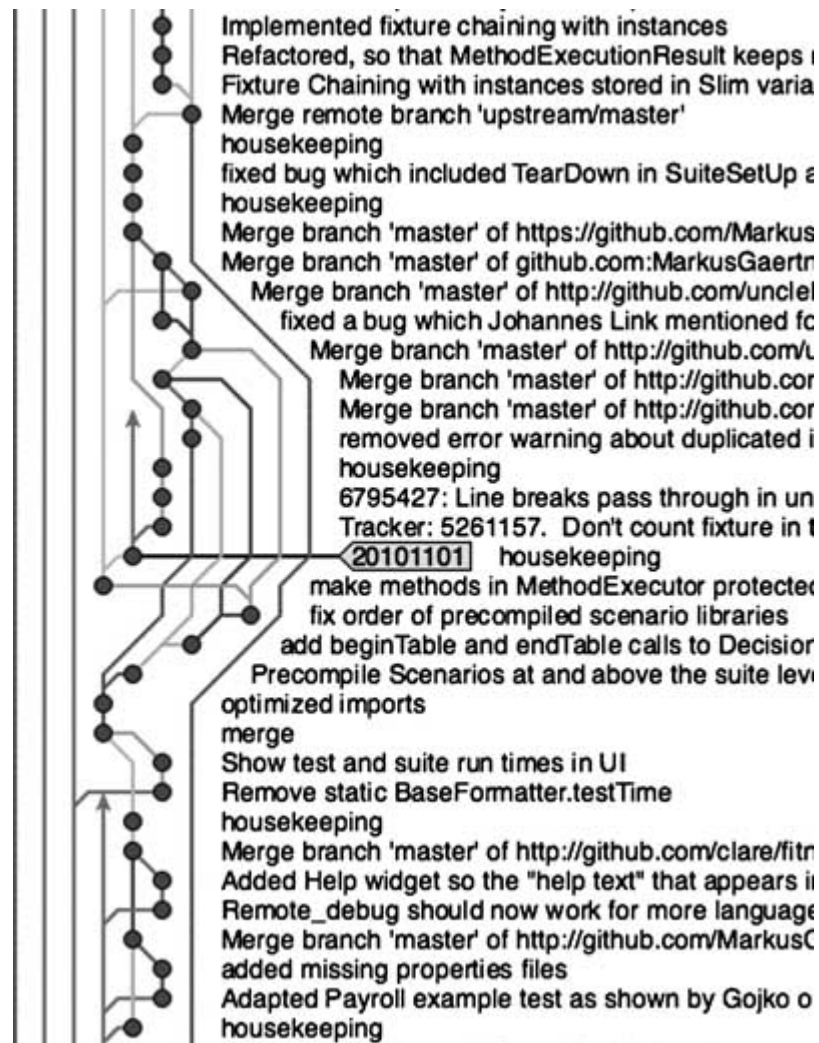
Empecé a utilizar `git` a finales de 2008, y desde entonces ha cambiado todo lo relacionado con mi forma de utilizar el control del código fuente. Entender por qué esta herramienta es un juego

cambiador está fuera del alcance de este libro. Pero la comparación de [la Figura A-1](#) con [la Figura A-2 debería](#) valer bastantes palabras que no voy a incluir aquí.

Figura A-1. FITNESSE bajo subversión

- More bug fixes
- Docs now say that Java 1.5 is required.
- Bug fix
- Many usability and behavioral improvements.
- Clean up
- Added PAGE_NAME and PAGE_PATH to pre-defined variables.
- Added ** to !path widget.
- link to the fixture gallery
- fixture gallery release 2.0 (2008-06-09) copied into the trunk wiki at
- Firefox compatability for invisible collapsible sections; removed .ce
- Updated documentation suite for all changes since last release.
- Enhancement to handle nulls in saved and recalled symbols. Adde
- Added a "Prune" Properties attribute to exclude a page and its chil
- Fixed type-o
- Added check for existing child page on rename.
- Added "Rename" link to Symbolic Links property section; renamed
- Adjusted page properties on recently added pages such that they c
- Enhanced Symbolic Links to allow all relative and absolute path for
- Cleaned up renamPageReponder a bit more.
- Cleaned Up PathParser names a bit. Pop -> RemoveNameFromE
- Cleaned up RenamePageResponder a bit. Fixed TestContentsHel
- updated usage message
- Fixed a bug wherein variables defined in a parent's preformatted bl
- Added explicit responder "getPage" to render a page in case query
- Tweaks to TOC help text.
- New property: Help text; TOCWidget has rollover balloon with new
- Redundant to the JUnit tests and elemental acceptance tests.
- Removed the last of the [acd] tags.
- !contents -f option enhancement to show suite filters in TOC list; fix
- TOC enhancements for properties (-p and PROPERTY_TOC and F
- 1) Render the tags on non-WikiWord links;
- Added http:// prefix to google.com for firewall transparency.
- Isolate query action from additional query arguments. For example
- Accommodate query strings like "?suite&suiteFilter=X"; prior logic v
- Cleaned up AliasLinkWidget a bit.

Figura A-2. FITNESSE bajo git



La figura A-1 muestra algunas semanas de desarrollo en el proyecto FITNESSE mientras estaba controlado por SVN. Puedes ver el efecto de mi austera regla de no bifurcarse. Sencillamente, no ramificamos. En su lugar, hicimos actualizaciones, fusiones y confirmaciones muy frecuentes a la línea principal.

La imagen A-2 muestra unas semanas de desarrollo en el mismo proyecto utilizando `git`. Como puedes ver, estamos ramificando y fusionando por todas partes. Esto no se debe a que haya relajado mi política de no bifurcarse, sino que simplemente se convirtió en la forma más obvia y conveniente de trabajar. Los desarrolladores individuales pueden hacer ramas de muy corta duración y luego fusionarlas entre sí a su antojo.

Fíjate también en que no puedes ver una verdadera línea principal. Eso es porque no *hay ninguna*. Cuando usas `git` no hay tal cosa como un repositorio central, o una línea principal. Cada desarrollador mantiene su propia copia de *toda* la historia

del proyecto en su máquina local. Entran y salen de esa copia local, y luego la fusionan con otras según sea necesario.

Es cierto que mantengo un repositorio dorado especial en el que introduzco todas las versiones y construcciones provisionales. Pero llamar a este repositorio la línea principal sería perder el punto. En

realidad es sólo una instantánea conveniente de toda la historia que cada desarrollador mantiene localmente.

Si no entiendes esto, no pasa nada. `git` es algo así como un rompecabezas al principio. Tienes que acostumbrarte a su funcionamiento. Pero te diré esto: `git`, y herramientas como ella, son el futuro del control del código fuente.

IDE/Editor

Como desarrolladores, pasamos la mayor parte de nuestro tiempo leyendo y editando código. Las herramientas que utilizamos para ello han cambiado mucho a lo largo de las décadas. Algunas son inmensamente potentes y otras han cambiado poco desde los años 70.

vi

Se podría pensar que los días en que se utilizaba `vi` como el principal editor de desarrollo ya han pasado. Hoy en día hay herramientas que superan con creces a `vi`, y a otros editores de texto sencillos como él. Pero la verdad es que `vi` ha disfrutado de un resurgimiento significativo en popularidad debido a su simplicidad, facilidad de uso, velocidad y flexibilidad. Puede que `Vi` no sea tan potente como Emacs, o Eclipse, pero sigue siendo un editor rápido y potente.

Dicho esto, ya no soy un usuario potente de `vi`. Hubo un día en que se me conocía como un "dios" del `vi`, pero esos días ya pasaron. Utilizo `vi` de vez en cuando si necesito hacer una edición rápida de un archivo de texto. Incluso lo he utilizado recientemente para hacer un cambio rápido en un archivo fuente de Java en un entorno remoto. Pero la cantidad de código real que he hecho en `vi` en los últimos diez años es muy pequeña.

Emacs

Emacs sigue siendo uno de los editores más potentes que existen, y probablemente lo seguirá siendo durante décadas. El modelo lisp interno lo garantiza. Como herramienta de edición de propósito general, ninguna otra se le acerca. Por otro lado, creo que Emacs no puede competir realmente con los IDE de propósito específico que ahora dominan. Editar código *no* es un trabajo de edición de propósito general.

En los años 90 yo era un fanático de Emacs. No me planteaba usar otra cosa. Los editores de apuntar y hacer clic de la época eran juguetes risibles que ningún desarrollador podía tomar en serio. Pero a principios de los años 00 conocí IntelliJ, mi IDE actual, y nunca he mirado atrás.

Eclipse/IntelliJ

Soy un usuario de IntelliJ. Me encanta. Lo uso para escribir Java, Ruby, Clojure, Scala, Javascript, y muchos otros. Esta herramienta fue escrita por programadores que entienden lo que los programadores necesitan al

escribir código. A lo largo de los años, rara vez me han decepcionado y casi siempre me han complacido.

Eclipse es similar en potencia y alcance a IntelliJ. Los dos están simplemente a pasos agigantados por encima de Emacs cuando se trata de editar Java. Hay otros IDEs en esta categoría, pero no los mencionaré aquí porque no tengo experiencia directa con ellos.

Las características que sitúan a estos IDEs por encima de herramientas como Emacs son las formas extremadamente potentes en las que te ayudan a manipular el código. En IntelliJ, por ejemplo, puedes extraer una superclase de una clase con un solo comando. Puedes renombrar variables, extraer métodos y convertir la herencia en composición, entre otras muchas grandes características.

Con estas herramientas, la edición de código ya no consiste en líneas y caracteres, sino en complejas manipulaciones. En lugar de pensar en los próximos caracteres y líneas que hay que escribir, se piensa en las próximas transformaciones que hay que hacer. En resumen, el modelo de programación es notablemente diferente y muy productivo.

Por supuesto, esta potencia tiene un coste. La curva de aprendizaje es alta, y el tiempo de preparación del proyecto no es insignificante.

Estas herramientas *no son* ligeras. Necesitan muchos recursos informáticos para funcionar.

TextMate

TextMate es potente y ligero. No puede hacer las maravillosas manipulaciones que pueden hacer IntelliJ y Eclipse. No tiene el potente motor y biblioteca lisp de Emacs. No tiene la velocidad y fluidez de vi. Por otro lado, la curva de aprendizaje es pequeña y su funcionamiento es intuitivo.

Yo uso TextMate de vez en cuando, especialmente para el C++ ocasional. Usaría Emacs para un proyecto grande de C++, pero estoy demasiado oxidado para molestarme con

Emacs para las pequeñas tareas de C++ que tengo.

Seguimiento de los problemas

En este momento estoy utilizando Pivotal Tracker. Es un sistema elegante y sencillo de utilizar. Encaja muy bien con el enfoque ágil/iterativo. Permite que todas las partes interesadas y los desarrolladores se comuniquen rápidamente. Estoy muy satisfecho con él.

Para proyectos muy pequeños, a veces he utilizado Lighthouse. Es muy rápido y fácil de configurar y utilizar. Pero no se acerca a la potencia de Tracker.

También he utilizado simplemente un wiki. Los wikis están bien para

proyectos internos. Te permiten establecer el esquema que quieras. No te obligan a seguir un determinado proceso o una estructura rígida. Son muy fáciles de entender y utilizar.

A veces, el mejor sistema de seguimiento de asuntos es un conjunto de tarjetas y un tablón de anuncios. El tablón de anuncios se divide en columnas como "Por hacer", "En curso" y "Hecho". Los desarrolladores se limitan a mover las tarjetas de una columna a la siguiente cuando sea oportuno. De hecho, este puede ser el sistema de seguimiento de problemas más común que utilizan los equipos ágiles hoy en día.

La recomendación que hago a los clientes es que empiecen con un sistema manual como el tablón de anuncios antes de comprar una herramienta de seguimiento. Una vez que domine el sistema manual, tendrá los conocimientos necesarios para elegir la herramienta adecuada. Y, de hecho, la elección adecuada puede ser simplemente seguir utilizando el sistema manual.

Recuento de insectos

Los equipos de desarrolladores necesitan, sin duda, una lista de problemas en los que trabajar. Estos problemas incluyen nuevas tareas y características, así como errores. Para cualquier equipo de tamaño razonable (de 5 a 12 desarrolladores) el tamaño de esa lista debería ser de docenas a cientos. *No miles*.

Si tienes miles de errores, algo va mal. Si tienes miles de características y/o tareas, algo va mal. En general, la lista de problemas debe ser relativamente pequeña, y por lo tanto manejable con una herramienta ligera como un wiki, Lighthouse, o Tracker.

Hay algunas herramientas comerciales que parecen ser bastante buenas. He visto que los clientes las utilizan pero no he tenido la oportunidad de trabajar con ellas

directamente. No me opongo a este tipo de herramientas, siempre que el número de problemas sea pequeño y manejable. Cuando las herramientas de seguimiento de incidencias se ven obligadas a seguir miles de ellas, la palabra "seguimiento" pierde su significado. Se convierten en "vertederos de problemas" (y a menudo también huelen a vertedero).

Construcción continua

Últimamente he estado usando Jenkins como mi motor de construcción continua. Es ligero, sencillo y casi no tiene curva de aprendizaje. Lo descargas, lo ejecutas, haces algunas configuraciones rápidas y sencillas, y ya estás en marcha.

Muy bonito.

Mi filosofía sobre la construcción continua es simple: Conéctalo a tu sistema de control de código fuente. Cada vez que alguien revise el código,

debería construirse automáticamente e informar del estado al equipo.

El equipo debe limitarse a mantener la construcción en funcionamiento en todo momento. Si la compilación falla, debe ser un evento de "parar las prensas" y el equipo debe reunirse para resolver rápidamente el problema. En ningún caso debe permitirse que el fallo persista durante un día o más.

Para el proyecto FITNESSE, hago que todos los desarrolladores ejecuten el script de construcción continua antes de confirmar. La compilación tarda menos de 5 minutos, por lo que no es onerosa. Si hay problemas, los desarrolladores los resuelven antes de la confirmación. Así que la compilación automática rara vez tiene problemas. La fuente más común de fallos en la compilación automática resulta ser problemas relacionados con el entorno, ya que mi entorno de compilación automática es bastante diferente de los entornos de desarrollo de los desarrolladores.

Herramientas de pruebas unitarias

Cada lenguaje tiene su propia herramienta de pruebas unitarias. Mis favoritas son JUNIT para Java, RSPEC para Ruby, NUNIT para .Net, Midje para Clojure, y CPPUTEST para C y C++.

Sea cual sea la herramienta de pruebas unitarias que elija, hay algunas características básicas que todas deberían soportar.

1. Debe ser rápido y fácil ejecutar las pruebas. Que esto se haga a través de plugins del IDE o de simples herramientas de línea de comandos es irrelevante, siempre y cuando los desarrolladores puedan ejecutar esas pruebas a su antojo. El gesto de ejecutar las pruebas debe ser trivial.

Por ejemplo, yo ejecuto mis pruebas CPPUTEST escribiendo `command-M` en TextMate. Tengo este comando configurado para ejecutar mi `makefile` que automáticamente ejecuta las pruebas e imprime un informe de una línea si todas las pruebas pasan. Tanto JUNIT como RSPEC son compatibles con INTELLIJ, así que todo lo que tengo que hacer es pulsar un botón. Para NUNIT, utilizo el plugin RESHARPER para que me dé el botón de prueba.

2. La herramienta debe darle una clara indicación visual de aprobado/desaprobado. No importa si se trata de una barra verde gráfica o de un mensaje de consola que diga "Todas las pruebas pasan". La cuestión es que debe poder decir que todas las pruebas han sido aprobadas rápidamente y sin ambigüedades. Si tiene que leer un informe de varias líneas, o peor aún, comparar la salida de dos archivos para saber si las pruebas pasaron, entonces ha fallado en este punto.
3. La herramienta debe darle una clara indicación visual del progreso. No importa si se trata de un medidor gráfico o de una cadena de

puntos, siempre y cuando puedas saber que se sigue avanzando y que las pruebas no se han estancado o abortado.

4. La herramienta debe evitar que los casos de prueba individuales se comuniquen entre sí. JUNIT lo hace creando una nueva instancia de la clase de prueba para cada método de prueba, evitando así que las pruebas utilicen variables de instancia para comunicarse entre sí. Otras herramientas ejecutan los métodos de prueba en orden aleatorio para que no se pueda depender de que una prueba preceda a otra. Sea cual sea el mecanismo, la herramienta debe ayudarle a mantener sus pruebas independientes entre sí. Las pruebas dependientes son una profunda trampa en la que no se quiere caer.
5. La herramienta debe facilitar la escritura de pruebas. JUNIT lo hace proporcionando una cómoda API para realizar aserciones. También utiliza la reflexión y los atributos de Java para distinguir las funciones de prueba de las funciones normales. Esto permite que un buen IDE identifique automáticamente todas sus pruebas, eliminando la molestia de cablear suites y crear listas de pruebas propensas a errores.

Herramientas de prueba de componentes

Estas herramientas sirven para probar componentes a nivel de API. Su función es asegurarse de que el comportamiento de un componente se especifica en un lenguaje que el personal de negocio y de control de calidad puede entender. De hecho, el caso ideal es que los analistas de negocio y el control de calidad puedan *escribir* esa especificación utilizando la herramienta.

La definición de *Hecho*

Más que cualquier otra herramienta, las herramientas de prueba de componentes son el medio por el que especificamos lo que significa "hecho". Cuando los analistas de negocio y el departamento de control de calidad colaboran para crear una especificación que define el comportamiento de un componente, y cuando esa especificación puede ejecutarse como un conjunto de pruebas que pasan o fallan, entonces *hecho* adquiere un significado muy inequívoco: "Todas las pruebas pasan".

FitNesse

Mi herramienta de pruebas de componentes favorita es FITNESSE. Yo escribí una gran parte de ella, y soy el principal comitente. Así que es mi bebé.

FITNESSE es un sistema basado en wiki que permite a los analistas de negocio y a los especialistas en control de calidad escribir pruebas en un formato tabular muy sencillo. Estas tablas son similares a las de Parnas, tanto en su forma como en su intención. Las pruebas pueden ensamblarse rápidamente en suites, y las suites pueden ejecutarse a capricho.

FITNESSE está escrito en Java, pero puede probar sistemas en cualquier lenguaje porque se comunica con un sistema de pruebas subyacente que puede estar escrito en cualquier lenguaje. Los lenguajes compatibles son Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi y otros.

Hay dos sistemas de pruebas que subyacen a FITNESSE: Fit y Slim. Fit fue escrito por Ward Cunningham y fue la inspiración original de FITNESSE y sus similares. Slim es un sistema de pruebas mucho más sencillo y portátil que es el preferido por los usuarios de FITNESSE en la actualidad.

Otras herramientas

Conozco otras herramientas que podrían clasificarse como herramientas de prueba de componentes.

- RobotFX es una herramienta desarrollada por ingenieros de Nokia. Utiliza un formato tabular similar al de FITNESSE, pero no se basa en un wiki. La herramienta simplemente se ejecuta en archivos planos preparados con Excel o similar. La herramienta está escrita en Python, pero puede probar sistemas en cualquier lenguaje utilizando los puentes adecuados.
- Green Pepper es una herramienta comercial que tiene varias similitudes con FITNESSE. Se basa en el popular wiki confluence.
- Cucumber es una herramienta de texto plano impulsada por un motor Ruby, pero capaz de probar muchas plataformas diferentes. El lenguaje de Cucumber es el popular estilo Given/When/Then.
- JBehave es similar a Cucumber y es el padre lógico de Cucumber. Está escrito en Java.

Herramientas de pruebas de integración

Las herramientas de pruebas de componentes también pueden utilizarse para muchas pruebas de integración, pero son menos apropiadas para las pruebas que se realizan a través de la interfaz de usuario.

En general, no queremos conducir muchas pruebas a través de la interfaz de usuario porque las interfaces de usuario son notoriamente volátiles. Esa volatilidad hace que las pruebas que pasan por la UI sean muy frágiles.

Dicho esto, hay algunas pruebas que *deben pasar por* la interfaz de usuario, sobre todo las pruebas *de* la interfaz de usuario. Además, algunas pruebas de extremo a extremo deben pasar por todo el sistema ensamblado, incluida la interfaz de usuario.

Las herramientas que más me gustan para las pruebas de UI son Selenium y Watir.

UML/MDA

A principios de los años 90 tenía muchas esperanzas de que la industria de

las herramientas CASE provocara un cambio radical en la forma de trabajar de los desarrolladores de software. Cuando miré hacia adelante desde aquellos días embriagadores, pensé que a estas alturas todo el mundo codificaría en diagramas a un nivel de abstracción superior y que el código textual sería cosa del pasado.

Me equivoqué. No sólo no se ha cumplido este sueño, sino que todos los intentos de avanzar en esa dirección han fracasado estrepitosamente. No es que no haya herramientas y sistemas que demuestren el potencial; es que esas herramientas simplemente no hacen realidad el sueño, y casi nadie parece querer usarlas.

El sueño era que los desarrolladores de software pudieran dejar atrás los detalles del código textual y crear sistemas en un lenguaje de alto nivel de diagramas.

De hecho, según el sueño, podríamos no necesitar programadores en absoluto. Los arquitectos podrían crear sistemas completos a partir de diagramas UML. Los motores, amplios y geniales, que no se preocupan por la situación de los simples programadores, transformarían esos diagramas en código ejecutable. Ese era el gran sueño de la Arquitectura Dirigida por Modelos (MDA).

Por desgracia, este gran sueño tiene un pequeño fallo. MDA asume que el problema es el código. Pero el código *no* es el problema. Nunca ha sido el problema. El problema es *el detalle*.

Los detalles

Los programadores son gestores de detalles. Eso es lo que hacemos. Especificamos el comportamiento de los sistemas hasta el más mínimo detalle. Para ello, utilizamos lenguajes textuales (código) porque son extraordinariamente cómodos (pensemos en el inglés, por ejemplo).

¿Qué tipo de detalles gestionamos?

¿Conoces la diferencia entre los dos caracteres `\n` y `\r`? El primero, `\n`, es un avance de línea. El segundo, `\r`, es un retorno de carro. ¿Qué es un carro?

En los años 60 y principios de los 70, uno de los dispositivos de salida más comunes para los ordenadores era el teletipo. El modelo ASR33 ²era el más común.

Este dispositivo consistía en un cabezal de impresión que podía imprimir diez caracteres por segundo. El cabezal de impresión estaba compuesto por un pequeño cilindro con los caracteres en relieve. El cilindro giraba y se elevaba de manera que el carácter correcto quedaba frente al papel, y entonces un pequeño martillo golpeaba el cilindro contra el papel. Había una cinta de tinta entre el cilindro y el papel, y la tinta se transfería al papel

con la forma del carácter.

El cabezal de impresión iba sobre un carro. Con cada carácter, el carro se desplazaba un espacio hacia la derecha, llevándose el cabezal de impresión. Cuando el carro llegaba al final de la línea de 72 caracteres, había que devolverlo explícitamente enviando los caracteres de retorno de carro ($\text{\r} = 0 \times 0D$), de lo contrario el cabezal de impresión seguiría imprimiendo caracteres en la columna 72, convirtiéndola en un desagradable rectángulo negro.

Por supuesto, eso no era suficiente. Devolver el carro no elevaba el papel a la siguiente línea. Si devolvía el carro y no enviaba un carácter de avance de línea ($\text{\n} = 0 \times 0A$), la nueva línea se imprimía encima de la anterior.

Por lo tanto, para un teletipo ASR33 la secuencia de fin de línea era `"\r\n"`. En realidad, había que tener cuidado con eso ya que el carro podía tardar más de 100ms en volver. Si se enviaba `"\n\r"`, el siguiente carácter podía imprimirse al volver el carro, creando así un carácter borroso en medio de la línea. Para estar seguros, a menudo rellenamos la secuencia de final de línea con uno o dos caracteres de borrado $\text{\u001B} (0 \times FF)$.

En los años 70, cuando los teletipos empezaron a dejar de utilizarse, los sistemas operativos como UNIX acortaron la secuencia de final de línea a simplemente `"\n"`. Sin embargo, otros sistemas operativos, como el DOS, siguieron utilizando la convención `"\r\n"`.

¿Cuándo fue la última vez que tuviste que lidiar con archivos de texto que utilizan la convención "equivocada"? Me enfrento a este problema al menos una vez al año. Dos archivos fuente idénticos no se comparan, y no generan sumas de comprobación idénticas, porque utilizan diferentes finales de línea. Los editores de texto no envuelven las palabras correctamente, o hacen doble espacio en el texto porque los finales de línea son "incorrectos". Los programas que no esperan líneas en blanco se bloquean porque interpretan `"\r\n"` como dos líneas. Algunos programas reconocen `"\r\n"` pero no reconocen `"\n\r"`. Y así sucesivamente.

*Eso es lo que quiero decir con *detalle*. ¡Intenta codificar la horrible lógica para ordenar los finales de línea en UML!*

Sin esperanza, no hay cambio

La esperanza del movimiento MDA era que se podría eliminar una gran cantidad de detalles utilizando diagramas en lugar de código. Hasta ahora, esa esperanza ha resultado ser desalentadora. Resulta que no hay tantos detalles adicionales en el código que puedan eliminarse con imágenes. Es más, las imágenes contienen sus propios detalles accidentales. Las imágenes tienen su propia gramática, sintaxis, reglas y restricciones. Así que, al final, la diferencia de detalles es un lavado.

La esperanza de MDA era que los diagramas demostraran estar a un nivel

de abstracción más alto que el código, al igual que Java está a un nivel más alto que el ensamblador. Pero, de nuevo, esa esperanza ha resultado ser errónea hasta ahora. La diferencia en el nivel de abstracción es mínima en el mejor de los casos.

Y, por último, digamos que un día alguien inventa un lenguaje de diagramas realmente útil. No serán los arquitectos quienes dibujen esos diagramas, sino los programadores. Los diagramas se convertirán sencillamente en el nuevo código, y los programadores serán necesarios para *dibujar* ese código porque, al final, se trata de detalles, y son los programadores los que gestionan esos detalles.

Conclusión:

Desde que empecé a programar, las herramientas de software son cada vez más potentes y abundantes. Mi kit de herramientas actual es un simple subconjunto de esa colección. Utilizo `git` para el control del código fuente, Tracker para la gestión de problemas, Jenkins para la construcción continua, IntelliJ como mi IDE, XUnit para las pruebas y FITNESSE para las pruebas de componentes.

Mi máquina es un Macbook Pro, 2.8Ghz Intel Core i7, con una pantalla mate de 17 pulgadas, 8GB de ram, 512GB SSD, con dos pantallas adicionales.

Índice

A

Pruebas de aceptación

automatizadas, [103-105](#)

comunicación y, [103](#)

integración continua y, [110-111](#)

definición de, [100](#)

el papel del desarrollador en, [106-107](#) el trabajo extra y, [105](#)

GUTs y, [109-111](#)

negociación y, [107-108](#) agresión pasiva y, [107-108](#) momento de, [105-106](#)

pruebas unitarias y, [108-109](#) escritores de, [105-106](#)

Roles adversos, [26-29](#)

Estimación de afinidad, [146-147](#)

Ambigüedad, en los requisitos, [98-](#)

[100](#) Disculpas, [12](#)
Aprendices, [183](#)
Aprendizaje, [180-184](#)
Argumentos, en las reuniones, [126-127](#)
Arrogancia, [22](#)
Pruebas de aceptación automatizadas, [103-105](#)
Garantía de calidad automatizada, [14](#)
Evitar, [131](#)

B

Callejones sin salida, [131-132](#)
Bossavit, Laurent, [89](#)
Juego de bolos, [89](#)
Ramificación, [191](#)
Recuento de errores, [197](#)
Objetivos empresariales, [160](#)

C

Cafeína, [128](#)
Certeza,
código [80](#)
control, [189-194](#)
propiedad, [163](#)
3 AM, [59-60](#)
preocupación, [60-61](#)
Coding Dojo, [89-93](#)
Colaboración, [20](#), [157-166](#)
Propiedad colectiva, [163-164](#)
Compromiso(s), [47-52](#)
control y, [50](#)
disciplina y, [53-56](#)
estimación y, [138](#)
expectativas y, [51](#)
identificar, [49-50](#)
implícito, [140-141](#)
importancia de, [138](#)
falta de, [48-49](#)
presión y, [152](#)
Comunicación
pruebas de aceptación y,

[103](#) presión y, [154](#)
de los requisitos, [95-100](#)
Pruebas de los componentes
en la estrategia de pruebas, [116-117](#)
herramientas para, [199-](#)
[200](#) Conflicto, en las
reuniones, [126-127](#)
Construcción continua, [197-198](#)
Integración continua, [110-111](#)
Aprendizaje continuo, [19](#)
Control, compromiso y, [50](#)
Valor, [81-82](#)
Artesanía, [184](#)____
Aportación creativa, [65-66](#), [129](#)
Disciplina de crisis, [153](#) Pepino,
[200](#)
Cliente, identificación con, [21](#)
CVS, [191](#)
Tiempo de ciclo, en el desarrollo dirigido por pruebas, [78](#)

D

Plazos

entrega falsa y, [73](#)
esperanza y, [71](#)
horas extras y, [72](#)
la prisa y, [71-72](#)
Depuración, [66-69](#) Tasa
de inyección de
defectos, [81](#) Reuniones
de demostración, [126](#)
Diseño, desarrollo dirigido por pruebas y, [82-83](#)
Patrones de diseño, [18](#)
Principios de diseño, [18](#)
Detalles, [201-203](#)
Desarrollo. *ver* desarrollo dirigido por pruebas
(TDD) Desacuerdos, en reuniones, [126-127](#)
Disciplina
compromiso y, [53-56](#)
crisis, [153](#)
Desenganche, [70](#)

Documentación, [82](#)

Dominio, conocimiento de, [21](#)

"Hecho", definición, [73](#), [100-103](#)

Enfoque de "no hacer daño", [11-16](#)

a la función, [11-14](#)

a la estructura, [14-16](#)

Conducción, [70](#)

E

Eclipse, [195-196](#)

Emacs, [195](#)

Empleador(
es)

identificación con, [21](#)

programadores vs. , [159-162](#)

Estimación

afinidad, [146-147](#)

ansiedad, [98](#)

compromiso y, [138](#)

definición de, [138-139](#)

ley de los grandes números y,
[147](#) nominal, [142](#)

optimista, [141-142](#)

PERT y, [141-144](#)

pesimista, [142](#)

probabilidad y, [139](#) de
las tareas, [144-147](#)

trivariado, [147](#)

Expectativas, compromiso y, [51](#)

Experiencia, ampliación, [93](#)

F

Fracaso, grados de, [174](#)

Falso parto, [73](#)

FitNesse, [199-200](#)

Flexibilidad, [15](#)

Zona de flujo, [62-](#)

[64](#) Dedos voladores,
[145](#)

Focus, [127-129](#)

Función, en el enfoque de "no hacer daño", [11-14](#)

G

Gaillot, Emmanuel, [89](#)

Equipo de gelatina, [168-170](#)

Git, [191-194](#)

Objetivos, [26-29](#), [124](#)

Interfaces gráficas de usuario (GUT),
[109-111](#) Green Pepper, [200](#)

Grenning, James, [145](#)

GUTs, [109-111](#)

H

Golpes duros, [179-180](#)

Ayuda, [73-76](#)

dando, [74](#)

la tutoría y, [75-76](#)

presión y, [154-155](#)

recibir, [74-75](#)

"Esperanza", [48](#)

Esperanza, plazos y, [71](#)

Humildad, [22](#)

I

TDE/editor, [194](#)

Identificación, con el empleador/cliente, [21](#)

Compromisos implícitos, [140-141](#)

Input, creativo, [65-66](#), [129](#)

Integración, continua, [110-111](#)

Pruebas de integración

en la estrategia de pruebas, [117-118](#)

herramientas para, [200-201](#)

IntelliJ, [195-196](#)

Interns, [183](#)

Interrupciones, [63-64](#)

Tissue tracking, [196-197](#)

Iteration planning meetings, [125](#)

Reuniones retrospectivas de iteración, [126](#)

J

JBehave, [200](#)
Oficiales, [182-183](#)

K

Kata, [90-91](#)
Conocimient

o

- de dominio, [21](#)
- mínimo, [18](#)
- ética del trabajo y, [17-19](#)

L

Retrasos, [71-73](#)
Ley de los grandes números,
[147](#) Aprendizaje, ética del
trabajo y, [19](#) "Vamos", [48](#)
Lindstrom, Lowell, [146](#)
Cierre, [190](#)

M

Pruebas exploratorias manuales, en la estrategia de
pruebas, [118-119](#) Maestros, [182](#)
MDA, [201-203](#)
Reuniones

- agenda en, [124](#)
- argumentos y desacuerdos en, [126-127](#)
 - en declive, [123](#)
- demonstración, [126](#)
- objetivos en, [124](#)
- planificación de la iteración, [125](#)
- iteración retrospectiva, [126](#)
- saliendo, [124](#)
- de pie, [125](#)
- gestión del tiempo y, [122-127](#)

Tutoría, [20-21](#), [75-76](#), [174-180](#)
Refactorización despiadada, [15](#)
Líos, [132-133](#), [152](#)
Métodos, [18](#)
Arquitectura dirigida por modelos (MDA), [201-203](#) Enfoque muscular, [129](#)

Música, [63](#)

N

"Necesidad", [48](#)

Negociación, pruebas de aceptación y, [107-108](#)

Estimación nominal, [142](#)

No profesional, [8](#)

O

Código abierto, [93](#)

Estimación optimista, [141-142](#)

Cierre optimista, [190](#)

Resultados, lo mejor posible, [26-29](#)

Horas extras, [72](#)

Código propio, [163](#)

Propiedad, colectiva, [163-164](#)

P

El ritmo, [69-70](#)

Emparejamiento, [64](#), [154-155](#), [164](#)

 Pánico, [153-154](#)

Pasión, [160](#)

Agresión pasiva, [34-36](#), [107-108](#)

Personas, programadores vs. , [159-164](#)

Problemas personales, [60-61](#)

PERT (Técnica de Evaluación y Revisión de Programas), [141-144](#)

Estimación pesimista, [142](#)

Cierre pesimista, [190](#)

Actividad física, [129](#)

Planificación del póquer,

[145-146](#) Práctica

 antecedentes, [86-89](#)

 ética, [93](#)

 experiencia y, [93](#) tiempo de

 entrega y, [88-89](#) ética de

 trabajo y, [19-20](#)

Precisión, prematura, en los requisitos, [97-98](#)

Preparación, [58-61](#)

Presión

 evitar, [151-153](#)

limpieza y, [152](#)
compromisos y, [152](#)
comunicación y, [154](#)
manipulación, [153-155](#)
ayuda y, [154-155](#)
desórdenes y, [152](#)
pánico y, [153-154](#)

Inversión de la prioridad, [131](#)

Probabilidad, [139](#)

Profesionalidad, [8](#)

programadores

empleadores vs. , [159-162](#)

personas vs. , [159-164](#)

programadores vs. ,

[163](#)Propuesta, proyecto, [37-](#)
[38](#)

Q

Garantía de calidad

(QA)

automatizada, [14](#)

como cazabichos, [12](#)

como caracterizadores, [114-115](#)

ideal de, como no encontrar problemas, [114-](#)
[115](#) problemas encontrados por, [12-13](#)

como especificadores, [114](#)

como miembro del equipo, [114](#)

R

Randori, [92-93](#)

La lectura, como aportación

creativa, [65](#) Recarga, [128-](#)
[129](#)

Reputación,

[11](#) Requisitos

comunicación de, [95-100](#)

ansiedad de estimación y, [98](#)

ambigüedad tardía en, [98-100](#)

precisión prematura en, [97-98](#)

incertidumbre y, [97-98](#)

Responsabilidad, [8-11](#)
disculpas y, [12](#)
enfoque de "no hacer daño" y, [11-16](#) función y, [11-14](#)
estructura y, [14-16](#) ética
del trabajo y, [16-22](#)

RobotFX, [200](#)

Roles, adversarios, [26-29](#)

Correr, [40-41](#), [71-72](#)

S

Santana, Carlos, [89](#)

"Debería", [48](#)

Ducha, [70](#)

Simplicidad, [40](#)

Dormir, [128](#)

Control del código fuente, [189-194](#) Estacas, [29-30](#)

Reuniones de pie, [125](#)

Estructura

en el enfoque de "no hacer daño",
[14-16](#) flexibilidad y, [15](#)
importancia de, [14](#)

SVN, [191-194](#)

Pruebas del sistema, en la estrategia de pruebas, [118](#)

T

Estimación de tareas, [144-147](#)

Equipos y trabajo en equipo,
[30-36](#)

gelificado, [168-170](#)

gestión de, [170](#)

agresión pasiva y, [34-36](#)

preservando, [169](#)

proyecto iniciado, [169-170](#)

dilema del propietario del proyecto con,
[170-171](#) intentar y, [32-34](#)

velocidad de, [170](#)

Desarrollo dirigido por pruebas

(TDD), [80](#) beneficios de, [80-](#)



[83](#)

la certeza y, [80](#)
valor y, [81-82](#) tiempo
de ciclo en, [78](#) debut
de, [77-78](#)

tasa de inyección de
defectos y, [81](#) definición
de, [13-14](#)
diseño y, [82-83](#)
documentación y, [82](#)
interrupciones y, [64](#)
tres leyes de, [79-80](#)
lo que no es, [83-84](#)

Prueba

aceptación
automatizado, [103-105](#)
comunicación y, [103](#)
integración continua y, [110-111](#)
definición de, [100](#)
el papel del desarrollador en,
[106-107](#) el trabajo extra y, [105](#)
GUTs y, [109-111](#)
negociación y, [107-108](#) agresión
pasiva y, [107-108](#) momento de,
[105-106](#)
pruebas unitarias y,
[108-109](#) escritores de,
[105-106](#)
pirámide de automatización, [115-119](#)
componente
en la estrategia de pruebas,
[116-117](#) herramientas para,
[199-200](#)
importancia de, [13-14](#)
integración

en la estrategia de pruebas,
[117-118](#) herramientas para,
[200-201](#)
manual exploratorio, [118-119](#)
estructura
y, [15](#) sistema,
unidad [118](#)
pruebas de aceptación y, [108-](#)
[109](#) en la estrategia de pruebas,
[116](#)
herramientas para, [198-199](#)

TextMate, [196](#)

Thomas, Dave, [90](#)

3 Código AM, [59-60](#)

Tiempo, depuración,

[69](#) Gestión del tiempo

evasión y, [131](#)_____

callejones sin salida y, [131-](#)

[132](#) ejemplos de, [122](#)

enfoque y, [127-129](#)

reuniones y, [122-127](#) líos

y, [132-133](#) inversión de

prioridad y, [131](#)

recarga y, [128-129](#) técnica

de "tomates" para,

[130](#) Cansancio, [59-60](#)

Técnica de gestión del tiempo "Tomates",

[130](#) Herramientas, [189](#)

Estimaciones trivariadas, [147](#) Tiempo

de entrega, práctica y, [88-89](#)

U

UML, [201](#)

Incertidumbre, requisitos y, [97-98](#) Tutoría no
convencional. [179](#) Véase también tutoría Pruebas
unitarias

pruebas de aceptación y, [108-](#)

[109](#) en la estrategia de

pruebas, [116](#)

herramientas para, [198-199](#)

V

Vi, [194](#)

W

Alejándose, [70](#)

Wasa, [91-92](#)

Delphi de banda ancha, [144-147](#)

"Deseo", [48](#)

Ética del trabajo, [16-22](#)

colaboración y, [20](#)

aprendizaje continuo y, [19](#)

conocimiento y, [17-19](#)

la tutoría y, [20-21](#)

práctica y, [19-20](#)

Código de la preocupación, [60-61](#)

Bloqueo del escritor, [64-66](#)

y

"Sí"

coste de, [36-40](#)

aprender a decir, [52-56](#)

Notas a pie de página

Introducción previa a los requisitos

¹ Que no cunda el pánico.

² Término técnico de origen desconocido.

³ EasyCoder era el ensamblador para el ordenador Honeywell H200, que era similar a Autocoder para el ordenador TBM 1401.

Capítulo 1

¹ ¡Esperemos que tenga una buena política de errores y omisiones!

² [\[PPP2001\]](#)

Capítulo 2

¹ Como Foghorn Leghorn: "Siempre tengo las plumas numeradas para una emergencia así".

² <http://raptureinvenice.com/?p=63>

³ Con la posible excepción del empleador directo de John, aunque T

apostarí­a que ellos también perdieron.

Capítulo 3

- ¹ No es que la patente valiera dinero para mí. T la había vendido a Teradyne por un dólar, según mi contrato de trabajo (y T no se quedó con el dólar).

Capítulo 4

- ¹ [\[Martin09\]](#)
- ² [\[Martin03\]](#)
- ³ No conozco ninguna disciplina que sea tan eficaz como el TDD, pero quizás tú sí.
- ⁴ Hay mucho más sobre esto en el capítulo sobre estimaciones.
- ⁵ Véase [el capítulo 7](#), "Pruebas de aceptación".
- ⁶ Esto es mucho más cierto para los hombres que para las mujeres. T tuvo una maravillosa conversación con @desi (Desi McAdam, fundadora de DevChix) sobre lo que motiva a las mujeres programadoras. T le dijo que cuando conseguía que un programa funcionara, era como matar a la gran bestia. Me dijo que para ella y para otras mujeres con las que había hablado, el acto de escribir código era un acto de creación nutritiva.

Capítulo 5

- ¹ Desde mi punto de vista, un niño es cualquier persona menor de 35 años. Durante mis veinte años, T pasó una gran cantidad de tiempo escribiendo pequeños juegos tontos en lenguajes interpretados. Escribió juegos de guerra espacial, juegos de aventura, juegos de carreras de caballos, juegos de serpientes, juegos de azar, de todo.
- ² <http://fitnesse.org>
- ³ El noventa por ciento es un mínimo. En realidad, la cifra es mayor que eso. La cantidad exacta es difícil de calcular porque las herramientas de cobertura no pueden ver el código que se ejecuta en procesos externos o en bloques de captura.
- ⁴ http://www.objectmentor.com/omSolutions/agile_customers.html
- ⁵ [\[Maximilien\]](#), [\[George2003\]](#), [\[Janzen2005\]](#), [\[Nagappan2008\]](#)

Capítulo 6

- ¹ El hecho de que algunos programadores esperen a las compilaciones es trágico e indica una falta de cuidado. En el mundo actual, los tiempos de compilación deberían medirse en

segundos, no en minutos, y desde luego no en horas.

² Se trata de una técnica que Rich Hickey denomina HDD (Hammock-Driven Development).

³ Esta se ha convertido en una kata muy popular, y una búsqueda en Google encontrará muchas instancias de la misma. El original está aquí:

<http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>.

⁴ Utilizamos el prefijo "Pragmático" para desambiguarlo del "Gran" Dave Thomas de OTT.

⁵ <http://codekata.pragprog.com>

⁶ <http://codingdojo.org/>

⁷ <http://katas.softwarecraftsmanship.org/?p=71>

⁸ <http://c2.com/cgi/wiki?PairProgrammingPingPongPattern>

Capítulo 7

¹ XPTmmersion

3, Mayo, 2000. <http://c2.com/cgi/wiki?TomsTalkAtXpTmmersionThree>

Capítulo 8

¹ http://www.satisfice.com/articles/what_is_et.shtml²

[COHN09] pp. 311-312

Capítulo 9

¹ El maná es un producto común en los juegos de fantasía y de rol como Dragones y Mazmorras. Cada jugador tiene una cierta cantidad de maná, que es una sustancia mágica que se gasta cada vez que un jugador lanza un hechizo mágico. Cuanto más potente sea el hechizo, más maná consumirá el jugador. El maná se recarga a un ritmo diario lento y fijo. Así que es fácil agotarlo en unas pocas sesiones de lanzamiento de hechizos.

² <http://www.pomodoratechnique.com/>

Capítulo 10

¹ La Ley de Murphy sostiene que si algo puede salir mal, saldrá mal.

² El número exacto para una distribución normal es 1:769, o el 0,13%, o 3 sigma. Las probabilidades de uno entre mil son probablemente seguras.

³ El PERT supone que esto se aproxima a una distribución beta. Esto

tiene sentido, ya que la duración mínima de una tarea suele ser mucho más cierta que la máxima. [[McConnell2006](#)] Fig. 1-3.

⁴ Si no sabes lo que es una desviación estándar, deberías encontrar un buen resumen de probabilidad y estadística. El concepto no es difícil de entender y te servirá mucho.

⁵ [[Boehm81](#)]

⁶ [[Grenning2002](#)]

⁷ <http://store.mountaingoatsoftware.com/products/planning-poker-cards>

⁸ http://en.wikipedia.org/wiki/Law_of_large_numbers

Capítulo 12

¹ Una referencia a la última palabra de la película *Soylent Green*.

Capítulo 13

¹ [[RCM2003](#)] pp. 20-22; [[COHN2006](#)] Busque en el índice muchas referencias excelentes sobre la velocidad.

Capítulo 14

¹ Hay muchos sitios web que ofrecen simuladores de este pequeño y estimulante ordenador.

² Todavía tengo este manual. Ocupa un lugar de honor en una de mis estanterías.

Apéndice A

¹ Estas cintas sólo podían moverse en una dirección. Así que cuando se producía un error de lectura, no había forma de que la unidad de cinta retrocediera y volviera a leer. Había que dejar lo que se estaba haciendo, enviar la cinta de vuelta al punto de carga y volver a empezar. Esto ocurría dos o tres veces al día. Los errores de escritura también eran muy comunes, y la unidad no tenía forma de detectarlos. Así que siempre escribíamos las cintas de dos en dos y comprobábamos los pares cuando terminábamos. Si una de las cintas estaba mal, hacíamos inmediatamente una copia. Si ambas estaban mal, lo que era muy poco frecuente, volvíamos a empezar toda la operación. Así era la vida en los años 70.

² http://en.wikipedia.org/wiki/ASR-33_Teletype

³ Los caracteres de borrado eran muy útiles para editar cintas de papel. Por convención, los caracteres de borrado se ignoraban. Su código, 0×FF, significaba que todos los agujeros de esa fila de la cinta estaban perforados. Esto significaba que cualquier carácter podía convertirse

en un rubout sobreperforándolo.

Por lo tanto, si cometieras un error mientras escribes tu programa, podrías retroceder el golpe y pulsar *rubout*, para luego continuar escribiendo.