

Ingeniería de software en Google

Lecciones sobre programación aprendidas a lo largo del tiempo



Marcombo

Organizadas por Titus Winters,
Tom Manshreck y Hyrum Wright

Ingeniería de software en Google

**Lecciones sobre programación aprendidas
a lo largo del tiempo**

**Organizadas por Titus Winters,
Tom Manshreck y Hyrum Wright**

Ingeniería de software en Google

Lecciones sobre programación aprendidas
a lo largo del tiempo

Organizadas por Titus Winters,
Tom Manshreck y Hyrum Wright



Edición original publicada en inglés por O'Reilly con el título *Software Engineering at Google*, ISBN 978-1-492-08279-8 © 2020 Google, LLC.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Título de la edición en español:

Ingeniería de software en Google

Primera edición en español, 2022

© 2022 MARCOMBO, S.L.

www.marcombo.com

Diseño de portada: Karen Montgomery

Ilustración: Rebecca Demarest

Traducción: Francisco Martínez Carreño

Corrección: Mónica Muñoz Marinero

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra solo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley. La presente publicación contiene la opinión del autor y tiene el objetivo de informar de forma precisa y concisa. La elaboración del contenido, aunque se ha trabajado de forma escrupulosa, no puede comportar una responsabilidad específica para el autor ni el editor de los posibles errores o imprecisiones que pudiera contener la presente obra.

ISBN: 978-84-267-3487-7

Producción del ePub: booqlab

Contenidos

Prólogo

Prefacio

Parte I. Tesis

1. ¿Qué es la ingeniería de *software*?

Tiempo y cambio

Ley de Hyrum

Ejemplo: ordenación *hash*

¿Por qué no aspirar a que «nada cambie»?

Escala y eficiencia

Políticas que no escalan

Políticas que escalan adecuadamente

Ejemplo: actualización del compilador

Desplazamiento hacia la izquierda

Contrapartidas y costes

Ejemplo: rotuladores

Aportaciones a la toma de decisiones

Ejemplo: compilaciones distribuidas

Ejemplo: decidir entre tiempo y escala

Revisar decisiones, cometer errores

Ingeniería de *software* frente a programación

Conclusión

Resumen

Parte II. Cultura

2. Cómo trabajar bien en equipo

Ayúdeme a ocultar mi código

El mito del genio

La ocultación se considera perjudicial

 Detección temprana

 El factor autobús

 Ritmo del progreso

 En resumen, no se esconda

Todo es cuestión de equipo

 Los tres pilares de la interacción social

 ¿Por qué importan estos pilares?

 Humildad, respeto y confianza en la práctica

Cultura *post mortem* sin sentimiento de culpa

 Ser Googley

Conclusión

Resumen

3. Compartir conocimientos

Desafíos para el aprendizaje

Filosofía

Preparación del escenario: seguridad psicológica

 Tutoría

 Seguridad psicológica en grupos grandes

Aumente sus conocimientos

 Haga preguntas

 Comprenda el contexto

Escalado de las preguntas: pregunte a la comunidad

 Chats de grupo

 Listas de correo electrónico

YAQS: plataforma de preguntas y respuestas

Escalado del conocimiento: siempre hay algo que enseñar

Horas de oficina

Charlas y clases de tecnología

Documentación

Código

Escalado de los conocimientos de la organización

Cultivar la cultura de compartir el conocimiento

Establecimiento de fuentes canónicas de información

Manténgase al día

Legibilidad: tutorías estandarizadas a través de la revisión del código

¿Qué es el proceso de legibilidad?

¿Por qué someterse a este proceso?

Conclusión

Resumen

4. Ingeniería para la equidad

Los prejuicios son la norma

Comprensión de la necesidad de la diversidad

Desarrollo de capacidades multiculturales

Hacer que se pueda procesar la diversidad

Rechazo de enfoques singulares

Desafío a los procesos establecidos

Valores frente a resultados

Mantener la curiosidad, seguir adelante

Conclusión

Resumen

5. Cómo liderar un equipo

Gerentes y líderes en tecnología (y ambos)

El gerente de ingeniería

El líder en tecnología

El gerente líder de tecnología

Pasar de la función de colaborador individual a la función de liderazgo

Lo único que hay que temer es..., bueno, todo

Liderazgo de servicio

El gerente de ingeniería

«Gerente» es una palabra de cuatro letras

El gerente de ingeniería en la actualidad

Antipatrones

Antipatrón: contratar a personas fáciles de manejar

Antipatrón: ignorar a las personas de bajo rendimiento

Antipatrón: ignorar los problemas de carácter personal

Antipatrón: ser amigo de todos

Antipatrón: comprometer el listón de contratación

Antipatrón: tratar al equipo como si fueran niños

Patrones positivos

Perder el ego

Ser un maestro zen

Ser catalizador

Eliminar obstáculos

Ser maestro y mentor

Establecer metas claras

Ser honesto

Rastrear la satisfacción

La pregunta inesperada

Otros consejos y trucos

Las personas somos como las plantas

Motivación intrínseca frente a motivación extrínseca

Conclusión

Resumen

6. Liderazgo a escala

Siempre hay que decidir

La parábola del aeroplano

Identificación de las orejeras

Señalar las contrapartidas clave

Decidir y, después, repetir

Siempre hay que dejar solo al equipo

Su misión: formar a un equipo «autónomo»

División del espacio del problema

Siempre hay que mantenerse escalando

El ciclo del éxito

Lo importante frente a lo urgente

Aprender a dejar caer pelotas al suelo

Proteja su energía

Conclusión

Resumen

7. Medición de la productividad de la ingeniería

¿Por qué debemos medir la productividad de la ingeniería?

Triaje: ¿vale la pena medirlo?

Selección de métricas significativas con objetivos y señales

Objetivos

Señales

Métricas

Uso de datos para validar métricas

Actuar y realizar un seguimiento de los resultados

Conclusión

Resumen

Parte III. Procesos

8. Guías de estilo y normas

¿Por qué tenemos normas?

Creación de normas

Principios rectores

Guía de estilo

Cambio de las normas

El proceso

Árbitros de estilo

Excepciones

Orientación

Aplicación de las normas

Comprobadores de errores

Formateadores de código

Conclusión

Resumen

9. Revisión del código

Flujo de revisión del código

Cómo funciona la revisión de código en Google

Beneficios de la revisión de código

Corrección de código

Comprensión de código

Coherencia del código

Beneficios psicológicos y culturales

Compartir conocimientos

Mejores prácticas de la revisión de código

Sea cortés y profesional

Escriba cambios pequeños

Escriba descripciones de los cambios que tengan calidad

Mantenga al mínimo el número de revisores

Automatizar donde sea posible

Tipos de revisiones de código

Revisiones del código *greenfield*

Cambios de comportamiento, mejoras y optimizaciones

Corrección de errores y reversiones

Refactorizaciones y cambios a gran escala

Conclusión

Resumen

10. Documentación

¿Qué calificar como «documentación»?

¿Por qué es necesaria la documentación?

La documentación es como el código

Conozca a su audiencia

Tipos de audiencias

Tipos de documentación

Documentación de referencia

Documentos de diseño

Tutoriales

Documentación conceptual

Páginas de destino

Revisiones de la documentación

Filosofía de la documentación

QUIÉN, QUÉ, CUÁNDO, DÓNDE y POR QUÉ

El principio, la parte central y el final

Parámetros de la documentación de calidad

Documentación obsoleta

¿Cuándo necesita a escritores técnicos?

Conclusión

Resumen

11. Descripción general de las pruebas

¿Por qué escribimos pruebas?

- La historia de Google Web Server

- Pruebas al ritmo del desarrollo moderno

- Escribir, ejecutar, reaccionar

- Ventajas de probar el código

Diseño de un conjunto de pruebas

- Tamaño de las pruebas

- Alcance de las pruebas

- Regla de Beyoncé

- Nota sobre la cobertura del código

Pruebas a escala de Google

- Dificultades de un gran conjunto de pruebas

Historial de pruebas en Google

- Clases de orientación

- Pruebas certificadas

- Pruebas en los aseos

- La cultura de pruebas actualmente

Límites de las pruebas automatizadas

Conclusión

Resumen

12. Pruebas unitarias

Importancia del mantenimiento

Prevención de pruebas frágiles

- Esfuércese en lograr pruebas que no cambien

- Pruebas a través de API públicas

- Comprobar el estado, no las interacciones

Escriba pruebas claras

- Haga que las pruebas sean completas y concisas

- Pruebe los comportamientos, no los métodos

- No ponga la lógica en las pruebas

- Escriba mensajes de error claros
- Pruebas y uso compartido de código: DAMP, no DRY
 - Valores compartidos
 - Configuración compartida
 - Helpers* compartidos y validación
 - Definición de infraestructura de pruebas
- Conclusión
- Resumen

13. Dobles de pruebas

- El impacto de los dobles de pruebas en el desarrollo del *software*
- Dobles de pruebas en Google
- Conceptos básicos
 - Un ejemplo de dobles de pruebas
 - Costuras
 - Marcos de trabajo de simulación
- Técnicas para utilizar los dobles de pruebas
 - Simulación
 - Stubbing*
 - Pruebas de interacción
- Implementaciones reales
 - Preferencia de las implementaciones reales a las aisladas
 - Cómo decidir cuándo utilizar una implementación real
- Simulación
 - ¿Por qué son importantes las simulaciones?
 - ¿Cuándo deben escribirse las simulaciones?
 - Fidelidad de las simulaciones
 - Las simulaciones se deben probar
 - ¿Qué hacer si no hay una simulación disponible?
- Stubbing*
 - Los peligros de abusar del *stubbing*

¿Cuándo es adecuado utilizar *stubbing*?

Pruebas de interacción

Preferencia de las pruebas de estado a las pruebas de interacción

¿Cuándo son apropiadas las pruebas de interacción?

Mejores prácticas para las pruebas de interacción

Conclusión

Resumen

14. Pruebas más grandes

¿Qué son las «pruebas más grandes»?

Fidelidad

Brechas frecuentes en las pruebas unitarias

¿Por qué no realizar pruebas más grandes?

Pruebas de gran tamaño en Google

Pruebas más grandes y el tiempo

Pruebas más grandes a escala de Google

Estructura de una prueba grande

El sistema bajo prueba

Datos para la prueba

Verificación

Tipos de pruebas más grandes

Prueba funcional de uno o más binarios interactivos

Pruebas de los navegadores y dispositivos

Pruebas de rendimiento, carga y estrés

Pruebas de la configuración de implementación

Pruebas exploratorias

Pruebas de regresión de diferencias A/B

UAT

Sistemas de sondeo y análisis canario

Recuperación en caso de catástrofe e ingeniería del caos

Evaluación al usuario

Grandes pruebas y el flujo de trabajo del desarrollador

Creación de pruebas grandes

Ejecución de pruebas grandes

Propiedad de las pruebas grandes

Conclusión

Resumen

15. Depreciación

¿Por qué hacer depreciación?

¿Por qué es tan difícil la depreciación?

Depreciación durante el diseño

Tipos de depreciación

Depreciación recomendada

Depreciación obligatoria

Advertencias de depreciación

Gestión del proceso de depreciación

Propietarios de los procesos

Hitos

Depreciación de las herramientas

Conclusión

Resumen

Parte IV. Herramientas

16. Control de versiones y gestión de ramas

¿Qué es el «control de versiones»?

¿Por qué es importante el control de versiones?

VCS centralizado frente a VCS distribuido

Fuente de verdad

Control de versiones frente a gestión de dependencias

Gestión de ramas

- El trabajo en curso es similar a una rama
- Ramas de desarrollo
- Ramas de versión
- Control de versiones en Google
 - One-Version
 - Escenario: varias versiones disponibles
 - La norma «One-Version»
 - Casi sin ramas longevas
 - ¿Qué pasa con las ramas de versión?
- Monorepos
- Futuro del control de versiones
- Conclusión
- Resumen

17. Code Search

- La interfaz de usuario de Code Search
- ¿Cómo utilizan los Googlers Code Search?
 - ¿Dónde?
 - ¿Qué?
 - ¿Cómo?
 - ¿Por qué?
 - ¿Quién y cuándo?
- ¿Por qué una herramienta web independiente?
 - Escala
 - Vista global del código sin necesidad de configuración
 - Especialización
 - Integración con otras herramientas para desarrolladores
 - Presentación de las API
- Impacto de la escala en el diseño
 - Latencia de la consulta de búsqueda
 - Latencia del índice

Implementación de Google

Índice de búsqueda

Clasificación

Selección de contrapartidas

Complejidad: repositorio en *head*

Complejidad: todos los resultados frente a los más relevantes

Complejidad: *head* versus ramas versus toda la historia versus espacios de trabajo

Expresividad: *token* frente a subcadena frente a expresión regular

Conclusión

Resumen

18. Sistemas de compilación y filosofía de la compilación

Propósito de un sistema de compilación

¿Qué sucede si no existe un sistema de compilación?

Pero todo lo que necesito ¡es un compilador!

¿*Scripts* de *shell* al rescate?

Sistemas de compilación modernos

Todo se trata de dependencias

Sistemas de compilación basados en tareas

Sistemas de compilación basados en artefactos

Compilaciones distribuidas

Tiempo, escala, contrapartidas

Tratamiento de los módulos y las dependencias

La utilización de módulos detallados y la regla 1:1:1

Minimización de la visibilidad de los módulos

Gestión de dependencias

Conclusión

Resumen

19. Critique, herramienta de revisión del código de Google

- Principios de las herramientas de revisión del código
- Flujo de revisión de código
 - Notificaciones
- Nivel 1: realización de un cambio
 - Diferenciaciones
 - Resultados del análisis
 - Integración estricta de herramientas
- Nivel 2: revisión de la solicitud
- Niveles 3 y 4: comprensión y comentarios del cambio
 - Comentarios
 - Comprensión del estado de un cambio
- Nivel 5: aprobación del cambio (puntuación del cambio)
- Nivel 6: confirmación del cambio
 - Después de la confirmación: seguimiento del historial
- Conclusión
- Resumen

20. Análisis estático

- Características del análisis estático eficaz
 - Escalabilidad
 - Usabilidad
- Lecciones clave para hacer que el análisis estático funcione
 - Céntrese en la satisfacción de los desarrolladores
 - Haga que el análisis estático forme parte del flujo principal de trabajo del desarrollador
 - Permita la contribución de los usuarios
- Tricorder: plataforma de análisis estático de Google
 - Herramientas integradas
 - Canales de retroalimentación integrados
 - Sugerencias de correcciones
 - Personalización por proyectos

Presubmits

Integración en el compilador

Análisis durante la edición y navegación por el código

Conclusión

Resumen

21. Gestión de dependencias

¿Por qué resulta tan difícil la gestión de dependencias?

Requisitos conflictivos y dependencias de diamante

Importación de dependencias

Promesas de compatibilidad

Consideraciones al hacer la importación

Cómo gestiona Google la importación de dependencias

Gestión de dependencias, en teoría

Nada cambia (también conocido como el «modelo de dependencias estáticas»)

Versionado semántico

Modelos de distribución por paquetes

Live at Head

Limitaciones de SemVer

SemVer puede que asegure una compatibilidad superior a la real

SemVer podría exagerar

Motivaciones

Minimum Version Selection

Entonces, ¿funciona SemVer?

Gestión de dependencias con recursos infinitos

Exportación de dependencias

Conclusión

Resumen

22. Cambios a gran escala

¿Qué es un «cambio a gran escala»?

¿Quién negocia con las LSC?

Obstáculos a los cambios atómicos

- Limitaciones técnicas

- Fusión de conflictos

- Sin cementerios encantados

- Heterogeneidad

- Pruebas

- Revisión de código

Infraestructura LSC

- Políticas y cultura

- Comprensión de la base de código

- Gestión de cambios

- Pruebas

- Soporte del lenguaje

El proceso LSC

- Autorización

- Creación de cambios

- Fragmentación y envío

- Limpieza

Conclusión

Resumen

23. Integración continua

Conceptos de IC

- Bucles de retroalimentación rápida

- Automatización

Prueba continua

- Desafíos de la IC

- Pruebas herméticas

Integración continua en Google

Caso de estudio de IC: Google Takeout

Pero no puedo permitirme IC

Conclusión

Resumen

24. Entrega continua

Modismos de entrega continua en Google

La rapidez es un deporte de equipo: cómo dividir una implementación en partes manejables

Evaluación de cambios en el aislamiento: funciones de protección de banderas

La lucha por conseguir agilidad: configuración de un tren de lanzamiento

Ningún binario es perfecto

Cumpla con su fecha límite de lanzamiento

Calidad y enfoque en el usuario: envíe solo lo que se utilice

Desplazamiento a la izquierda: tomar antes decisiones basadas en los datos

Cambio de la cultura del equipo: creación de disciplina en el despliegue

Conclusión

Resumen

25. La computación como servicio

Dominio del entorno informático

Automatización del trabajo

Contenerización y tenencia múltiple

Resumen

Escritura de *software* para la computación gestionada

Arquitectura para el fracaso

Trabajos por lotes frente a trabajos de servicio

Gestión del estado

- Conexión a un servicio
- Código único
- CaaS con el tiempo y la escala
 - Los contenedores como abstracción
 - Un servicio para gobernarlos a todos
 - Configuración enviada
- Elección de un servicio informático
 - Centralización frente a personalización
 - Nivel de abstracción: sin servidores
 - Público versus privado
- Conclusión
- Resumen

Parte V. Conclusión

Epílogo

Índice

Prólogo

Siempre me han fascinado infinitamente los detalles de cómo hace Google las cosas.

He preguntado a mis amigos de Google para conseguir información sobre cómo funcionan realmente las cosas dentro de la empresa.

¿Cómo gestionan el depósito del código monolítico y masivo sin caerse?

¿Cómo colaboran con éxito decenas de miles de ingenieros en miles de proyectos?

¿Cómo mantienen la calidad de sus sistemas?

Trabajar con antiguos empleados de Google solo ha aumentado mi curiosidad. Si alguna vez ha trabajado con un exingeniero de Google (o «Xoogler», como se lo llama a veces), sin duda habrá escuchado la frase «en Google nosotros...». Salir de Google a otras empresas parece ser una experiencia impactante, al menos desde el lado de la ingeniería. Por lo que este forastero puede decir, los sistemas y procesos para escribir código en Google deben estar entre los mejores del mundo, dada la escala de la empresa y la frecuencia con la que la gente canta sus alabanzas.

En la ingeniería de *software* en Google, un conjunto de Googlers (y algunos Xooglers) nos brindan un plan extenso para muchas de las prácticas, herramientas e incluso elementos culturales que subyacen en la ingeniería de *software* en Google. Es fácil centrarse en las increíbles herramientas que Google ha creado para ayudar a escribir código, y este libro ofrece muchos detalles sobre esas herramientas. Pero también va más allá de la simple descripción de las herramientas, para ofrecernos la filosofía y los procesos que siguen los equipos de Google. Estos se pueden adaptar a

una variedad de circunstancias, tenga o no la escala y las herramientas. Para mi deleite, hay varios capítulos en los que se profundiza en varios aspectos de las pruebas automatizadas, un tema que continúa encontrando demasiada resistencia en nuestra industria.

Lo mejor de la tecnología es que nunca hay una sola forma de hacer algo. En cambio, existen una serie de contrapartidas que todos debemos tener en cuenta, dependiendo de las circunstancias de nuestro equipo y la situación. ¿Qué podemos obtener de forma económica del código abierto? ¿Qué puede construir nuestro equipo? ¿Qué tiene sentido apoyar para nuestra escala? Cuando interrogaba a mis amigos Googlers, quería que me hablaran del mundo en el extremo de la escala: rico en recursos, tanto en talento como en dinero, con grandes exigencias en el *software* que se crea.

Esta información anecdótica me dio ideas sobre algunas opciones que, de otro modo, no habría considerado.

Con este libro, hemos escrito esas opciones para que todos las lean. Por supuesto, Google es una empresa única y sería una tontería suponer que la forma correcta de administrar su organización de ingeniería de *software* es copiar con precisión su fórmula. Aplicado de manera práctica, este libro le dará ideas sobre cómo se pueden hacer las cosas y mucha información que puede utilizar para reforzar sus argumentos para adoptar las mejores prácticas como las pruebas, la compartición de conocimientos y la creación de equipos colaborativos.

Es posible que nunca tenga usted que construir algo parecido a Google, y que ni siquiera desee recurrir en su organización a las mismas técnicas que ellos aplican. Pero, si no está familiarizado con las prácticas que ha desarrollado Google, se está perdiendo una perspectiva sobre la ingeniería de *software* que proviene de decenas de miles de ingenieros que han trabajado en colaboración en el *software* durante más de dos décadas. Ese conocimiento es demasiado valioso para ignorarlo.

—Camille Fournier

Autora, *The Manager's Path*

Prefacio

Este libro se titula *Ingeniería de software en Google*. ¿Qué entendemos exactamente por «ingeniería de *software*»? ¿Qué distingue a la «ingeniería de *software*» de la «programación» o la «informática»? ¿Y por qué Google tendría una perspectiva única que añadir al corpus de bibliografía previo sobre ingeniería de *software* escrita durante los últimos cincuenta años?

Las expresiones «programación» e «ingeniería de *software*» se han utilizado de manera intercambiable durante bastante tiempo en nuestra industria, aunque cada término tiene un énfasis diferente y diversas implicaciones. Los estudiantes universitarios tienden a estudiar Ciencias de la computación y consiguen trabajo escribiendo código como «programadores».

La «ingeniería de *software*», sin embargo, suena más seria, como si implicara la aplicación de algunos conocimientos teóricos para construir algo real y preciso. Los ingenieros mecánicos, los ingenieros civiles, los ingenieros aeronáuticos y los de otras disciplinas de la ingeniería practican la ingeniería. Todos trabajan en el mundo real y utilizan la aplicación de sus conocimientos teóricos para crear algo real. Los ingenieros de *software* también crean «algo real», aunque es menos tangible que las cosas que crean otros ingenieros.

A diferencia de las profesiones de ingeniería más establecidas, la teoría o práctica actual de la ingeniería de *software* no es tan rigurosa. Los ingenieros aeronáuticos deben seguir pautas y prácticas rígidas, porque los errores en sus cálculos pueden causar daños reales; la programación, en general, no ha seguido tradicionalmente prácticas tan rigurosas. Pero, a medida que el *software* se integra más en nuestras vidas, debemos adoptar y confiar en

métodos de ingeniería más rigurosos. Esperamos que este libro ayude a otros a ver un camino hacia prácticas de *software* más confiables.

Programación a lo largo del tiempo

Proponemos que la «ingeniería de *software*» abarque no solo el acto de escribir código, sino todas las herramientas y procesos que utiliza una organización para construir y mantener ese código a lo largo del tiempo. ¿Qué prácticas puede introducir una organización de *software* que mejor mantengan su valioso código a largo plazo? ¿Cómo pueden los ingenieros hacer más sostenible la base de código y más rigurosa la propia disciplina de la ingeniería del *software*? No tenemos respuestas esenciales a estas preguntas, pero esperamos que la experiencia colectiva de Google durante las últimas dos décadas ilumine posibles caminos para encontrar esas respuestas.

Una idea clave que compartimos en este libro es que la ingeniería de *software* se puede considerar como una «programación integrada a lo largo del tiempo». ¿Qué prácticas podemos introducir en nuestro código para hacerlo sostenible, capaz de reaccionar ante los cambios necesarios, a lo largo de su ciclo de vida, desde su concepción hasta su introducción, pasando por su mantenimiento y su eliminación?

En el libro, se hace hincapié en tres principios fundamentales que, en nuestra opinión, las organizaciones de *software* deberían tener en cuenta a la hora de diseñar, crear la arquitectura y escribir su código:

Tiempo y cambio

Cómo deberá adaptarse el código a lo largo de su vida.

Escala y crecimiento

Cómo deberá adaptarse una organización a medida que evoluciona.

Contrapartidas y costes

Cómo una organización toma decisiones, basándose en las lecciones del tiempo y el cambio, la escala y el crecimiento.

A lo largo de los capítulos, hemos tratado de enlazar con estos temas y señalar las formas en que tales principios afectan a las prácticas de ingeniería y permiten que sean sostenibles (véase el capítulo 1 para un análisis más completo).

Perspectiva de Google

Google tiene una perspectiva única sobre el crecimiento y la evolución de un ecosistema de *software* sostenible, derivado de nuestra escala y longevidad. Esperamos que las lecciones que hemos aprendido sean útiles a medida que su organización evolucione y adopte prácticas más sostenibles.

Dividimos los temas de este libro en tres aspectos principales del panorama de la ingeniería de *software* de Google:

- Cultura
- Procesos
- Herramientas

La cultura de Google es única, pero las lecciones que hemos aprendido en el desarrollo de nuestra cultura de ingeniería son de amplia aplicación. En los capítulos sobre la cultura (parte II), se destaca la naturaleza colectiva de una empresa de desarrollo de *software*, que el desarrollo de *software* es un esfuerzo de equipo y que los principios culturales adecuados son esenciales para que una organización crezca y se mantenga en buen funcionamiento.

Las técnicas descritas en los capítulos de «Procesos» (parte II) resultan familiares para la mayoría de los ingenieros de *software*, pero la base de código de gran tamaño y larga duración de Google proporciona una prueba de esfuerzo más completa para desarrollar las mejores prácticas. En esos capítulos, hemos intentado enfatizar lo que hemos descubierto que funciona a lo largo del tiempo y a escala, así como identificar áreas en las que aún no tenemos respuestas satisfactorias.

Por último, los capítulos dedicados a las herramientas (parte IV) ilustran el modo en que aprovechamos nuestras inversiones en la infraestructura de herramientas para proporcionar beneficios a la base de código, a medida que crece y envejece. En algunos casos, estas herramientas son específicas de Google, aunque indicamos las alternativas de código abierto o de terceros cuando corresponde. Esperamos que estos conocimientos básicos se apliquen a la mayoría de las organizaciones de ingeniería.

La cultura, los procesos y las herramientas que se describen en este libro describen las lecciones que, con suerte, un ingeniero de *software* típico aprende en el trabajo. Ciertamente, Google no tiene el monopolio de los buenos consejos, y nuestras experiencias, presentadas aquí, no pretenden dictar lo que su organización debe hacer. Este libro es nuestra perspectiva, pero esperamos que le resulte útil, ya sea adoptando estas lecciones directamente o utilizándolas como punto de partida a la hora de considerar sus propias prácticas, especializadas para su propio ámbito de problemas.

Esta obra tampoco pretende ser un sermón. El propio Google todavía aplica de forma imperfecta muchos de los conceptos de estas páginas. Las lecciones que hemos aprendido las hemos asimilado a través de nuestros fracasos: todavía cometemos errores, implementamos soluciones imperfectas y necesitamos repetir para mejorar. Sin embargo, el gran tamaño de la organización de ingeniería de Google garantiza que exista una diversidad de soluciones para cada problema. Esperamos que este libro contenga lo mejor de ese grupo.

Lo que no es este libro

Este libro no pretende contener el diseño de *software*, una disciplina que requiere su propio libro (y para el que ya existe mucho contenido). Aunque en la obra hay algo de código con fines ilustrativos, los principios son neutrales en cuanto al lenguaje y hay pocos consejos de «programación» en estos capítulos. Como resultado, este texto no trata muchos temas importantes en el desarrollo de *software*: administración de proyectos,