

O'REILLY®

Software Engineering at Google

Lessons Learned
from Programming
Over Time



Curated by Titus Winters,
Tom Mansreck & Hyrum Wright

Ingeniería de software en Google

Hoy en día, los ingenieros de software necesitan saber no solo cómo programar de manera efectiva, sino también cómo desarrollar prácticas de ingeniería adecuadas para que su base de código sea sostenible y saludable. Este libro enfatiza esta diferencia entre programación e ingeniería de software.

¿Cómo pueden los ingenieros de software administrar una base de código viva que evoluciona y responde a requisitos y demandas cambiantes a lo largo de su vida? Con base en su experiencia en Google, los ingenieros de software Titus Winters y Hyrum Wright, junto con el escritor técnico Tom Mansreck, presentan una mirada sincera y perspicaz sobre cómo algunos de los principales profesionales del mundo construyen y mantienen software. Este libro cubre la cultura, los procesos y las herramientas de ingeniería únicos de Google y cómo estos aspectos contribuyen a la eficacia de una organización de ingeniería.

Explorará tres principios fundamentales que las organizaciones de software deben tener en cuenta al diseñar, diseñar, escribir y mantener el código:

- Cómo *hora* afecta la sostenibilidad del software y cómo hacer que su código sea resistente con el tiempo
- Cómo *escala* afecta la viabilidad de las prácticas de software dentro de una organización de ingeniería
- Qué *compensaciones* un ingeniero típico debe hacer al evaluar las decisiones de diseño y desarrollo

"Siendo sincero

sobre compensaciones,
este libro explica la forma
en que Google hace
ingeniería de software,
lo que me hace más
productivo y feliz".

—Eric Haugh

Ingeniero de software en Google

tito inviernos, un ingeniero de software senior del personal de Google, es el líder de la biblioteca para la base de código C++ de Google: 250 millones de líneas de código editadas por miles de ingenieros distintos por mes.

Tom Mansreck es redactor técnico del personal de ingeniería de software de Google. Es miembro del equipo de bibliotecas de C++, desarrolla documentación, lanza clases de capacitación y documenta Aboseil, el código C++ de fuente abierta de Google.

hyrum-wright es ingeniero de software de plantilla en Google, donde dirige el grupo de herramientas de cambio automatizado de Google. Hyrum ha realizado más ediciones individuales en el código base de Google que cualquier otro ingeniero en la historia de la empresa.

INGENIERÍA DE SOFTWARE

EE.UU. \$59,99

79,99 dólares canadienses

Teléfono: 978-1-492-08279-8

5 59 99

9

18141

1082798

11111111



Twitter: @oreillymedia
facebook.com/oreilly

Ingeniería de software en Google

Lecciones aprendidas de la programación a lo largo del tiempo

Titus Winters, Tom Manshreck y Hyrum Wright

Pekín Boston Farnham Sebastopol Tokio

O'REILLY®

Ingeniería de software en Google

por Titus Winters, Tom Mansreck y Hyrum Wright

Derechos de autor © 2020 Google, LLC. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar con fines educativos, comerciales o de promoción de ventas. Las ediciones en línea también están disponibles para la mayoría de los títulos (<http://oreilly.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo/institucional: 800-998-9938 o corporativo@oreilly.com.

Redactor de Adquisiciones:ryan shaw

Indexador:Ellen Troutman-Zaig

Editores de desarrollo:alicia joven

Diseñador de interiores:david futato

Redactor de producción:cristobal faucher

Diseñador de la portada:karen montgomery

Editor de copia:Octal Publishing, LLC

Ilustrador:rebeca demarest

Corrector de pruebas:acebo bauer forsyth

Marzo 2020: Primera edición

Historial de revisiones de la primera edición

2020-02-28: Primer lanzamiento

2020-09-04: Segundo lanzamiento

Ver <http://oreilly.com/catalog/errata.csp?isbn=9781492082798>para detalles de lanzamiento.

El logotipo de O'Reilly es una marca registrada de O'Reilly Media, Inc.*Ingeniería de software en Google*, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Las opiniones expresadas en este trabajo son las de los autores y no representan las opiniones del editor. Si bien el editor y los autores se han esforzado de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y los autores renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por daños resultantes del uso o confianza en este trabajo. El uso de la información e instrucciones contenidas en este trabajo es bajo su propio riesgo. Si alguna muestra de código u otra tecnología que este trabajo contiene o describe está sujeta a licencias de código abierto o a los derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

978-1-492-08279-8

[LSI]

Tabla de contenido

Prefacio.....xvii

Prefacio.....xix

Parte I.Tesis

1.¿Qué es la ingeniería de software?.....	3
Tiempo y Cambio	6
Ley de Hyrum	8
Ejemplo: pedido de hash	9
¿Por qué no apuntar simplemente a “nada cambia”?	10
Escala y eficiencia	11
Políticas que no escalan Políticas que escalan bien Ejemplo: Actualización del	12
compilador Desplazamiento a la izquierda	14
Compensaciones y costos	14
Ejemplo: Marcadores	17
Entradas para el ejemplo de toma de decisiones: compilaciones distribuidas	18
Ejemplo: Decidir entre tiempo y escala	19
Revisar decisiones, cometer errores	20
Ingeniería de software versus programación	20
Conclusión	22
TL; DR	23
	24

Parte II.Cultura

2.Cómo trabajar bien en equipos.	27
Ayúdame a ocultar mi	27
código El mito del genio	28
Esconderse Considerado Dañino	30
Detección temprana	31
El factor autobús	31
Ritmo de progreso	32
En resumen, no se esconda,	34
todo se trata del equipo	34
Los tres pilares de la interacción social ¿Por	34
qué son importantes estos pilares?	35
Humildad, respeto y confianza en la práctica	36
Cultura post-mortem sin culpa	39
ser google	41
Conclusión	42
TL; DR	42
3.El intercambio de conocimientos.	43
Desafíos para el aprendizaje	43
Filosofía	45
Preparando el escenario: seguridad psicológica	46
Tutoría	46
Seguridad Psicológica en Grupos Grandes	47
Aumentando tu Conocimiento	48
Hacer preguntas	48
Comprender el contexto	49
Escalando sus preguntas: pregunte a la comunidad	50
Chats grupales	50
Listas de correo	50
YAQS: Plataforma de Preguntas y Respuestas	51
Escalando tu conocimiento: siempre tienes algo que enseñar	52
Horas de oficina	52
Charlas Técnicas y	52
Documentación de Clases	53
Código	56
Escalando el conocimiento de su organización	56
Cultivar una cultura de intercambio de conocimientos	56
Establecimiento de fuentes canónicas de información	58

Mantenerse en el bucle	61
Legibilidad: Tutoría estandarizada a través de la revisión del código	62
¿Qué es el proceso de legibilidad? ¿Por	63
qué tener este proceso?	64
Conclusión	66
TL; DR	67
4. Ingeniería para la Equidad.	69
El sesgo es el predeterminado Entender la necesidad de la diversidad Desarrollar la capacidad multicultural Hacer que la diversidad sea procesable	70
Rechazar enfoques singulares	72
Desafiar procesos establecidos	72
Valores versus resultados	74
Manténgase curioso, siga adelante Conclusión	75
TL; DR	76
5. Cómo liderar un equipo.	81
Gerentes y líderes tecnológicos (y ambos)	81
El Gerente de Ingeniería El Líder Técnico	82
El gerente líder de tecnología	82
Pasar de un rol de colaborador individual a un rol de liderazgo	83
Lo único que hay que temer es... bueno, todo	84
Liderazgo de servicio	85
El Gerente de Ingeniería	86
Gerente es una palabra de cuatro letras Gerente de ingeniería de hoy	86
Antipatrones	87
Antipatrón: Contratar a los empujones Antipatrón:	88
Ignorar a los empleados de bajo rendimiento	89
Antipatrón: Ignorar los problemas humanos Antipatrón:	89
Ser amigo de todos Antipatrón: Poner en peligro la barra de contratación Antipatrón: Tratar a su equipo como niños Patrones positivos	90
perder el ego	91
ser un maestro zen	92
ser un catalizador	93
	93
	94
	96

Eliminar obstáculos	96
Sea un maestro y un mentor	97
Establezca metas claras	97
Se honesto	98
Seguimiento de la felicidad	99
La pregunta inesperada Otros consejos y trucos Las personas son como las plantas	100 101 103
Motivación intrínseca versus extrínseca	104
Conclusión	105
TL; DR	105
6. Liderando a Escala.....	107
Estar siempre decidiendo	108
La parábola del avión	108
Identifique a los ciegos	109
Identificar las compensaciones	109
clave Decidir, luego iterar	110
Siempre se va	112
Su misión: Construir un equipo "autodirigido"	112
dividiendo el espacio del problema	113
Siempre esté escalando	116
El Ciclo del Éxito Importante	116
versus Urgente Aprenda a dejar caer las bolas	118
Protegiendo tu energía	119
Conclusión	120
TL; DR	122
7. Medición de la productividad de la ingeniería.	123
¿Por qué debemos medir la productividad de la ingeniería? Triage: ¿Vale la pena medirlo?	123 125
Selección de métricas significativas con objetivos y señales	129
Objetivos	130
Señales	132
Métrica	132
Uso de datos para validar métricas Toma de medidas y seguimiento de resultados	133 137
Conclusión	137
TL; DR	137

Parte III. Procesos

8. Guías y reglas de estilo.	141
¿Por qué tener reglas?	142
Crear las reglas	143
Principios rectores	143
La guía de estilo	151
Cambiar las reglas	154
El proceso	155
Los árbitros del estilo	156
Excepciones	156
Guía	157
Aplicar las reglas	158
Comprobadores de errores	160
Formateadores de código	161
Conclusión	163
TL; DR	163
9. Revisión de código.	165
Flujo de revisión de código	166
Cómo funciona Code Review en Google	167
Code Review Beneficios	170
Corrección del código	171
Comprensión de código	172
Coherencia de código	173
Beneficios psicológicos y culturales	174
Intercambio de conocimientos	175
Mejores prácticas de revisión de código	176
Sea Cortés y Profesional	176
Escriba Pequeños Cambios	177
Escribir buenas descripciones de cambios	178
Mantener a los revisores al mínimo	179
Automatizar cuando sea posible	179
Tipos de revisiones de código	180
Reseñas de Código Greenfield	180
Cambios de comportamiento, mejoras y optimizaciones	181
Corrección de errores y reverisiones	181
Refactorizaciones y cambios a gran escala	182
Conclusión	182
TL; DR	183

10 Documentación	185
¿Qué califica como documentación? ¿Por qué se necesita la documentación? La documentación es como el código	185
Conozca a su audiencia	188
Tipos de audiencias	190
Tipos de documentación	191
Documentación de referencia	192
Documentos de diseño	193
Tutoriales	195
Documentación Conceptual	196
Landing Pages	198
Revisões de documentación	198
Filosofía de la documentación	199
QUIÉN, QUÉ, CUÁNDΟ, DÓNDE y POR QUÉ	201
El principio, el medio y el final	201
Los parámetros de la buena documentación	202
Documentos obsoletos	202
¿Cuándo necesita escritores técnicos?	203
Conclusión	204
TL; DR	204
11 Descripción general de las pruebas	205
¿Por qué escribimos pruebas?	208
La historia del servidor web de Google	209
Pruebas a la velocidad del desarrollo moderno Escribir, ejecutar, reaccionar	210
Beneficios de Probar Código	212
Diseñando una Suite de Pruebas	213
Tamaño de prueba	214
Alcance de prueba	215
La regla de Beyoncé	219
Una nota sobre las pruebas de cobertura de código a escala de Google	221
Las trampas de un gran conjunto de pruebas Historia de las pruebas en Google	222
Clases de orientación	223
Prueba certificada	224
Pruebas en el inodoro	225
Testing Culture Today	226
Pruebas en el inodoro	227
Testing Culture Today	228

Los límites de las pruebas automatizadas	229
Conclusión	230
TL; DR	230
12Examen de la unidad	231
La importancia de la mantenibilidad para prevenir pruebas frágiles	232
Esfúércese por pruebas inalterables	233
Prueba a través de API públicas	234
Estado de prueba, no interacciones	238
Escribir pruebas claras	239
Haga que sus pruebas sean completas y concisas	240
Comportamientos de prueba, no métodos	241
No ponga lógica en las pruebas	246
Escriba mensajes de error claros	247
Pruebas y código compartido: HÚMEDO, no SECO	248
Valores compartidos	251
Configuración compartida	253
Asistentes compartidos y validación	254
Definición de la infraestructura de prueba	255
Conclusión	256
TL; DR	256
13Dobles de prueba	257
El impacto de los dobles de prueba en el desarrollo de software	258
Dobles de prueba en Google	258
Conceptos básicos	259
Un ejemplo de prueba de costuras	259
dobles	260
Marcos burlones	261
Técnicas para usar dobles de prueba	262
fingiendo	263
talonar	263
Pruebas de interacción	264
Implementaciones reales	264
Prefiere el realismo al aislamiento	265
Cómo decidir cuándo usar una simulación de implementación real	266
¿Por qué son importantes las falsificaciones?	269
¿Cuándo se deben escribir las falsificaciones? La fidelidad de las falsificaciones	270
	271

Las falsificaciones deben ser probadas	272
Qué hacer si una falsificación no está disponible	272
Los peligros del uso excesivo de stubbing	273
¿Cuándo es apropiado stubbing? Pruebas de interacción	275
Preferir las pruebas estatales a las pruebas de interacción	275
¿Cuándo son apropiadas las pruebas de interacción? Mejores prácticas para las pruebas de interacción	277
Conclusión	277
	280
TL; DR	280
14 Pruebas más grandes.	281
¿Qué son las pruebas más grandes?	281
Fidelidad	282
Brechas comunes en las pruebas unitarias	283
¿Por qué no tener pruebas más grandes?	285
Pruebas más grandes en Google	286
Pruebas más grandes y pruebas más largas en el tiempo a escala de Google	286
Estructura de una prueba grande	288
El sistema bajo prueba Datos de prueba	289
Verificación	294
Tipos de pruebas más grandes	295
Pruebas funcionales de uno o más navegadores binarios interactivos y pruebas de dispositivos	296
Pruebas de rendimiento, carga y estrés	297
Pruebas de configuración de implementación Pruebas exploratorias	297
Prueba de regresión de diferencia A/B	298
UAT	299
Evaluación de usuarios de ingeniería del caos y recuperación ante desastres de Probers y Canary	301
Analysis	302
Pruebas grandes y el flujo de trabajo del desarrollador	303
Creación de pruebas grandes	304
Ejecución de pruebas grandes	305
Posesión de pruebas grandes	305
Conclusión	308
TL; DR	309

15.Deprecación.....	311
¿Por qué desaprobar?	312
¿Por qué es tan difícil la desaprobación?	313
Deprecación durante el diseño	315
Tipos de depreciación	316
Depreciación de aviso	316
Desaprobación obligatoria	317
Advertencias de obsolescencia	318
Gestión del proceso de desaprobación	319
Propietarios de procesos	320
Hitos	320
Herramientas de desaprobación	321
Conclusión	322
TL; DR	323

Parte IV.Herramientas

diecisésis.Control de Versiones y Gestión de Sucursales.....	327
¿Qué es el control de versiones?	327
¿Por qué es importante el control de versiones? VCS	329
centralizado frente a VCS distribuido Fuente de la verdad	331
verdad	334
Control de versiones frente a gestión de dependencias	336
Gestión de sucursales	336
El trabajo en progreso es similar a una sucursal	336
Desarrollo de sucursales	337
Ramas de lanzamiento	339
Control de versiones en Google	340
Una versión	340
Escenario: Múltiples versiones disponibles La regla de “una versión”	341
(Casi) No hay ramas de larga duración	342
¿Qué pasa con las ramas de liberación?	343
Monorepos	344
Futuro del control de versiones	345
Conclusión	346
TL; DR	348

17Búsqueda de código.....	351
La interfaz de usuario de búsqueda de código	352
¿Cómo utilizan los Googlers la búsqueda de códigos?	353
¿Donde?	353
¿Qué?	354
¿Cómo?	354
¿Por qué?	354
¿Quién y cuándo?	355
¿Por qué una herramienta web independiente?	355
Escala	355
Especialización de vista de código global de configuración cero	356
Integración con otras herramientas de desarrollo	356
API Exposición	359
Impacto de la escala en el diseño	359
Latencia de consulta de búsqueda	359
Índice de latencia	360
Implementación de Google	361
Índice de búsqueda	361
Clasificación	363
Compensaciones seleccionadas	366
Integridad: Repositorio en la cabecera Integridad: Todo frente a los resultados más relevantes Integridad: La cabeza frente a las sucursales frente a todo el historial frente a espacios de trabajo	366
367	367
Expresividad: Token Versus Substring Versus Regex	368
Conclusión	369
TL; DR	370
18Construir sistemas y construir filosofía.....	371
Propósito de un sistema de compilación	371
¿Qué sucede sin un sistema de compilación?	372
¡Pero todo lo que necesito es un compilador! Shell Scripts al rescate	373
Sistemas de construcción modernos	375
Se trata de dependencias Sistemas de compilación basados en tareas Sistemas de compilación basados en artefactos	375
376	376
380	380
Compilaciones distribuidas	386
Tiempo, escala, compensaciones	390

Manejo de módulos y dependencias	390
Uso de módulos granulares y la regla 1:1:1 para minimizar la visibilidad del módulo	391
Gestión de dependencias	392
Conclusión	397
TL; DR	397
19Crítica: herramienta de revisión de código de Google.....	399
Principios de herramientas de revisión de código	399
Flujo de revisión de código	400
Notificaciones	402
Etapa 1: Crear un cambio	402
diferenciando	403
Resultados de análisis	404
Integración estrecha de herramientas	406
Etapa 2: Solicitud de revisión	406
Etapas 3 y 4: comprender y comentar un cambio	408
Comentando	408
Comprensión del estado de un cambio Etapa 5:	410
Aprobaciones de cambios (puntuación de un cambio) Etapa	412
6: Confirmación de un cambio	413
Después de la confirmación: Conclusión del historial	414
de seguimiento	415
TL; DR	416
20Análisis estático.....	417
Características del análisis estático eficaz	418
Escalabilidad	418
usabilidad	418
Lecciones clave para hacer que el análisis estático funcione	419
Centrarse en la felicidad del desarrollador	419
Convierta el análisis estático en parte del flujo de trabajo principal del desarrollador Permita que los usuarios contribuyan	420
desarrollador	420
Tricorder: la plataforma de análisis estático de Google	421
Herramientas integradas	422
Correcciones sugeridas de canales de retroalimentación integrados	423
Envíos previos de personalización por proyecto	424
Integración del compilador	425
Análisis durante la edición y navegación de código	426
	427

Conclusión	428
TL; DR	428
21 Gestión de Dependencias.....	429
¿Por qué es tan difícil la gestión de dependencias?	431
Requisitos en conflicto y dependencias Diamond	431
Importación de dependencias	433
Promesas de compatibilidad	433
Consideraciones a la hora de importar	436
Cómo gestiona Google la importación de dependencias	437
Gestión de dependencias, en teoría	439
Nada cambia (también conocido como el modelo de dependencia estática)	439
Versionado semántico	440
Modelos de distribución agrupados	441
Live at Head	442
Las limitaciones de SemVer	443
SemVer podría limitar en exceso	444
SemVer podría prometer en exceso	445
las motivaciones	446
Selección de versión mínima	447
Entonces, ¿funciona SemVer?	448
Gestión de dependencias con recursos infinitos	449
Exportación de dependencias	452
Conclusión	456
TL; DR	456
22 Cambios a gran escala.....	459
¿Qué es un cambio a gran escala?	460
¿Quién se ocupa de los LSC?	461
Barreras a los cambios atómicos	463
Limitaciones técnicas	463
Fusionar conflictos	463
Sin cementerios embrujados	464
Heterogeneidad	464
Pruebas	465
Revisión de código	467
Infraestructura LSC	468
Políticas y Cultura	469
Perspectiva de la base de código	470
Gestión del cambio	470
Pruebas	471

Ayuda de idioma	471
El proceso LSC	472
Autorización	473
Creación de cambios	473
Limpieza de fragmentación	474
y envío	477
Conclusión	477
TL; DR	478
23Integración continua.....	479
Conceptos de IC	481
Bucles de retroalimentación rápidos	481
Automatización	483
Pruebas continuas	485
Desafíos de IC	490
Pruebas herméticas	491
IC en Google	493
Estudio de caso de CI: Google Takeout	496
pero no puedo pagar CI	503
Conclusión	503
TL; DR	503
24Entrega continua.....	505
Modismos de entrega continua en Google	506
Velocity es un deporte de equipo: cómo dividir una implementación en manejable	
Piezas	507
Evaluación de cambios en aislamiento: características de protección de banderas	508
Esfuerzo por la agilidad: configuración de un tren de liberación	509
Ningún binario es perfecto Cumpla con su	509
fecha límite de lanzamiento	510
Calidad y enfoque en el usuario: enviar solo lo que se usa Desplazamiento a la	511
izquierda: tomar decisiones basadas en datos antes Cambiar la cultura del	512
equipo: crear disciplina en la implementación Conclusión	513
514	514
TL; DR	514
25Cálculo como servicio.....	517
Domar el entorno informático	518
Automatización del Trabajo	518
Resumen de contenedores y tenencia	520
múltiple	523

Software de escritura para computación administrada	523
Arquitectura para fallas	523
Lote versus servicio	525
Estado administrador	527
Conexión a un código	528
único de servicio	529
CaaS a lo largo del tiempo y la escala	530
Contenedores como un servicio de	530
Abstracción Uno para gobernarlos a	533
todos Configuración enviada Elección	535
de un servicio informático	535
Centralización frente a personalización	537
Nivel de abstracción: público sin servidor	539
frente a privado	543
Conclusión	544
TL; DR	545

Parte V. Conclusión

Epílogo.....	549
Índice.....	551

Prefacio

Siempre me han fascinado infinitamente los detalles de cómo Google hace las cosas. He interrogado a mis amigos de Google para obtener información sobre cómo funcionan realmente las cosas dentro de la empresa. ¿Cómo manejan un depósito de 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 ex Googlers solo ha aumentado mi curiosidad. Si alguna vez ha trabajado con un ex ingeniero de Google (o "Xoogler", como a veces se les llama), 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 de las cosas. 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 Ingeniería de software en Google, un conjunto de Googlers (y algunos Xooglers) nos brindan un plan detallado para muchas de las prácticas, herramientas e incluso elementos culturales que subyacen a la ingeniería de software en Google. Es fácil concentrarse demasiado en las increíbles herramientas que Google ha creado para admitir la escritura de código, y este libro brinda muchos detalles sobre esas herramientas. Pero también va más allá de simplemente describir las herramientas para brindarnos la filosofía y los procesos que siguen los equipos de Google. Estos se pueden adaptar para adaptarse a una variedad de circunstancias, ya sea que tenga o no la escala y las herramientas. Para mi deleite, hay varios capítulos que profundizan en varios aspectos de las pruebas automatizadas, un tema que continúa encontrando demasiada resistencia en nuestra industria.

Lo bueno de la tecnología es que nunca hay una sola forma de hacer algo. En cambio, hay una serie de compensaciones que todos debemos hacer dependiendo de las circunstancias de nuestro equipo y situación. ¿Qué podemos sacar a bajo precio del código abierto? ¿Qué puede construir nuestro equipo? ¿Qué tiene sentido apoyar para nuestra escala? Cuando estaba interrogando a mis amigos Googlers, quería escuchar sobre el mundo en el extremo final de la escala: rico en recursos, tanto en talento como en dinero, con altas demandas en el software que se está desarrollando.

construido. 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 ejecutar su organización de ingeniería de software es copiar con precisión su fórmula. Aplicado en la práctica, este libro le dará ideas sobre cómo se pueden hacer las cosas y mucha información que puede usar para reforzar sus argumentos para adoptar las mejores prácticas, como las pruebas, el intercambio de conocimientos y la creación de equipos colaborativos.

Es posible que nunca necesite crear Google usted mismo, y es posible que ni siquiera desee utilizar las mismas técnicas que aplican en su organización. 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 trabajaron en colaboración en software a lo largo de más de dos décadas. Ese conocimiento es demasiado valioso para ignorarlo.

—*Camille Fournier*
Autor, El camino del gerente

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 de las “ciencias de la computación”? ¿Y por qué Google tendría una perspectiva única para agregar al corpus de literatura previa sobre ingeniería de software escrita en los últimos 50 años?

Los términos “programación” e “ingeniería de software” se han usado indistintamente durante bastante tiempo en nuestra industria, aunque cada término tiene un énfasis diferente y diferentes implicaciones. Los estudiantes universitarios tienden a estudiar informática y obtienen trabajos escribiendo código como “programadores”.

“Ingeniería de software”, sin embargo, suena más serio, como si implicara la aplicación de algún conocimiento teórico para construir algo real y preciso. Los ingenieros mecánicos, los ingenieros civiles, los ingenieros aeronáuticos y aquellos en otras disciplinas de 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, tradicionalmente no ha seguido 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 mantenga su

código valioso a largo plazo? ¿Cómo pueden los ingenieros hacer que una base de código sea más sostenible y que la disciplina de la ingeniería de software sea más rigurosa? No tenemos respuestas fundamentales 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 “programación integrada a lo largo del tiempo”. ¿Qué prácticas podemos introducir en nuestro código para hacerlos *sostenible*—capaz de reaccionar al cambio necesario—a lo largo de su ciclo de vida, desde la concepción hasta la introducción, el mantenimiento y la desaprobación?

El libro enfatiza tres principios fundamentales que creemos que las organizaciones de software deben tener en cuenta al diseñar, diseñar y escribir su código:

Tiempo y Cambio

Cómo tendrá que adaptarse el código a lo largo de su vida

Escala y crecimiento

Cómo una organización necesitará adaptarse a medida que evoluciona

Compensaciones y costos

Cómo una organización toma decisiones, con base en las lecciones de tiempo y cambio y escala y crecimiento

A lo largo de los capítulos, hemos tratado de relacionarnos con estos temas y señalar las formas en que dichos principios afectan las prácticas de ingeniería y les permiten ser sostenibles. (Ver [Capítulo 1](#) para una discusión completa.)

Perspectiva de Google

Google tiene una perspectiva única sobre el crecimiento y la evolución de un ecosistema de software sostenible, derivada 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.

Hemos dividido los temas de este libro en tres aspectos principales del panorama de 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 ampliamente aplicables. Nuestros capítulos de Cultura ([Parte II](#)) enfatizan 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 saludable.

Las técnicas descritas en nuestros capítulos de Procesos ([Parte III](#)) son familiares para la mayoría de los ingenieros de software, pero el gran tamaño y la base de código de larga duración de Google proporcionan una prueba de estrés más completa para desarrollar las mejores prácticas. Dentro de esos capítulos, hemos tratado de enfatizar lo que hemos encontrado que funciona a lo largo del tiempo y a escala, así como identificar áreas en las que aún no tenemos respuestas satisfactorias.

Finalmente, nuestros capítulos de Herramientas ([Parte IV](#)) ilustran cómo aprovechamos nuestras inversiones en infraestructura de herramientas para brindar beneficios a nuestra base de código a medida que crece y envejece. En algunos casos, estas herramientas son específicas de Google, aunque señalamos alternativas de código abierto o de terceros cuando corresponda. Esperamos que estos conocimientos básicos se apliquen a la mayoría de las organizaciones de ingeniería.

La cultura, los procesos y las herramientas descritas en este libro describen las lecciones que un ingeniero de software típico aprende con suerte en el trabajo. Google ciertamente no tiene el monopolio de los buenos consejos, y nuestras experiencias presentadas aquí no pretenden dictar lo que debe hacer su organización. Este libro es nuestra perspectiva, pero esperamos que lo encuentre útil, ya sea adoptando estas lecciones directamente o usándolas como punto de partida cuando considere sus propias prácticas, especializadas para su propio dominio de problemas.

Este libro tampoco pretende ser un sermón. Google mismo aún aplica de manera imperfecta muchos de los conceptos dentro de estas páginas. Las lecciones que hemos aprendido, las aprendimos a través de nuestros fracasos: todavía cometemos errores, implementamos soluciones imperfectas y necesitamos iterar hacia la mejora. Sin embargo, el gran tamaño de la organización de ingeniería de Google garantiza que haya una diversidad de soluciones para cada problema. Esperamos que este libro contenga lo mejor de ese grupo.

Lo que este libro no es

Este libro no pretende cubrir el diseño de software, una disciplina que requiere su propio libro (y para la cual ya existe mucho contenido). Aunque hay algo de código en este libro con fines ilustrativos, los principios son neutrales en términos de lenguaje y hay pocos consejos reales de "programación" dentro de estos capítulos. Como resultado, este texto no cubre muchos temas importantes en el desarrollo de software: administración de proyectos, diseño de API, fortalecimiento de la seguridad, internacionalización, marcos de interfaz de usuario u otras preocupaciones específicas del idioma. Su omisión en este libro no implica su falta de importancia. En cambio, elegimos no cubrirlas aquí sabiendo que no podríamos brindarles el tratamiento que merecen. Hemos tratado de hacer que las discusiones en este libro sean más sobre ingeniería y menos sobre programación.

Comentarios de despedida

Este texto ha sido un trabajo de amor en nombre de todos los que han contribuido, y esperamos que lo reciba tal como se entrega: como una ventana a cómo una gran organización de ingeniería de software construye sus productos. También esperamos que sea una de las muchas voces que ayuden a que nuestra industria adopte prácticas más progresistas y sostenibles. Lo que es más importante, esperamos que disfrute leyéndolo y que pueda adoptar algunas de sus lecciones para sus propias preocupaciones.

—*Tom Mansreck*

Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

Itálico

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

Ancho constante

Se utiliza para listas de programas, así como dentro de párrafos para referirse a elementos de programas como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

Negrita de ancho constante

Muestra comandos u otro texto que el usuario debe escribir literalmente.

Cursiva de ancho constante

Muestra texto que debe ser reemplazado con valores proporcionados por el usuario o por valores determinados por el contexto.



Este elemento significa una nota general.

Aprendizaje en línea de O'Reilly



Durante más de 40 años, *Medios O'Reilly* ha brindado capacitación en tecnología y negocios, conocimiento y perspectiva para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparte su conocimiento y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le brinda acceso a pedido a cursos de capacitación en vivo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de texto y video de O'Reilly y más de 200 editores más. Para mayor información por favor visite <http://oreilly.com>.

Cómo contactarnos

Dirija sus comentarios y preguntas sobre este libro a la editorial:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (en los Estados Unidos o Canadá)
707-829-0515 (internacional o local)
707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos las erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en <https://oreilly.ingineria-de-software-en-google>.

Correo electrónico bookquestions@oreilly.com para comentar o hacer preguntas técnicas sobre este libro.

Para noticias y más información sobre nuestros libros y cursos, visite nuestro sitio web en <http://www.oreilly.com>.

Encuentranos en Facebook: <http://facebook.com/oreilly> Síganos en

Twitter: <http://twitter.com/oreillymedia> Míranos en YouTube: <http://www.youtube.com/oreillymedia>

Expresiones de gratitud

Un libro como este no sería posible sin el trabajo de muchos otros. Todo el conocimiento contenido en este libro nos ha llegado a todos a través de la experiencia de tantos otros en Google a lo largo de nuestras carreras. Somos los mensajeros; otros vinieron antes que nosotros, en Google y en otros lugares, y nos enseñaron lo que ahora les presentamos. No podemos enumerarlos a todos aquí, pero deseamos reconocerlos.

También nos gustaría agradecer a Melody Meckfessel por apoyar este proyecto en su infancia, así como a Daniel Jasper y Danny Berlin por apoyarlo hasta su finalización.

Este libro no hubiera sido posible sin el enorme esfuerzo colaborativo de nuestros curadores, autores y editores. Aunque los autores y editores reciben un reconocimiento específico en cada capítulo o llamada, nos gustaría tomarnos un tiempo para reconocer a quienes contribuyeron en cada capítulo brindando comentarios, debates y revisiones reflexivos.

- **¿Qué es la ingeniería de software?**:Sanjay Ghemawat, Andrew Hyatt
- **Trabajando bien en equipos**:Sibley Bacon, Joshua Morton
- **El intercambio de conocimientos**:Dimitri Glazkov, Kyle Lemons, John Reese, David Symonds, Andrew Trenk, James Tucker, David Kohlbrenner, Rodrigo Damazio Bovendorp
- **Ingeniería para la Equidad**:Kamau Bobb, Bruce Lee
- **Cómo liderar un equipo**:Jon WileyLaurent Le Brun
- **Liderando a escala**:Bryan O'Sullivan, Bharat Mediratta, Daniel Jasper, Shaindel Schwartz
- **Medición de la productividad de la ingeniería**:Andrea Knight, Collin Green, Caitlin Sadowski, Max-Kanat Alexander, Yilei Yang
- **Guías de estilo y reglas**:Max Kanat-Alexander, Titus Winters, Matt Austern, James Dennett
- **Revisión de código**:Max Kanat-Alexander, Brian Ledger, Mark Barolak
- **Documentación**:Jonas Wagner, Smit Hinsu, Geoffrey Romer
- **Descripción general de las pruebas**:Erik Kuefler, Andrew Trenk, Dillon Bly, Joseph Graves, Neal Norwitz, Jay Corbett, Mark Striebeck, Brad Green, Miško Hevery, Antoine Picard, Sarah Storck
- **Examen de la unidad**:Andrew Trenk, Adam Bender, Dillon Bly, Joseph Graves, Titus Winters, Hyrum Wright, Augie Fackler
- **Dobles de prueba**:Joseph Graves, General Civil

- **Pruebas más grandes:** Adam Bender, Andrew Trenk, Erik Kuefler, Matthew Beaumont-Gay
- **Deprecación:** Greg Miller, Andy Shulman
- **Control de Versiones y Gestión de Sucursales:** Rachel Potvin, Victoria Clarke
- **Búsqueda de código:** jenny wang
- **Sistemas de construcción y filosofía de construcción:** Hyrum Wright, Titus Winters, Adam Bender, Jeff Cox, Jacques Pienaar
- **Crítica: Herramienta de revisión de código de Google:** Mikołaj Dądela, Hermann Loose, Eva May, Alice Kober-Sotzek, Edwin Kempin, Patrick Hiesel, Ole Rehmsen, Jan Macek
- **Análisis estático:** Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, Edward Aftandilian, Collin Winter, Eric Haugh
- **Gestión de dependencias:** Russ Cox, Nicholas Dunn
- **Cambios a gran escala:** Matthew Fowles, Kulukundis, Adam Zarek
- **Integración continua:** Jeff Listfield, John Penix, Kaushik Sridharan, Sanjeev Dhanda
- **Entrega continua:** Dave Owens, Sheri Shipe, Bobbi Jones, Matt Duftler, Brian Szuter
- **Servicios informáticos:** Tim Hockin, Collin Winter, Jarek Kuśmierenk

Además, nos gustaría agradecer a Betsy Beyer por compartir su visión y experiencia al haber publicado el original *Ingeniería de confiabilidad del sitio* libro, lo que hizo que nuestra experiencia fuera mucho más fluida. Christopher Guzikowski y Alicia Young en O'Reilly hicieron un trabajo increíble lanzando y guiando este proyecto hasta su publicación.

Los curadores también quisieran agradecer personalmente a las siguientes personas:

Tom Mansreck: A mi mamá y mi papá por hacerme creer en mí mismo y trabajar conmigo en la mesa de la cocina para hacer mi tarea.

Tito inviernos: A papá, por mi camino. A mamá, por mi voz. Para Victoria, para mi corazón. A Raf, por apoyarme. También, al Sr. Snyder, Ranwa, Z, Mike, Zach, Tom (y todos los Payne), mec, Toby, cgd y Melody por las lecciones, la tutoría y la confianza.

Hyrum Wright: A mamá y papá por su aliento. A Bryan y los habitantes de Bakerland, por mi primera incursión en el software. A Dewayne, por continuar ese viaje. A Hannah, Jonathan, Charlotte, Spencer y Ben por su amor e interés. A Heather por estar ahí durante todo el proceso.

PARTE I

Tesis

¿Qué es la ingeniería de software?

*Escrito por Titus Winters
Editado por Tom Mansreck*

Nada está construido sobre piedra; todo está construido sobre arena, pero debemos construir como si la arena fuera piedra.

—Jorge Luis Borges

Vemos tres diferencias críticas entre la programación y la ingeniería de software: tiempo, escala y las compensaciones en juego. En un proyecto de ingeniería de software, los ingenieros deben preocuparse más por el paso del tiempo y la eventual necesidad de cambio. En una organización de ingeniería de software, debemos preocuparnos más por la escala y la eficiencia, tanto del software que producimos como de la organización que lo produce. Finalmente, como ingenieros de software, se nos pide que tomemos decisiones más complejas con resultados más importantes, a menudo basados en estimaciones imprecisas de tiempo y crecimiento.

Dentro de Google, a veces decimos: "La ingeniería de software es una programación integrada a lo largo del tiempo". La programación es sin duda una parte importante de la ingeniería de software: después de todo, la programación es la forma de generar nuevo software en primer lugar. Si acepta esta distinción, también queda claro que es posible que debamos delinejar entre tareas de programación (desarrollo) y tareas de ingeniería de software (desarrollo, modificación, mantenimiento). La adición de tiempo añade una nueva dimensión importante a la programación. Los cubos no son cuadrados, la distancia no es velocidad. La ingeniería de software no es programación.

Una forma de ver el impacto del tiempo en un programa es pensar en la pregunta: "¿Cuál es la vida útil esperada¹ de tu código? Respuestas razonables a esta pregunta.

¹ No nos referimos a "vida útil de ejecución", nos referimos a "vida útil de mantenimiento": ¿cuánto tiempo seguirá existiendo el código? construido, ejecutado y mantenido? ¿Durante cuánto tiempo proporcionará valor este software?

variar en aproximadamente un factor de 100.000. Es tan razonable pensar en un código que debe durar unos minutos como imaginar un código que vivirá durante décadas. Generalmente, el código en el extremo corto de ese espectro no se ve afectado por el tiempo. Es poco probable que necesite adaptarse a una nueva versión de sus bibliotecas subyacentes, sistema operativo (SO), hardware o versión de idioma para un programa cuya utilidad dura solo una hora. Estos sistemas de corta duración son efectivamente "solo" un problema de programación, de la misma manera que un cubo comprimido lo suficiente en una dimensión es un cuadrado. A medida que ampliamos ese tiempo para permitir vidas más largas, el cambio se vuelve más importante. En un lapso de una década o más, la mayoría de las dependencias del programa, ya sean implícitas o explícitas, probablemente cambiarán.

Esta distinción está en el centro de lo que llamamos *sostenibilidad* para programas tu proyecto es *sostenible*, durante la vida útil esperada de su software, es capaz de reaccionar ante cualquier cambio valioso que se presente, ya sea por razones técnicas o comerciales. Es importante destacar que solo buscamos la capacidad: puede optar por no realizar una actualización determinada, ya sea por falta de valor u otras prioridades.² Cuando es fundamentalmente incapaz de reaccionar ante un cambio en la tecnología subyacente o en la dirección del producto, está haciendo una apuesta de alto riesgo con la esperanza de que dicho cambio nunca se vuelva crítico. Para proyectos a corto plazo, esa podría ser una apuesta segura. Durante varias décadas, probablemente no lo sea.³

Otra forma de ver la ingeniería de software es considerar la escala. ¿Cuántas personas están implicadas? ¿Qué papel juegan en el desarrollo y mantenimiento a lo largo del tiempo? Una tarea de programación es a menudo un acto de creación individual, pero una tarea de ingeniería de software es un esfuerzo de equipo. Un intento temprano de definir la ingeniería de software produjo una buena definición para este punto de vista: "El desarrollo de múltiples personas de programas de múltiples versiones".⁴ Esto sugiere que la diferencia entre la ingeniería de software y la programación es tanto de tiempo como de personas. La colaboración en equipo presenta nuevos problemas, pero también proporciona más potencial para producir sistemas valiosos que cualquier programador individual.

La organización del equipo, la composición del proyecto y las políticas y prácticas de un proyecto de software dominan este aspecto de la complejidad de la ingeniería de software. Estos problemas son inherentes a la escala: a medida que crece la organización y se expanden sus proyectos, ¿se vuelve más eficiente en la producción de software? ¿Nuestro flujo de trabajo de desarrollo

2 Esta es quizás una definición razonable de deuda técnica: cosas que "deberían" hacerse, pero no se hacen. sin embargo, el delta entre nuestro código y lo que deseamos que sea.

3 Considere también la cuestión de si sabemos de antemano que un proyecto tendrá una larga vida.

4 Hay algunas dudas sobre la atribución original de esta cita; el consenso parece ser que originalmente fue expresado por Brian Randell o Margaret Hamilton, pero podría haber sido totalmente inventado por Dave Parnas. La cita común es "Técnicas de ingeniería de software: Informe de una conferencia patrocinada por el Comité Científico de la OTAN", Roma, Italia, 27-31 de octubre de 1969, Bruselas, División de Asuntos Científicos, OTAN.

volvernos más eficientes a medida que crecemos, o nuestras políticas de control de versiones y estrategias de prueba nos cuestan proporcionalmente más? Los problemas de escala relacionados con la comunicación y el escalado humano se han discutido desde los primeros días de la ingeniería de software, remontándose a la *Mes del hombre mítico*.⁵ Estos problemas de escala a menudo son cuestiones de política y son fundamentales para la cuestión de la sostenibilidad del software: ¿cuánto costará hacer las cosas que necesitamos hacer repetidamente?

También podemos decir que la ingeniería de software es diferente de la programación en términos de la complejidad de las decisiones que deben tomarse y lo que está en juego. En ingeniería de software, nos vemos forzados regularmente a evaluar las compensaciones entre varios caminos a seguir, a veces con mucho en juego y, a menudo, con métricas de valor imperfectas. El trabajo de un ingeniero de software, o un líder de ingeniería de software, es apuntar a la sostenibilidad y la gestión de los costos de escalado para la organización, el producto y el flujo de trabajo de desarrollo. Con esos datos en mente, evalúe sus compensaciones y tome decisiones racionales. A veces podemos posponer los cambios de mantenimiento, o incluso adoptar políticas que no escalan bien, sabiendo que necesitaremos revisar esas decisiones. Esas opciones deben ser explícitas y claras sobre los costos diferidos.

Rara vez existe una solución única para todos en ingeniería de software, y lo mismo se aplica a este libro. Dado un factor de 100 000 para respuestas razonables sobre "¿Cuánto tiempo vivirá este software?", un rango de quizás un factor de 10 000 para "¿Cuántos ingenieros hay en su organización?" y quién sabe cuántos para "¿Cuántos hay recursos disponibles para su proyecto", es probable que la experiencia de Google no coincida con la suya. En este libro, nuestro objetivo es presentar lo que hemos descubierto que funciona para nosotros en la construcción y mantenimiento de software que esperamos dure décadas, con decenas de miles de ingenieros y recursos informáticos de alcance mundial. La mayoría de las prácticas que encontramos que son necesarias a esa escala también funcionarán bien para proyectos más pequeños: considere este informe sobre un ecosistema de ingeniería que creemos que podría ser bueno a medida que aumenta su escala. En algunos lugares, la escala súper grande tiene sus propios costos, y estaríamos más contentos de no tener que pagar gastos generales adicionales. Los llamamos como una advertencia. Con suerte, si su organización crece lo suficiente como para preocuparse por esos costos, puede encontrar una mejor respuesta.

Antes de llegar a los detalles sobre el trabajo en equipo, la cultura, las políticas y las herramientas, primero elaboraremos estos temas principales de tiempo, escala y compensaciones.

5 Frederick P. Brooks Jr. *El hombre-mes mítico: ensayos sobre ingeniería de software* (Boston: Addison-Wesley, 1995).

Tiempo y Cambio

Cuando un novato está aprendiendo a programar, la vida útil del código resultante generalmente se mide en horas o días. Las asignaciones y ejercicios de programación tienden a escribirse una sola vez, con poca o ninguna refactorización y ciertamente sin mantenimiento a largo plazo. Estos programas a menudo no se reconstruyen ni ejecutan nunca más después de su producción inicial. Esto no es sorprendente en un entorno pedagógico. Tal vez en la educación secundaria o postsecundaria, podemos encontrar un curso de proyecto en equipo o una tesis práctica. Si es así, es probable que dichos proyectos sean la única vez que el código de estudiante vivirá más de un mes más o menos. Es posible que esos desarrolladores necesiten refactorizar algún código, tal vez como respuesta a requisitos cambiantes, pero es poco probable que se les pida que se ocupen de cambios más amplios en su entorno.

También encontramos desarrolladores de código de corta duración en entornos industriales comunes. Las aplicaciones móviles a menudo tienen una vida útil bastante corta,⁶ y para bien o para mal, las reescrituras completas son relativamente comunes. Los ingenieros de una startup en etapa inicial podrían optar por centrarse en objetivos inmediatos en lugar de inversiones a largo plazo: es posible que la empresa no viva lo suficiente como para cosechar los beneficios de una inversión en infraestructura que se amortiza lentamente. Un desarrollador de inicio en serie podría tener muy razonablemente 10 años de experiencia en desarrollo y poca o ninguna experiencia en el mantenimiento de cualquier pieza de software que se espera que exista durante más de un año o dos.

En el otro extremo del espectro, algunos proyectos exitosos tienen una vida útil ilimitada: no podemos predecir razonablemente un punto final para la Búsqueda de Google, el kernel de Linux o el proyecto Apache HTTP Server. Para la mayoría de los proyectos de Google, debemos asumir que vivirán indefinidamente; no podemos predecir cuándo no necesitaremos actualizar nuestras dependencias, versiones de idioma, etc. A medida que crecen sus vidas, estos proyectos de larga duración eventualmente tienen una sensación diferente para ellos que las asignaciones de programación o el desarrollo de inicio.

Considerar Figura 1-1, que muestra dos proyectos de software en extremos opuestos de este espectro de "vida útil esperada". Para un programador que trabaja en una tarea con una vida útil esperada de horas, ¿qué tipo de mantenimiento es razonable esperar? Es decir, si sale una nueva versión de su sistema operativo mientras está trabajando en un script de Python que se ejecutará una vez, ¿debería dejar lo que está haciendo y actualizar? Por supuesto que no: la actualización no es crítica. Pero en el extremo opuesto del espectro, la Búsqueda de Google bloqueada en una versión de nuestro sistema operativo de la década de 1990 sería un problema claro.

⁶ Acelerador, “[Nada es seguro excepto la muerte, los impuestos y una vida útil corta de la aplicación móvil](#),” Desarrollador de Axway blog, 6 de diciembre de 2012.

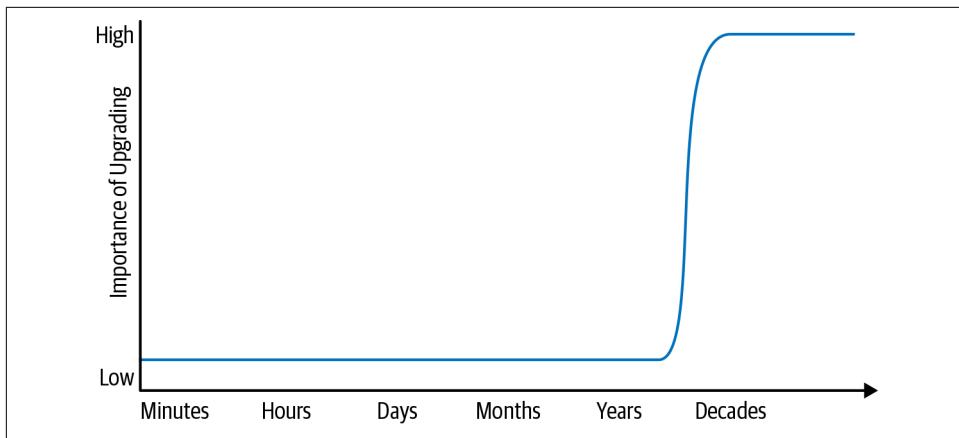


Figura 1-1. Vida útil y la importancia de las actualizaciones

Los puntos altos y bajos en el espectro de esperanza de vida sugieren que hay una transición en alguna parte. En algún punto entre un programa único y un proyecto que dura décadas, ocurre una transición: un proyecto debe comenzar a reaccionar ante las externalidades cambiantes.⁷ Para cualquier proyecto que no planeó actualizaciones desde el principio, esa transición probablemente sea muy dolorosa por tres razones, cada una de las cuales agrava a las demás:

- Está realizando una tarea que aún no se ha realizado para este proyecto; se han incorporado suposiciones más ocultas.
- Es menos probable que los ingenieros que intentan realizar la actualización tengan experiencia en este tipo de tareas.
- El tamaño de la actualización suele ser más grande de lo habitual, y realiza actualizaciones de varios años a la vez en lugar de una actualización más incremental.

Y, por lo tanto, después de pasar por una actualización de este tipo una vez (o renunciar a la mitad), es bastante razonable sobreestimar el costo de hacer una actualización posterior y decidir "Nunca más". Las empresas que llegan a esta conclusión terminan comprometiéndose a tirar cosas y reescribir su código, o decidir no volver a actualizar nunca más. En lugar de adoptar el enfoque natural evitando una tarea dolorosa, a veces la respuesta más responsable es invertir en hacerla menos dolorosa. Todo depende del costo de su actualización, el valor que proporciona y la vida útil esperada del proyecto en cuestión.

⁷ Sus propias prioridades y gustos informarán dónde ocurre exactamente esa transición. Hemos encontrado que la mayoría los proyectos parecen estar dispuestos a actualizarse dentro de cinco años. En algún lugar entre 5 y 10 años parece una estimación conservadora para esta transición en general.

Superar no solo la primera gran actualización, sino llegar al punto en el que pueda mantenerse actualizado de manera confiable en el futuro, es la esencia de la sostenibilidad a largo plazo para su proyecto. La sostenibilidad requiere planificar y gestionar el impacto del cambio requerido. Para muchos proyectos en Google, creemos que hemos logrado este tipo de sostenibilidad, en gran medida a través de prueba y error.

Entonces, concretamente, ¿en qué se diferencia la programación a corto plazo de la producción de código con una vida útil esperada mucho más larga? Con el tiempo, debemos ser mucho más conscientes de la diferencia entre "sucede que funciona" y "es mantenible". No existe una solución perfecta para identificar estos problemas. Eso es desafortunado, porque mantener el software mantenible a largo plazo es una batalla constante.

Ley de Hyrum

Si está manteniendo un proyecto que utilizan otros ingenieros, la lección más importante sobre "funciona" frente a "es mantenible" es lo que llamamos *Ley de Hyrum*:

Con un número suficiente de usuarios de una API, no importa lo que prometa en el contrato: alguien dependerá de todos los comportamientos observables de su sistema.

En nuestra experiencia, este axioma es un factor dominante en cualquier discusión sobre cambios de software a lo largo del tiempo. Es conceptualmente similar a la entropía: las discusiones sobre el cambio y el mantenimiento a lo largo del tiempo deben tener en cuenta la Ley de Hyrum.⁸ así como las discusiones sobre eficiencia o termodinámica deben tener en cuenta la entropía. El hecho de que la entropía nunca disminuya no significa que no debamos tratar de ser eficientes. El hecho de que la Ley de Hyrum se aplique al mantener el software no significa que no podamos planificarlo o tratar de comprenderlo mejor. Podemos mitigarlo, pero sabemos que nunca podrá ser erradicado.

La Ley de Hyrum representa el conocimiento práctico de que, incluso con las mejores intenciones, los mejores ingenieros y prácticas sólidas para la revisión del código, no podemos asumir el cumplimiento perfecto de los contratos publicados o las mejores prácticas. Como propietario de una API, obtendrá *alguna* flexibilidad y libertad al ser claro sobre las promesas de la interfaz, pero en la práctica, la complejidad y dificultad de un cambio dado también depende de qué tan útil encuentre un usuario algún comportamiento observable de su API. Si los usuarios no pueden depender de tales cosas, su API será fácil de cambiar. Con suficiente tiempo y suficientes usuarios, incluso el cambio más inocuo *voluntadromper algo*,⁹ su análisis del valor de ese cambio debe incorporar la dificultad de investigar, identificar y resolver esas fallas.

⁸ Para su crédito, Hyrum se esforzó mucho en llamar humildemente a esto "La ley de las dependencias implícitas", pero "la ley de Hyrum Law" es la forma abreviada que la mayoría de la gente de Google ha elegido.

⁹ Véase "flujode trabajo," un *xkcd* cómic.

Ejemplo: pedido de hash

Consideré el ejemplo de ordenamiento de iteraciones hash. Si insertamos cinco elementos en un conjunto basado en hash, ¿en qué orden los sacamos?

```
>>> porien{"manzana","plátano","zanahoria","durián","berenjena":imprimir(i)
...
durián
zanahoria
manzana
berenjena
plátano
```

La mayoría de los programadores saben que las tablas hash no están ordenadas de manera obvia. Pocos saben los detalles de si la tabla hash particular que están usando es con la intención para proporcionar ese pedido en particular para siempre. Esto puede parecer anodino, pero durante la última década o dos, la experiencia de la industria informática con el uso de estos tipos ha evolucionado:

- *Inundación de hash*: Los ataques proporcionan un mayor incentivo para la iteración de hash no determinista.
- Las posibles ganancias de eficiencia de la investigación de algoritmos de hash mejorados o contenedores de hash requieren cambios en el orden de iteración de hash.
- Según la Ley de Hyrum, los programadores escribirán programas que dependen del orden en que se recorre una tabla hash, si tienen la capacidad de hacerlo.

Como resultado, si le pregunta a cualquier experto "¿Puedo asumir una secuencia de salida particular para mi contenedor de hash?" ese experto probablemente dirá "No". En general eso es correcto, pero quizás simplista. Una respuesta más matizada es: "Si su código es de corta duración, sin cambios en su hardware, tiempo de ejecución del lenguaje o elección de estructura de datos, tal suposición está bien. Si no sabe cuánto tiempo vivirá su código, o si no puede prometer que nunca cambiará nada de lo que dependa, tal suposición es incorrecta". Además, incluso si su propia implementación no depende del orden del contenedor hash, podría ser utilizado por otro código que crea implícitamente tal dependencia. Por ejemplo, si su biblioteca serializa valores en una respuesta de llamada de procedimiento remoto (RPC), la persona que llama RPC podría terminar dependiendo del orden de esos valores.

Este es un ejemplo muy básico de la diferencia entre "funciona" y "es correcto". Para un programa de corta duración, dependiendo del orden de iteración de sus contenedores no causará ningún problema técnico. Para un proyecto de ingeniería de software, por otro lado, tal confianza en un orden definido es un riesgo—dado el tiempo suficiente, algo sucederá.

10 Un tipo de ataque de denegación de servicio (DoS) en el que un usuario que no es de confianza conoce la estructura de una tabla hash y la función hash y proporciona datos de tal manera que degrada el rendimiento algorítmico de las operaciones en la tabla.

haga que sea valioso cambiar ese orden de iteración. Ese valor puede manifestarse de varias maneras, ya sea eficiencia, seguridad o simplemente preparar la estructura de datos para el futuro para permitir cambios futuros. Cuando ese valor quede claro, deberá sopesar las compensaciones entre ese valor y el dolor de romper con sus desarrolladores o clientes.

Algunos lenguajes aleatorizan específicamente el orden de hash entre versiones de la biblioteca o incluso entre la ejecución del mismo programa en un intento de evitar dependencias. Pero incluso esto todavía permite algunas sorpresas de la Ley de Hyrum: hay un código que usa el orden de iteración hash como un generador de números aleatorios ineficiente. Eliminar tal aleatoriedad ahora rompería a esos usuarios. Así como la entropía aumenta en todos los sistemas termodinámicos, la Ley de Hyrum se aplica a todos los comportamientos observables.

Pensando en las diferencias entre el código escrito con una mentalidad de “funciona ahora” y “funciona indefinidamente”, podemos extraer algunas relaciones claras. Al considerar el código como un artefacto con un requisito de vida útil (altamente) variable, podemos comenzar a clasificar los estilos de programación: el código que depende de características frágiles y no publicadas de sus dependencias probablemente se describa como “hacky” o “inteligente”. ”, mientras que el código que sigue las mejores prácticas y se ha planificado para el futuro es más probable que se describa como “limpio” y “mantenible”. Ambos tienen sus propósitos, pero el que seleccione depende de manera crucial de la vida útil esperada del código en cuestión. Nos hemos acostumbrado a decir: “Es *programación*si ‘inteligente’ es un cumplido, pero es *Ingeniería de software*si ‘inteligente’ es una acusación.”

¿Por qué no apuntar simplemente a “nada cambia”?

Implícita en toda esta discusión sobre el tiempo y la necesidad de reaccionar al cambio está la suposición de que el cambio podría ser necesario. ¿Lo es?

Como con todo lo demás en este libro, depende. Nos comprometemos fácilmente a “Para la mayoría de los proyectos, durante un período de tiempo lo suficientemente largo, es posible que sea necesario cambiar todo lo que hay debajo de ellos”. Si tiene un proyecto escrito en C puro sin dependencias externas (o solo dependencias externas que prometen una gran estabilidad a largo plazo, como POSIX), es posible que pueda evitar cualquier forma de refactorización o actualización difícil. C hace un gran trabajo al proporcionar estabilidad; en muchos aspectos, ese es su propósito principal.

La mayoría de los proyectos están mucho más expuestos a cambios en la tecnología subyacente. La mayoría de los lenguajes de programación y tiempos de ejecución cambian mucho más que C. Incluso las bibliotecas implementadas en C puro pueden cambiar para admitir nuevas funciones, lo que puede afectar a los usuarios intermedios. Los problemas de seguridad se revelan en todo tipo de tecnología, desde procesadores hasta bibliotecas de redes y código de aplicación. *Todas* La pieza de tecnología de la que depende su proyecto tiene algún riesgo (con suerte, pequeño) de contener errores críticos y vulnerabilidades de seguridad que podrían salir a la luz solo después de que haya comenzado a confiar en ella. Si no puede implementar un parche para [Heartbleed](#) mitigar

problemas de ejecución especulativa como fusión y espectro porque has asumido (o prometido) que nada cambiará nunca, es una apuesta importante.

Las mejoras en la eficiencia complican aún más el panorama. Queremos equipar nuestros centros de datos con equipos informáticos rentables, especialmente mejorando la eficiencia de la CPU. Sin embargo, los algoritmos y las estructuras de datos de los primeros días de Google son simplemente menos eficientes en los equipos modernos: una lista enlazada o un árbol de búsqueda binario seguirán funcionando bien, pero la brecha cada vez mayor entre los ciclos de CPU y la latencia de la memoria afecta lo que es "eficiente". parece el código. Con el tiempo, el valor de actualizar a un hardware más nuevo puede disminuir sin cambios de diseño acompañantes en el software. La compatibilidad con versiones anteriores garantiza que los sistemas más antiguos sigan funcionando, pero eso no garantiza que las optimizaciones antiguas sigan siendo útiles. No estar dispuesto o no poder aprovechar tales oportunidades corre el riesgo de incurrir en grandes costos. Las preocupaciones de eficiencia como esta son particularmente sutiles: el diseño original podría haber sido perfectamente lógico y seguir las mejores prácticas razonables. Solo después de una evolución de cambios compatibles con versiones anteriores, una opción nueva y más eficiente se vuelve importante. No se cometieron errores, pero el paso del tiempo hizo que el cambio fuera valioso.

Inquietudes como las que acabamos de mencionar explican por qué existen grandes riesgos para los proyectos a largo plazo que no han invertido en sostenibilidad. Debemos ser capaces de responder a este tipo de problemas y aprovechar estas oportunidades, independientemente de si nos afectan directamente o se manifiestan solo en el cierre transitivo de la tecnología sobre la que construimos. El cambio no es inherentemente bueno. No deberíamos cambiar por el simple hecho de cambiar. Pero necesitamos ser capaces de cambiar. Si permitimos esa eventual necesidad, también deberíamos considerar si invertir en hacer que esa capacidad sea barata. Como todo administrador de sistemas sabe, una cosa es saber en teoría que se puede recuperar desde cinta, y otra saber en la práctica exactamente cómo hacerlo y cuánto costará cuando sea necesario.

Escala y eficiencia

Como se señaló en el *Ingeniería de confiabilidad del sitio* (SRE) libro,¹¹ El sistema de producción de Google en su conjunto se encuentra entre las máquinas más complejas creadas por la humanidad. La complejidad involucrada en construir una máquina de este tipo y mantenerla funcionando sin problemas ha requerido innumerables horas de reflexión, debate y rediseño por parte de expertos de nuestra organización y de todo el mundo. Entonces, ya hemos escrito un libro sobre la complejidad de mantener esa máquina funcionando a esa escala.

¹¹ Beyer, B. et al. *Ingeniería de confiabilidad del sitio: cómo Google ejecuta los sistemas de producción*. (Boston: O'Reilly Media, 2016).

Gran parte de estoEl libro se centra en la complejidad de la escala de la organización que produce una máquina de este tipo y los procesos que utilizamos para mantener esa máquina en funcionamiento a lo largo del tiempo. Considere nuevamente el concepto de sustentabilidad de la base de código: "La base de código de su organización *essostenible*cuando está *capaz*para cambiar todas las cosas que debe cambiar, de forma segura, y puede hacerlo durante la vida de su base de código". Oculto en la discusión de la capacidad también está el de los costos: si cambiar algo tiene un costo desmesurado, es probable que se posponga. Si los costos crecen superlinealmente con el tiempo, la operación claramente no es escalable.¹²Eventualmente, el tiempo tomará control y surgirá algo inesperado que absolutamente debes cambiar. Cuando su proyecto duplique el alcance y necesite realizar esa tarea nuevamente, ¿requerirá el doble de mano de obra? ¿Tendrá los recursos humanos necesarios para abordar el problema la próxima vez?

Los costos humanos no son el único recurso finito que necesita escalar. Así como el software en sí necesita escalar bien con recursos tradicionales como cómputo, memoria, almacenamiento y ancho de banda, el desarrollo de ese software también necesita escalar, tanto en términos de participación del tiempo humano como de los recursos de cómputo que impulsan su flujo de trabajo de desarrollo. Si el costo de cómputo para su clúster de prueba crece superlinealmente, consumiendo más recursos de cómputo por persona cada trimestre, está en un camino insostenible y necesita hacer cambios pronto.

Finalmente, el activo máspreciado de una organización de software, la base de código en sí, también necesita escalar. Si su sistema de compilación o sistema de control de versiones se escala superlinealmente con el tiempo, tal vez como resultado del crecimiento y el aumento del historial de cambios, podría llegar a un punto en el que simplemente no pueda continuar. Muchas preguntas, como "¿Cuánto tiempo se tarda en hacer una compilación completa?", "¿Cuánto tiempo se tarda en obtener una copia nueva del repositorio?" o "¿Cuánto costará actualizar a una nueva? versión de idioma? no se controlan activamente y cambian a un ritmo lento. Fácilmente pueden volverse como el *rana hervida metafórica*; es demasiado fácil que los problemas empeoren lentamente y nunca se manifiesten como un momento singular de crisis. Solo con la conciencia y el compromiso de escalar de toda la organización es probable que se mantenga al tanto de estos problemas.

Todo en lo que se base su organización para producir y mantener el código debe ser escalable en términos de costo general y consumo de recursos. En particular, todo lo que su organización debe hacer repetidamente debe ser escalable en términos de esfuerzo humano. Muchas políticas comunes no parecen ser escalables en este sentido.

Políticas que no escalan

Con un poco de práctica, se vuelve más fácil detectar políticas con malas propiedades de escalado. Más comúnmente, estos pueden identificarse considerando el trabajo impuesto a un solo

¹² Cada vez que usamos "escalable" en un contexto informal en este capítulo, nos referimos a "escalamiento sublineal con respecto a interacciones humanas".

ingeniero e imaginando la organización escalando por 10 o 100 veces. Cuando seamos 10 veces más grandes, ¿agregaremos 10 veces más trabajo con el que nuestro ingeniero de muestras necesita mantenerse al día? ¿Cree la cantidad de trabajo que nuestro ingeniero debe realizar en función del tamaño de la organización? ¿El trabajo se amplía con el tamaño del código base? Si alguno de estos es cierto, ¿tenemos algún mecanismo para automatizar u optimizar ese trabajo? Si no, tenemos problemas de escalado.

Consideré un enfoque tradicional de la desaprobación. Discutimos la desaprobación mucho más en [Capítulo 15](#), pero el enfoque común de la desaprobación sirve como un gran ejemplo de problemas de escalado. Se ha desarrollado un nuevo Widget. Se toma la decisión de que todos deben usar el nuevo y dejar de usar el anterior. Para motivar esto, los líderes del proyecto dicen: "Eliminaremos el Widget anterior el 15 de agosto; asegúrese de haberse convertido al nuevo Widget".

Este tipo de enfoque podría funcionar en una configuración de software pequeña, pero falla rápidamente a medida que aumenta la profundidad y la amplitud del gráfico de dependencia. Los equipos dependen de un número cada vez mayor de Widgets, y una sola interrupción en la compilación puede afectar a un porcentaje cada vez mayor de la empresa. Resolver estos problemas de manera escalable significa cambiar la forma en que hacemos la desaprobación: en lugar de enviar el trabajo de migración a los clientes, los equipos pueden internalizarlo ellos mismos, con todas las economías de escala que eso proporciona.

En 2012, tratamos de poner fin a esto con reglas que mitigan la rotación: los equipos de infraestructura deben hacer el trabajo de mover a sus usuarios internos a nuevas versiones ellos mismos o hacer la actualización en el lugar, de manera compatible con versiones anteriores. Esta política, a la que llamamos la "Regla de abandono", escala mejor: los proyectos dependientes ya no gastan un esfuerzo cada vez mayor solo para mantenerse al día. También aprendimos que tener un grupo dedicado de expertos ejecuta las escalas de cambio mejor que pedir más esfuerzo de mantenimiento a cada usuario: los expertos pasan algún tiempo aprendiendo todo el problema en profundidad y luego aplican esa experiencia a cada subproblema. Obligar a los usuarios a responder a la rotación significa que cada equipo afectado hace un peor trabajo al aumentar, resuelve su problema inmediato y luego desecha ese conocimiento ahora inútil. La experiencia escala mejor.

El uso tradicional de ramas de desarrollo es otro ejemplo de política que tiene problemas de escala incorporados. Una organización podría identificar que la fusión de funciones grandes en el tronco ha desestabilizado el producto y concluir: "Necesitamos controles más estrictos sobre cuándo se fusionan las cosas. Deberíamos fusionarnos con menos frecuencia". Esto lleva rápidamente a que cada equipo o cada característica tenga ramas de desarrollo separadas. Cada vez que se decide que una rama está "completa", se prueba y se fusiona con el tronco, lo que desencadena un trabajo potencialmente costoso para otros ingenieros que aún trabajan en su rama de desarrollo, en forma de sincronización y prueba. Esta gestión de sucursales se puede hacer funcionar para una organización pequeña que haga malabares con 5 a 10 sucursales de este tipo. A medida que aumenta el tamaño de una organización (y el número de sucursales), rápidamente se hace evidente que estamos pagando una cantidad cada vez mayor de gastos generales para hacer la misma tarea. Necesitaremos un enfoque diferente a medida que aumentemos la escala, y lo discutimos en [capítulo 16](#).

Políticas que escalan bien

¿Qué tipo de políticas dan como resultado mejores costos a medida que crece la organización? O, mejor aún, ¿qué tipo de políticas podemos implementar que proporcionen un valor superlineal a medida que crece la organización?

Una de nuestras políticas internas favoritas es un gran facilitador de los equipos de infraestructura, protegiendo su capacidad para realizar cambios en la infraestructura de manera segura. "Si un producto experimenta interrupciones u otros problemas como resultado de cambios en la infraestructura, pero el problema no salió a la luz en las pruebas de nuestro sistema de integración continua (CI), no es culpa del cambio de infraestructura". Más coloquialmente, esto se expresa como "Si te gustó, deberías haberle hecho una prueba de CI", a lo que llamamos "La regla de Beyoncé".¹³ Desde una perspectiva de escalamiento, la regla de Beyoncé implica que las pruebas personalizadas únicas y complicadas que no son activadas por nuestro sistema común de IC no cuentan. Sin esto, un ingeniero en un equipo de infraestructura posiblemente necesite rastrear a cada equipo con cualquier código afectado y preguntarles cómo ejecutar sus pruebas. Podríamos hacer eso cuando hubiera cien ingenieros. Definitivamente no podemos darnos el lujo de hacer eso nunca más.

Descubrimos que la experiencia y los foros de comunicación compartida ofrecen un gran valor a medida que una organización crece. A medida que los ingenieros discuten y responden preguntas en foros compartidos, el conocimiento tiende a difundirse. Crecen nuevos expertos. Si tiene cien ingenieros escribiendo Java, un solo experto en Java amable y servicial dispuesto a responder preguntas pronto producirá cien ingenieros escribiendo mejor código Java. El conocimiento es viral, los expertos son portadores y hay mucho que decir sobre el valor de eliminar los obstáculos comunes para sus ingenieros. Cubrimos esto con mayor detalle en [Capítulo 3](#).

Ejemplo: actualización del compilador

Considere la abrumadora tarea de actualizar su compilador. Teóricamente, una actualización del compilador debería ser barata dado el esfuerzo que requieren los lenguajes para ser compatibles con versiones anteriores, pero ¿qué tan barata es la operación en la práctica? Si nunca antes ha realizado una actualización de este tipo, ¿cómo evaluaría si su base de código es compatible con ese cambio?

¹³ Esta es una referencia a la popular canción "Single Ladies", que incluye el estribillo "Si te gustó entonces Debería haberle puesto un anillo.

Según nuestra experiencia, las actualizaciones de lenguajes y compiladores son tareas sútiles y difíciles, incluso cuando se espera que sean compatibles con versiones anteriores. Una actualización del compilador casi siempre dará como resultado cambios menores en el comportamiento: corregir errores de compilación, ajustar optimizaciones o cambiar potencialmente los resultados de cualquier cosa que no haya sido definida previamente. ¿Cómo evaluaría la corrección de todo su código base frente a todos estos posibles resultados?

La actualización del compilador más famosa en la historia de Google tuvo lugar en 2006. En ese momento, habíamos estado operando durante algunos años y teníamos varios miles de ingenieros en el personal. No habíamos actualizado los compiladores en unos cinco años. La mayoría de nuestros ingenieros no tenían experiencia con un cambio de compilador. La mayor parte de nuestro código había estado expuesto a una sola versión del compilador. Fue una tarea difícil y dolorosa para un equipo de (en su mayoría) voluntarios, que finalmente se convirtió en una cuestión de encontrar atajos y simplificaciones para solucionar los cambios de lenguaje y compilador ascendente que no sabíamos cómo adoptar.¹⁴ Al final, la actualización del compilador de 2006 fue extremadamente dolorosa.

Muchos problemas de la Ley de Hyrum, grandes y pequeños, se infiltraron en el código base y sirvieron para profundizar nuestra dependencia de una versión particular del compilador. Romper esas dependencias implícitas fue doloroso. Los ingenieros en cuestión se estaban arriesgando: todavía no teníamos la Regla de Beyoncé, ni teníamos un sistema de CI generalizado, por lo que era difícil saber el impacto del cambio con anticipación o estar seguros de que no lo harían. culpados de las regresiones.

Esta historia no es nada inusual. Los ingenieros de muchas empresas pueden contar una historia similar sobre una actualización dolorosa. Lo que es inusual es que nos dimos cuenta después del hecho de que la tarea había sido dolorosa y comenzamos a centrarnos en la tecnología y los cambios organizacionales para superar los problemas de escala y convertir la escala en nuestra ventaja: automatización (para que un solo ser humano pueda hacer más), consolidación/ consistencia (para que los cambios de bajo nivel tengan un alcance de problema limitado) y experiencia (para que unos pocos humanos puedan hacer más).

Cuanto más frecuentemente cambie su infraestructura, más fácil será hacerlo. Descubrimos que la mayoría de las veces, cuando el código se actualiza como parte de algo como una actualización del compilador, se vuelve menos frágil y más fácil de actualizar en el futuro. En un ecosistema en el que la mayoría del código ha pasado por varias actualizaciones, se detiene dependiendo de los matices de la implementación subyacente; en cambio, depende de la abstracción real garantizada por el lenguaje o el sistema operativo. Independientemente de lo que esté actualizando exactamente, espere que la primera actualización para una base de código sea significativamente más costosa que las actualizaciones posteriores, incluso controlando otros factores.

¹⁴ Específicamente, era necesario hacer referencia a las interfaces de la biblioteca estándar de C++ en el espacio de nombres std, y una opción cambio de mización paraestándar::cadenaresultó ser una pesimización significativa para nuestro uso, por lo que requirió algunas soluciones adicionales.

A través de esta y otras experiencias, hemos descubierto muchos factores que afectan la flexibilidad de una base de código:

Pericia

Sabemos cómo hacer esto; para algunos idiomas, ahora hemos realizado cientos de actualizaciones del compilador en muchas plataformas.

Estabilidad

Hay menos cambios entre lanzamientos porque adoptamos lanzamientos más regularmente; para algunos lenguajes, ahora implementamos actualizaciones del compilador cada semana o dos.

Conformidad

Hay menos código que aún no ha pasado por una actualización, nuevamente porque estamos actualizando regularmente.

Familiaridad

Debido a que hacemos esto con la suficiente regularidad, podemos detectar redundancias en el proceso de realizar una actualización e intentar automatizar. Esto se superpone significativamente con las opiniones de SRE sobre el trabajo.¹⁵

Política

Tenemos procesos y políticas como la Regla Beyoncé. El efecto neto de estos procesos es que las actualizaciones siguen siendo factibles porque los equipos de infraestructura no necesitan preocuparse por todos los usos desconocidos, solo por los que son visibles en nuestros sistemas de CI.

La lección subyacente no es sobre la frecuencia o la dificultad de las actualizaciones del compilador, sino que tan pronto como nos dimos cuenta de que las tareas de actualización del compilador eran necesarias, encontramos formas de asegurarnos de realizar esas tareas con un número constante de ingenieros, incluso como la base de código. creció.¹⁶ Si, en cambio, hubiéramos decidido que la tarea era demasiado costosa y debería evitarse en el futuro, aún podríamos estar usando una versión del compilador de hace una década. Estaríamos pagando quizás un 25 % más por los recursos informáticos como resultado de las oportunidades de optimización perdidas. Nuestra infraestructura central podría ser vulnerable a importantes riesgos de seguridad dado que un compilador de la era de 2006 ciertamente no está ayudando a mitigar las vulnerabilidades de ejecución especulativa. El estancamiento es una opción, pero a menudo no es sabia.

¹⁵Beyer et al. *Ingeniería de confiabilidad del sitio: cómo Google ejecuta los sistemas de producción*, Capítulo 5, "Eliminación del esfuerzo".

¹⁶ Según nuestra experiencia, un ingeniero de software promedio (SWE) produce una cantidad bastante constante de líneas de código por unidad de tiempo. Para una población de SWE fija, una base de código crece de forma lineal, proporcional al recuento de SWEmonths a lo largo del tiempo. Si sus tareas requieren un esfuerzo que escala con líneas de código, eso es preocupante.

Desplazamiento a la izquierda

Una de las verdades generales que hemos visto que es cierta es la idea de que encontrar problemas antes en el flujo de trabajo del desarrollador generalmente reduce los costos. Considere una línea de tiempo del flujo de trabajo del desarrollador para una característica que progresó de izquierda a derecha, comenzando desde la concepción y el diseño, progresando a través de la implementación, revisión, prueba, compromiso, control y eventual implementación de producción. Cambiar la detección de problemas a la "izquierda" antes en esta línea de tiempo hace que sea más económico solucionarlo que esperar más tiempo, como se muestra en [Figura 1-2](#).

Este término parece haberse originado a partir de argumentos de que la seguridad no debe aplazarse hasta el final del proceso de desarrollo, con llamados necesarios para "cambiar a la izquierda en seguridad". El argumento en este caso es relativamente simple: si se descubre un problema de seguridad solo después de que su producto haya pasado a producción, tiene un problema muy costoso. Si se detecta antes de la implementación en producción, aún puede llevar mucho trabajo identificar y solucionar el problema, pero es más económico. Si puede detectarlo antes de que el desarrollador original envíe la falla al control de versiones, es aún más barato: ya conocen la función; revisar de acuerdo con las nuevas restricciones de seguridad es más barato que comprometerse y obligar a otra persona a clasificarlo y arreglarlo.

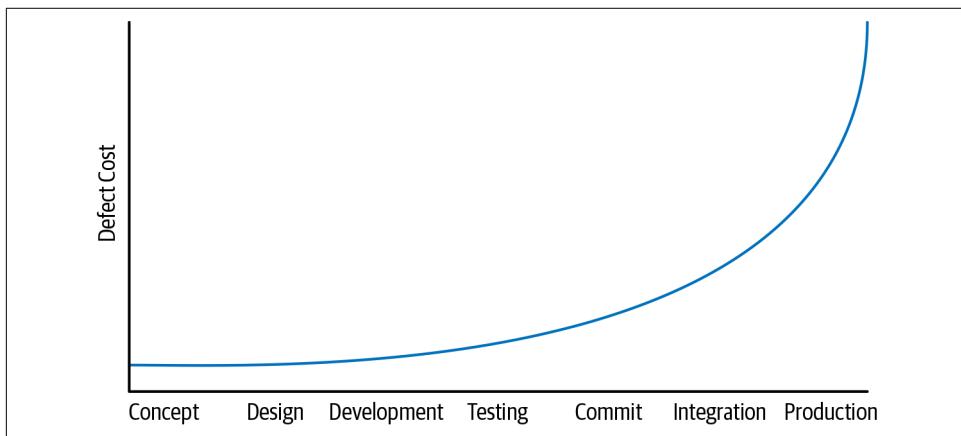


Figura 1-2. Cronología del flujo de trabajo del desarrollador

El mismo patrón básico surge muchas veces en este libro. Los errores que se detectan mediante el análisis estático y la revisión del código antes de que se confirmen son mucho más baratos que los errores que llegan a producción. Proporcionar herramientas y prácticas que destaque la calidad, la confiabilidad y la seguridad en las primeras etapas del proceso de desarrollo es un objetivo común para muchos de nuestros equipos de infraestructura. No es necesario que ningún proceso o herramienta sea perfecto, por lo que podemos asumir un enfoque de defensa en profundidad, con la esperanza de detectar tantos defectos en el lado izquierdo del gráfico como sea posible.

Compensaciones y costos

Si entendemos cómo programar, entendemos la vida útil del software que mantenemos y entendemos cómo mantenerlo a medida que crecemos con más ingenieros que producen y mantienen nuevas funciones, todo lo que queda es tomar buenas decisiones. Esto parece obvio: en ingeniería de software, como en la vida, las buenas elecciones conducen a buenos resultados. Sin embargo, las ramificaciones de esta observación se pasan por alto fácilmente. Dentro de Google, hay un fuerte disgusto por "porque yo lo digo". Es importante que haya un decisor para cualquier tema y claros caminos de escalada cuando las decisiones parecen estar equivocadas, pero el objetivo es el consenso, no la unanimidad. Está bien y se espera ver algunos casos de "No estoy de acuerdo con sus métricas/valoración, pero veo cómo puede llegar a esa conclusión. Inherente a todo esto está la idea de que debe haber una razón para todo; "solo porque", "porque yo lo digo" o "porque todos los demás lo hacen de esta manera" son lugares donde acechan las malas decisiones. Siempre que sea eficiente hacerlo, deberíamos poder explicar nuestro trabajo al decidir entre los costos generales para dos opciones de ingeniería.

¿Qué entendemos por costo? No solo estamos hablando de dólares aquí. "Costo" se traduce aproximadamente como esfuerzo y puede involucrar cualquiera o todos estos factores:

- Costos financieros (por ejemplo, dinero)
- Costos de recursos (por ejemplo, tiempo de CPU)
- Costos de personal (p. ej., esfuerzo de ingeniería)
- Costos de transacción (p. ej., ¿cuánto cuesta tomar medidas?)
- Costos de oportunidad (p. ej., ¿cuánto cuesta no tomar medidas?)
- Costos sociales (p. ej., ¿qué impacto tendrá esta elección en la sociedad en general?)

Históricamente, ha sido particularmente fácil ignorar la cuestión de los costos sociales. Sin embargo, Google y otras grandes empresas tecnológicas ahora pueden implementar productos de manera creíble con miles de millones de usuarios. En muchos casos, estos productos son un claro beneficio neto, pero cuando operamos a tal escala, incluso las pequeñas discrepancias en la usabilidad, la accesibilidad, la equidad o la posibilidad de abuso se magnifican, a menudo en detrimento de grupos que ya están marginados. El software impregna tantos aspectos de la sociedad y la cultura; por lo tanto, es prudente que seamos conscientes de lo bueno y lo malo que permitimos al tomar decisiones técnicas y de productos. Discutimos esto mucho más en [Capítulo 4](#).

Además de los costos antes mencionados (o nuestra estimación de ellos), existen sesgos: sesgo de statu quo, aversión a las pérdidas y otros. Cuando evaluamos el costo, debemos tener en cuenta todos los costos enumerados anteriormente: la salud de una organización no es solo si hay dinero en el banco, sino también si sus miembros se sienten valorados y productivos. En campos altamente creativos y lucrativos como la ingeniería de software, el costo financiero no suele ser el factor limitante, sino el costo del personal. Ganancias de eficiencia de

mantener a los ingenieros contentos, enfocados y comprometidos puede dominar fácilmente otros factores, simplemente porque el enfoque y la productividad son muy variables, y es fácil imaginar una diferencia del 10 al 20 %.

Ejemplo: Marcadores

En muchas organizaciones, los marcadores de pizarra se tratan como bienes preciosos. Están estrictamente controlados y siempre escasean. Invariablemente, la mitad de los marcadores en cualquier pizarra están secos e inutilizables. ¿Con qué frecuencia ha estado en una reunión que se vio interrumpida por la falta de un marcador de trabajo? ¿Con qué frecuencia ha tenido que descarrilar su línea de pensamiento por haberse quedado sin marcador? ¿Con qué frecuencia han desaparecido todos los marcadores, presumiblemente porque otro equipo se quedó sin marcadores y tuvo que fugarse con el suyo? Todo por un producto que cuesta menos de un dólar.

Google tiende a tener armarios abiertos llenos de suministros de oficina, incluidos marcadores de pizarra, en la mayoría de las áreas de trabajo. Con un aviso de un momento, es fácil agarrar docenas de marcadores en una variedad de colores. En algún lugar a lo largo de la línea hicimos una compensación explícita: es mucho más importante optimizar para una lluvia de ideas libre de obstáculos que protegerse contra alguien que deambula con un montón de marcadores.

Nuestro objetivo es tener el mismo nivel de evaluación abierta y explícita de las ventajas y desventajas de costo/beneficio involucradas en todo lo que hacemos, desde suministros de oficina y beneficios para empleados a través de la experiencia diaria para desarrolladores hasta cómo aprovisionar y ejecutar a escala global servicios. A menudo decimos: "Google es una cultura basada en datos". De hecho, eso es una simplificación: incluso cuando no hay *datos*, todavía podría haber *evidencia*, *precedente*, y *argumento*. Tomar buenas decisiones de ingeniería consiste en sopesar todas las entradas disponibles y tomar decisiones informadas sobre las compensaciones. A veces, esas decisiones se basan en el instinto o en las mejores prácticas aceptadas, pero solo después de haber agotado los enfoques que intentan medir o estimar los verdaderos costos subyacentes.

Al final, las decisiones en un grupo de ingeniería deben reducirse a muy pocas cosas:

- Estamos haciendo esto porque debemos hacerlo (requisitos legales, requisitos del cliente).
- Estamos haciendo esto porque es la mejor opción (según lo determinado por algún decisor apropiado) que podemos ver en ese momento, según la evidencia actual.

Las decisiones no deben ser "Estamos haciendo esto porque yo lo digo".¹⁷

17 Esto no quiere decir que las decisiones deban tomarse por unanimidad, o incluso con un amplio consenso; en el final, alguien debe ser el decisor. Esta es principalmente una declaración de cómo debe fluir el proceso de toma de decisiones para quien sea realmente responsable de la decisión.

Entradas para la toma de decisiones

Cuando estamos pesando datos, encontramos dos escenarios comunes:

- Todas las cantidades involucradas son medibles o al menos pueden estimarse. Esto generalmente significa que estamos evaluando las ventajas y desventajas entre CPU y red, o dólares y RAM, o considerando si invertir dos semanas de tiempo de ingeniería para ahorrar *norteCPU* en nuestros centros de datos.
- Algunas de las cantidades son sutiles o no sabemos cómo medirlas. A veces esto se manifiesta como "No sabemos cuánto tiempo de ingeniería llevará esto". A veces es aún más confuso: ¿cómo se mide el costo de ingeniería de una API mal diseñada? ¿O el impacto social de la elección de un producto?

Hay pocas razones para ser deficiente en el primer tipo de decisión. Cualquier organización de ingeniería de software puede y debe realizar un seguimiento del costo actual de los recursos informáticos, las horas de ingeniería y otras cantidades con las que interactúa regularmente. Incluso si no desea dar a conocer a su organización las cantidades exactas en dólares, aún puede generar una tabla de conversión: esta cantidad de CPU cuesta lo mismo que tanta RAM o tanto ancho de banda de red.

Con una tabla de conversión acordada en la mano, cada ingeniero puede hacer su propio análisis. "Si dedico dos semanas a cambiar esta lista vinculada a una estructura de mayor rendimiento, usaré cinco gibibytes más de RAM de producción pero ahorraré dos mil CPU. ¿Debería hacerlo?" Esta pregunta no solo depende del costo relativo de RAM y CPU, sino también de los costos de personal (dos semanas de soporte para un ingeniero de software) y los costos de oportunidad (¿qué más podría producir ese ingeniero en dos semanas?).

Para el segundo tipo de decisión, no hay una respuesta fácil. Confiamos en la experiencia, el liderazgo y los precedentes para negociar estos temas. Estamos invirtiendo en investigación para ayudarnos a cuantificar lo difícil de cuantificar (ver [Capítulo 7](#)). Sin embargo, la mejor sugerencia amplia que tenemos es ser conscientes de que no todo es medible o predecible e intentar tratar tales decisiones con la misma prioridad y mayor cuidado. A menudo son igual de importantes, pero más difíciles de manejar.

Ejemplo: compilaciones distribuidas

Considere su construcción. De acuerdo con encuestas de Twitter completamente a científicas, entre el 60 y el 70 % de los desarrolladores construyen localmente, incluso con las construcciones grandes y complicadas de la actualidad. Esto lleva directamente a bromas como lo ilustra [este cómic "Compilando"](#). ¿Cuánto tiempo productivo se pierde en su organización esperando una compilación? Compare eso con el costo de ejecutar algo como `distcc` para un grupo pequeño. O, ¿cuánto cuesta administrar una granja de construcción pequeña para un grupo grande? ¿Cuántas semanas/meses se necesitan para que esos costos sean una ganancia neta?

A mediados de la década de 2000, Google se basaba exclusivamente en un sistema de compilación local: revisaba el código y lo compilaba localmente. Teníamos máquinas locales masivas en algunos casos (¡podías construir Maps en tu escritorio!), pero los tiempos de compilación se hicieron más y más largos a medida que crecía la base de código. Como era de esperar, incurrimos en gastos generales cada vez mayores en costos de personal debido al tiempo perdido, así como mayores costos de recursos para máquinas locales más grandes y potentes, y así sucesivamente. Estos costos de recursos fueron particularmente problemáticos: por supuesto, queremos que las personas tengan una compilación lo más rápida posible, pero la mayoría de las veces, una máquina de desarrollo de escritorio de alto rendimiento permanecerá inactiva. Esta no se siente como la forma correcta de invertir esos recursos.

Eventualmente, Google desarrolló su propio sistema de compilación distribuida. El desarrollo de este sistema tuvo un costo, por supuesto: a los ingenieros les tomó tiempo desarrollarlo, les tomó más tiempo a los ingenieros cambiar los hábitos y el flujo de trabajo de todos y aprender el nuevo sistema y, por supuesto, costó recursos computacionales adicionales. Pero los ahorros generales claramente valieron la pena: las compilaciones se hicieron más rápidas, se recuperó el tiempo de los ingenieros y la inversión en hardware se pudo enfocar en la infraestructura compartida administrada (en realidad, un subconjunto de nuestra flota de producción) en lugar de máquinas de escritorio cada vez más potentes. [capítulo 18](#) entra en más detalles sobre nuestro enfoque de compilaciones distribuidas y las compensaciones relevantes.

Entonces, construimos un nuevo sistema, lo implementamos en producción y aceleramos la construcción de todos. ¿Es ese el final feliz de la historia? No del todo: proporcionar un sistema de compilación distribuida mejoró enormemente la productividad de los ingenieros, pero con el paso del tiempo, las compilaciones distribuidas se inflaron. Lo que estaba restringido en el caso anterior por los ingenieros individuales (porque tenían un interés creado en mantener sus compilaciones locales lo más rápido posible) no estaba restringido dentro de un sistema de compilación distribuido. Las dependencias infladas o innecesarias en el gráfico de compilación se volvieron demasiado comunes. Cuando todos sintieron directamente el dolor de una construcción no óptima y fueron incentivados para estar atentos, los incentivos estuvieron mejor alineados. Al eliminar esos incentivos y ocultar dependencias infladas en una compilación distribuida en paralelo, creamos una situación en la que el consumo podía correr desenfrenado, y casi nadie tenía incentivos para vigilar el aumento de volumen. Esto es una reminiscencia de [Paradoja de Jevons](#): el consumo de un recurso puede *aumentar* como respuesta a una mayor eficiencia en su uso.

En general, los costos ahorrados asociados con la adición de un sistema de compilación distribuido superaron con creces los costos negativos asociados con su construcción y mantenimiento. Pero, como vimos con el aumento del consumo, no previmos todos estos costos. Habiendo avanzado, nos encontramos en una situación en la que necesitábamos reconceptualizar los objetivos y las limitaciones del sistema y nuestro uso, identificar las mejores prácticas (pequeñas dependencias, administración de dependencias de máquinas) y financiar las herramientas y el mantenimiento para el nuevo ecosistema. Incluso una compensación relativamente simple de la forma "Gastaremos \$\$\$ en recursos informáticos para recuperar el tiempo del ingeniero" tuvo efectos posteriores imprevistos.

Ejemplo: decidir entre tiempo y escala

Gran parte del tiempo, nuestros principales temas de tiempo y escala se superponen y trabajan en conjunto. Una política como la Regla Beyoncé escala bien y nos ayuda a mantener las cosas a lo largo del tiempo. Un cambio en la interfaz de un sistema operativo puede requerir muchas refactorizaciones pequeñas para adaptarse, pero la mayoría de esos cambios se escalarán bien porque tienen una forma similar: el cambio del sistema operativo no se manifiesta de manera diferente para cada persona que llama y cada proyecto.

Ocasionalmente, el tiempo y la escala entran en conflicto, y en ninguna parte tan claramente como en la pregunta básica: ¿deberíamos agregar una dependencia o bifurcarla/reimplementarla para que se adapte mejor a nuestras necesidades locales?

Esta pregunta puede surgir en muchos niveles de la pila de software porque, con frecuencia, una solución personalizada para su reducido espacio de problemas puede superar a la solución de utilidad general que necesita manejar todas las posibilidades. Al bifurcar o volver a implementar el código de la utilidad y personalizarlo para su dominio limitado, puede agregar nuevas funciones con mayor facilidad u optimizar con mayor certeza, independientemente de si estamos hablando de un microservicio, un caché en memoria, una rutina de compresión o cualquier otra cosa en nuestro ecosistema de software. Quizás lo más importante es que el control que obtiene de dicha bifurcación lo aísla de los cambios en sus dependencias subyacentes: esos cambios no están dictados por otro equipo o proveedor externo. Usted tiene el control de cómo y cuándo reaccionar ante el paso del tiempo y la necesidad de cambiar.

Por otro lado, si cada desarrollador bifurca todo lo que usa en su proyecto de software en lugar de reutilizar lo que existe, la escalabilidad sufre junto con la sostenibilidad. Reaccionar a un problema de seguridad en una biblioteca subyacente ya no se trata de actualizar una sola dependencia y sus usuarios: ahora se trata de identificar cada bifurcación vulnerable de esa dependencia y los usuarios de esas bifurcaciones.

Como ocurre con la mayoría de las decisiones de ingeniería de software, no existe una respuesta única para esta situación. Si la vida útil de su proyecto es corta, las bifurcaciones son menos riesgosas. Si la bifurcación en cuestión tiene un alcance limitado demostrable, eso también ayuda: evite bifurcaciones para interfaces que podrían operar a través del tiempo o los límites del tiempo del proyecto (estructuras de datos, formatos de serialización, protocolos de red). La consistencia tiene un gran valor, pero la generalidad tiene sus propios costos y, a menudo, puede ganar haciendo lo suyo, si lo hace con cuidado.

Revisar decisiones, cometer errores

Uno de los beneficios no reconocidos de comprometerse con una cultura basada en datos es la capacidad combinada y la necesidad de admitir errores. Se tomará una decisión en algún momento, con base en los datos disponibles, con suerte en base a buenos datos y solo unas pocas suposiciones, pero implícitamente en base a los datos actualmente disponibles. A medida que ingresan nuevos datos, cambian los contextos o se disipan las suposiciones, puede quedar claro que se tomó una decisión.

error o que tenía sentido en ese momento pero ya no lo tiene. Esto es particularmente crítico para una organización de larga duración: el tiempo no solo provoca cambios en las dependencias técnicas y los sistemas de software, sino también en los datos utilizados para impulsar las decisiones.

Creemos firmemente en los datos que informan las decisiones, pero reconocemos que los datos cambiarán con el tiempo y es posible que se presenten nuevos datos. Esto significa, inherentemente, que las decisiones deberán revisarse de vez en cuando durante la vida útil del sistema en cuestión. Para proyectos de larga duración, a menudo es fundamental tener la capacidad de cambiar de dirección después de tomar una decisión inicial. Y, lo que es más importante, significa que quienes toman las decisiones deben tener derecho a admitir errores. Contrariamente a los instintos de algunas personas, los líderes que admiten errores son más respetados, no menos.

Sea impulsado por la evidencia, pero también tenga en cuenta que las cosas que no se pueden medir aún pueden tener valor. Si eres un líder, eso es lo que se te ha pedido que hagas: ejercitarse el juicio, afirmar que las cosas son importantes. Hablaremos más sobre liderazgo en los capítulos [5](#) y [6](#).

Ingeniería de software versus programación

Cuando se le presente nuestra distinción entre ingeniería de software y programación, podría preguntarse si hay un juicio de valor inherente en juego. ¿Es la programación algo peor que la ingeniería de software? ¿Es un proyecto que se espera que dure una década con un equipo de cientos de personas inherentemente más valioso que uno que es útil solo por un mes y construido por dos personas?

Por supuesto no. Nuestro punto no es que la ingeniería de software sea superior, simplemente que estos representan dos dominios de problemas diferentes con restricciones, valores y mejores prácticas distintas. Más bien, el valor de señalar esta diferencia proviene de reconocer que algunas herramientas son excelentes en un dominio pero no en el otro. Probablemente no necesite confiar en las pruebas de integración (ver [capítulo 14](#)) y prácticas de implementación continua (CD) (ver [capítulo 24](#)) para un proyecto que durará solo unos días. Del mismo modo, todas nuestras preocupaciones a largo plazo sobre el control de versiones semánticas (SemVer) y la gestión de dependencias en proyectos de ingeniería de software (ver [capítulo 21](#)) realmente no se aplican a proyectos de programación a corto plazo: use lo que esté disponible para resolver la tarea en cuestión.

Creemos que es importante diferenciar entre los términos relacionados pero distintos "programación" e "ingeniería de software". Gran parte de esa diferencia se deriva de la gestión del código a lo largo del tiempo, el impacto del tiempo en la escala y la toma de decisiones frente a esas ideas. La programación es el acto inmediato de producir código. La ingeniería de software es el conjunto de políticas, prácticas y herramientas que son necesarias para hacer que ese código sea útil durante el tiempo que sea necesario y que permita la colaboración en un equipo.

Conclusión

Este libro analiza todos estos temas: políticas para una organización y para un solo programador, cómo evaluar y refinar sus mejores prácticas y las herramientas y tecnologías que intervienen en el software mantenable. Google ha trabajado duro para tener una base de código y una cultura sostenibles. No creemos necesariamente que nuestro enfoque sea la única forma verdadera de hacer las cosas, pero proporciona pruebas con el ejemplo de que se puede hacer. Esperamos que proporcione un marco útil para pensar en el problema general: ¿cómo mantiene su código durante el tiempo que necesita para seguir funcionando?

TL; DR

- La “ingeniería de software” se diferencia de la “programación” en la dimensionalidad: la programación se trata de producir código. La ingeniería de software amplía eso para incluir el mantenimiento de ese código durante su vida útil.
- Hay un factor de al menos 100.000 veces entre la vida útil del código de corta duración y el código de larga duración. Es una tontería suponer que las mismas mejores prácticas se aplican universalmente en ambos extremos de ese espectro.
- El software es sustentable cuando, durante la vida útil esperada del código, somos capaces de responder a los cambios en las dependencias, la tecnología o los requisitos del producto. Podemos optar por no cambiar las cosas, pero tenemos que ser capaces.
- Ley de Hyrum: con un número suficiente de usuarios de una API, no importa lo que prometas en el contrato: alguien dependerá de todos los comportamientos observables de tu sistema.
- Cada tarea que su organización tiene que hacer repetidamente debe ser escalable (lineal o mejor) en términos de aporte humano. Las políticas son una herramienta maravillosa para hacer que el proceso sea escalable.
- Las ineficiencias de los procesos y otras tareas de desarrollo de software tienden a aumentar lentamente. Tenga cuidado con los problemas de la rana hervida.
- La experiencia es particularmente rentable cuando se combina con economías de escala.
- “Porque yo lo digo” es una razón terrible para hacer las cosas.
- Ser impulsado por datos es un buen comienzo, pero en realidad, la mayoría de las decisiones se basan en una combinación de datos, suposiciones, precedentes y argumentos. Es mejor cuando los datos objetivos constituyen la mayoría de esas entradas, pero rara vez puede ser *todas* de ellos.
- Ser impulsado por los datos a lo largo del tiempo implica la necesidad de cambiar de dirección cuando los datos cambian (o cuando se disipen las suposiciones). Los errores o los planes revisados son inevitables.

PARTE II

Cultura

Cómo trabajar bien en equipos

*Escrito por Brian Fitzpatrick
Editado por Riona MacNamara*

Debido a que este capítulo trata sobre los aspectos culturales y sociales de la ingeniería de software en Google, tiene sentido comenzar centrándose en la única variable sobre la que definitivamente tiene control: usted.

Las personas son inherentemente imperfectas; nos gusta decir que los humanos son en su mayoría una colección de errores intermitentes. Pero antes de que pueda comprender los errores en sus compañeros de trabajo, debe comprender los errores en usted mismo. Le pediremos que piense en sus propias reacciones, comportamientos y actitudes y, a cambio, esperamos que obtenga una idea real de cómo convertirse en un ingeniero de software más eficiente y exitoso que gaste menos energía tratando con personas, problemas y más tiempo escribiendo buen código.

La idea fundamental de este capítulo es que el desarrollo de software es un esfuerzo de equipo. Y para tener éxito en un equipo de ingeniería, o en cualquier otra colaboración creativa, debe reorganizar sus comportamientos en torno a los principios básicos de humildad, respeto y confianza.

Antes de adelantarnos, comencemos observando cómo tienden a comportarse los ingenieros de software en general.

Ayúdame a ocultar mi código

Durante los últimos 20 años, mi colega Ben¹ y he hablado en muchas conferencias de programación. En 2006, lanzamos el Proyecto de código abierto de Google (ahora en desuso)

¹ Ben Collins-Sussman, también autor de este libro.

Servicio de alojamiento, y al principio solíamos recibir muchas preguntas y solicitudes sobre el producto. Pero a mediados de 2008, comenzamos a notar una tendencia en el tipo de solicitudes que recibíamos:

"¿Puede dar a Subversion en Google Code la capacidad de ocultar ramas específicas?"

"¿Puedes hacer posible la creación de proyectos de código abierto que comiencen ocultos para el mundo y luego se revelen cuando estén listos?"

"Hola, quiero reescribir todo mi código desde cero, ¿puedes borrar todo el historial?"

¿Puedes identificar un tema común a estas solicitudes?

La respuesta es *inseguridad*. La gente tiene miedo de que otros vean y juzguen su trabajo en progreso. En cierto sentido, la inseguridad es solo una parte de la naturaleza humana: a nadie le gusta que lo critiquen, especialmente por cosas que no están terminadas. Reconocer este tema nos indicó una tendencia más general dentro del desarrollo de software: la inseguridad es en realidad un síntoma de un problema mayor.

El mito del genio

Muchos humanos tienen el instinto de encontrar y adorar ídolos. Para los ingenieros de software, esos podrían ser Linus Torvalds, Guido Van Rossum, Bill Gates, todos héroes que cambiaron el mundo con hazañas heroicas. Linus escribió Linux por sí mismo, ¿verdad?

En realidad, lo que hizo Linus fue escribir solo los comienzos de un núcleo similar a Unix de prueba de concepto y mostrárselo a una lista de correo electrónico. Ese no fue un logro pequeño, y definitivamente fue un logro impresionante, pero fue solo la punta del iceberg. Linux es cientos de veces más grande que el núcleo inicial y fue desarrollado por *miles* de gente inteligente. El verdadero logro de Linus fue liderar a estas personas y coordinar su trabajo; Linux es el brillante resultado no de su idea original, sino del trabajo colectivo de la comunidad. (Y Unix en sí mismo no fue escrito en su totalidad por Ken Thompson y Dennis Ritchie, sino por un grupo de personas inteligentes en Bell Labs).

En esa misma nota, ¿Guido Van Rossum escribió personalmente todo Python? Ciertamente, él escribió la primera versión. Pero cientos de otros fueron responsables de contribuir a las versiones posteriores, incluidas ideas, funciones y correcciones de errores. Steve Jobs dirigió todo un equipo que construyó Macintosh, y aunque Bill Gates es conocido por escribir un intérprete BASIC para las primeras computadoras domésticas, su mayor logro fue construir una empresa exitosa en torno a MS-DOS. Sin embargo, todos se convirtieron en líderes y símbolos de los logros colectivos de sus comunidades. El Mito del Genio es la tendencia que nosotros, como humanos, necesitamos atribuir el éxito de un equipo a una sola persona/líder.

¿Y Michael Jordan?

Es la misma historia. Lo idolatrábamos, pero el hecho es que no ganó todos los partidos de baloncesto él solo. Su verdadero genio estaba en la forma en que trabajaba con su equipo. El entrenador del equipo, Phil Jackson, fue extremadamente inteligente y sus técnicas de entrenamiento son legendarias.

Reconoció que un jugador solo nunca gana un campeonato, por lo que reunió un "equipo de ensueño" completo en torno a MJ. Este equipo era una máquina bien engrasada, al menos tan impresionante como el propio Michael.

Entonces, ¿por qué idolatramos repetidamente al individuo en estas historias? ¿Por qué la gente compra productos respaldados por celebridades? ¿Por qué queremos comprar el vestido de Michelle Obama o los zapatos de Michael Jordan?

La celebridad es una gran parte de esto. Los seres humanos tienen un instinto natural para encontrar líderes y modelos a seguir, idolatrarlos e intentar imitarlos. Todos necesitamos héroes para inspirarnos, y el mundo de la programación también tiene sus héroes. El fenómeno de la "celebridad tecnológica" casi se ha extendido a la mitología. Todos queremos escribir algo que cambie el mundo como Linux o diseñar el próximo lenguaje de programación brillante.

En el fondo, muchos ingenieros desean secretamente ser vistos como genios. Esta fantasía es algo así:

- Le llama la atención un nuevo concepto asombroso.
- Te desvaneces en tu cueva durante semanas o meses, esclavizándote en una implementación perfecta de tu idea.
- A continuación, "libera" su software en el mundo, sorprendiendo a todos con su genialidad.
- Tus compañeros están asombrados por tu astucia.
- La gente hace fila para usar su software.
- La fama y la fortuna siguen naturalmente.

Pero espera: es hora de un control de la realidad. Probablemente no seas un genio.

Sin ofender, por supuesto, estamos seguros de que es una persona muy inteligente. ¿Pero te das cuenta de lo raros que son realmente los genios reales? Claro, escribes código, y esa es una habilidad complicada. Pero incluso si eres un genio, resulta que eso no es suficiente. Los genios aún cometen errores, y tener ideas brillantes y habilidades de programación de élite no garantiza que su software sea un éxito. Peor aún, es posible que se encuentre resolviendo solo problemas analíticos y *no humano* problemas. Definitivamente, ser un genio no es una excusa para ser un imbécil: cualquier persona, genio o no, con pocas habilidades sociales tiende a ser un mal compañero de equipo. La gran mayoría del trabajo en Google (¡y en la mayoría de las empresas!) no requiere un intelecto a nivel de genio, pero el 100 % del trabajo requiere un nivel mínimo de habilidades sociales. Lo que hará o arruinará tu carrera, especialmente en una empresa como Google, es lo bien que colabores con los demás.

Resulta que este Mito del Genio es solo otra manifestación de nuestra inseguridad. Muchos programadores tienen miedo de compartir el trabajo que acaban de empezar porque significa que sus compañeros verán sus errores y sabrán que el autor del código no es un genio.

Para citar a un amigo:

Sé que me siento SERIAMENTE inseguro acerca de las personas que buscan antes de que se haga algo. Como si me juzgaran seriamente y pensaran que soy un idiota.

Este es un sentimiento extremadamente común entre los programadores, y la reacción natural es esconderse en una cueva, trabajar, trabajar, trabajar, y luego pulir, pulir, pulir, seguro de que nadie verá tus errores y que aún así tenga la oportunidad de revelar su obra maestra cuando haya terminado. Escóndete hasta que tu código sea perfecto.

Otra motivación común para ocultar su trabajo es el temor de que otro programador pueda tomar su idea y ejecutarla antes de que usted comience a trabajar en ella. Al mantenerlo en secreto, controlas la idea.

Sabemos lo que probablemente estés pensando ahora: ¿y qué? ¿No se debería permitir que la gente trabaje como quiera?

En realidad no. En este caso, afirmamos que lo estás haciendo mal y que es un gran problema. Este es el por qué.

Esconderse Considerado Dañino

Si pasa todo su tiempo trabajando solo, está aumentando el riesgo de fallas innecesarias y engañando a su potencial de crecimiento. Aunque el desarrollo de software es un trabajo profundamente intelectual que puede requerir una profunda concentración y tiempo a solas, debe contrastar eso con el valor (y la necesidad!) de colaboración y revisión.

En primer lugar, ¿cómo sabes siquiera si estás en el camino correcto?

Imagina que eres un entusiasta del diseño de bicicletas y un día se te ocurre una idea brillante para una forma completamente nueva de diseñar una palanca de cambios. Pide repuestos y pasa semanas encerrado en su garaje tratando de construir un prototipo. Cuando tu vecino, también defensor de las bicicletas, te pregunta qué pasa, decides no hablar del tema. No querrás que nadie sepa sobre tu proyecto hasta que sea absolutamente perfecto. Pasan otros meses y tiene problemas para que su prototipo funcione correctamente. Pero debido a que está trabajando en secreto, es imposible solicitar el consejo de sus amigos con inclinaciones mecánicas.

Entonces, un día, su vecino saca su bicicleta de su garaje con un mecanismo de cambio radicalmente nuevo. Resulta que ha estado construyendo algo muy similar a tu invento, pero con la ayuda de unos amigos en la tienda de bicicletas. En este punto, estás exasperado. Le muestras tu trabajo. Señala que su diseño tenía algunas fallas simples, que podrían haberse solucionado en la primera semana si se lo hubiera mostrado. Hay una serie de lecciones que aprender aquí.

Detección temprana

Si mantiene su gran idea oculta del mundo y se niega a mostrarle nada a nadie hasta que la implementación esté pulida, está tomando una gran apuesta. Es fácil cometer errores fundamentales de diseño desde el principio. Corre el riesgo de reinventar las ruedas.² Y también pierde los beneficios de la colaboración: ¿observa cuánto más rápido se movió su vecino al trabajar con otros? Esta es la razón por la cual las personas sumergen los dedos de los pies en el agua antes de saltar en la parte más profunda: debe asegurarse de que está trabajando en lo correcto, lo está haciendo correctamente y no se ha hecho antes. Las posibilidades de un paso en falso temprano son altas. Cuantos más comentarios solicite desde el principio, más reducirá este riesgo.³ Recuerde el mantra probado y verdadero de "Falla temprano, falla rápido, falla a menudo".

El intercambio temprano no se trata solo de prevenir errores personales y hacer que sus ideas sean examinadas. También es importante fortalecer lo que llamamos el factor bus de tu proyecto.

El factor autobús

Bus factor (sustantivo): la cantidad de personas que necesitan ser atropelladas por un autobús antes de que su proyecto esté completamente condenado.

¿Qué tan disperso está el conocimiento y el saber hacer en su proyecto? Si usted es la única persona que entiende cómo funciona el código prototípico, es posible que disfrute de una buena seguridad laboral, pero si lo atropella un autobús, el proyecto está arruinado. Sin embargo, si está trabajando con un colega, ha duplicado el factor bus. Y si tiene un equipo pequeño que diseña y crea prototipos juntos, las cosas son aún mejores: el proyecto no se verá interrumpido cuando un miembro del equipo desaparezca. Recuerde: es posible que los autobuses no atropellen literalmente a los miembros del equipo, pero aún suceden otros eventos impredecibles de la vida. Alguien puede casarse, mudarse, dejar la empresa o tomar una licencia para cuidar a un pariente enfermo. Asegurarse de que haya *a/* *menos* una buena documentación, además de un propietario principal y secundario para cada área de responsabilidad, ayuda a garantizar el éxito de su proyecto en el futuro y aumenta el factor bus de su proyecto. Es de esperar que la mayoría de los ingenieros reconozcan que es mejor ser parte de un proyecto exitoso que la parte crítica de un proyecto fallido.

Más allá del factor autobús, está la cuestión del ritmo general de progreso. Es fácil olvidar que trabajar solo suele ser una tarea difícil, mucho más lenta de lo que la gente quiere admitir. ¿Cuánto aprendes cuando trabajas solo? ¿Qué tan rápido te mueves? Google y Stack Overflow son excelentes fuentes de opiniones e información, pero no reemplazan la experiencia humana real. Trabajar con otras personas aumenta directamente la sabiduría colectiva detrás del esfuerzo. Cuando te quedas atascado en algo absurdo, ¿cuánto tiempo pierdes en salir del agujero? Piensa en cómo

² Literalmente, si eres, de hecho, un diseñador de bicicletas.

³ Debo señalar que a veces es peligroso recibir demasiados comentarios demasiado pronto en el proceso si todavía estás inseguro de su dirección u objetivo general.

Diferente sería la experiencia si tuviera un par de compañeros que miraran por encima del hombro y le dijeran, al instante, cómo cometió el error y cómo superar el problema. Esta es exactamente la razón por la que los equipos se sientan juntos (o hacen programación en pareja) en las empresas de ingeniería de software. La programación es difícil. La ingeniería de software es aún más difícil. Necesitas ese segundo par de ojos.

Ritmo de progreso

Aquí hay otra analogía. Piense en cómo trabaja con su compilador. Cuando te sientes a escribir una gran pieza de software, ¿pasas días escribiendo 10 000 líneas de código y luego, después de escribir esa última línea perfecta, presionas el botón "compilar" por primera vez? Por supuesto que no. ¿Te imaginas qué tipo de desastre resultaría? Los programadores trabajan mejor en circuitos cerrados de retroalimentación: escribir una nueva función, compilar. Agregar una prueba, compilar. Refactorizar algún código, compilar. De esta forma, descubrimos y solucionamos errores tipográficos y errores lo antes posible después de generar el código. Queremos que el compilador esté a nuestro lado en cada pequeño paso; algunos entornos pueden incluso compilar nuestro código a medida que escribimos. Así es como mantenemos alta la calidad del código y nos aseguramos de que nuestro software evolucione correctamente, poco a poco. La filosofía actual de DevOps hacia la productividad tecnológica es explícita sobre este tipo de objetivos: obtener comentarios lo antes posible, realizar pruebas lo antes posible y pensar en entornos de seguridad y producción lo antes posible. Todo esto está incluido en la idea de "desplazarse a la izquierda" en el flujo de trabajo del desarrollador; cuanto antes encontramos un problema, más barato será solucionarlo.

Se necesita el mismo tipo de ciclo de retroalimentación rápida no solo a nivel de código, sino también a nivel de proyecto completo. Los proyectos ambiciosos evolucionan rápidamente y deben adaptarse a entornos cambiantes a medida que avanzan. Los proyectos se topan con obstáculos de diseño impredecibles o peligros políticos, o simplemente descubrimos que las cosas no funcionan según lo planeado. Los requisitos se transforman inesperadamente. ¿Cómo obtienes ese circuito de retroalimentación para que sepas el instante en que tus planes o diseños deben cambiar? Respuesta: trabajando en equipo. La mayoría de los ingenieros conocen la cita: "Muchos ojos hacen que todos los errores sean superficiales", pero una mejor versión podría ser: "Muchos ojos se aseguran de que su proyecto se mantenga relevante y encaminado". Las personas que trabajan en las cuevas se despiertan para descubrir que, si bien su visión original puede estar completa, el mundo ha cambiado y su proyecto se ha vuelto irrelevante.

Estudio de caso: ingenieros y oficinas

Hace veinticinco años, la sabiduría convencional decía que para que un ingeniero fuera productivo, necesitaba tener su propia oficina con una puerta que se cerrara. Supuestamente, esta era la única forma en que podían tener grandes bloques de tiempo ininterrumpidos para concentrarse profundamente en escribir montones de código.

Creo que no solo es innecesario para la mayoría de los ingenieros⁴ estar en una oficina privada, es francamente peligroso. Hoy en día, el software está escrito por equipos, no por individuos, y una conexión de gran ancho de banda y fácilmente disponible para el resto de su equipo es incluso más valiosa que su conexión a Internet. Puedes tener todo el tiempo ininterrumpido del mundo, pero si lo estás usando para trabajar en algo incorrecto, estás perdiendo el tiempo.

Desafortunadamente, parece que las empresas de tecnología de hoy en día (incluido Google, en algunos casos) han movido el péndulo exactamente hacia el extremo opuesto. Entre en sus oficinas y, a menudo, encontrará ingenieros agrupados en enormes salas (cien o más personas juntas) sin paredes. Este "plano de planta abierto" es ahora un tema de gran debate y, como resultado, la hostilidad hacia las oficinas abiertas va en aumento. La más mínima conversación se hace pública y la gente acaba sin hablar por riesgo de molestar a decenas de vecinos. ¡Esto es tan malo como las oficinas privadas!

Creemos que el término medio es realmente la mejor solución. Agrupe equipos de cuatro a ocho personas en salas pequeñas (u oficinas grandes) para que sea fácil (y no vergonzoso) que se produzca una conversación espontánea.

Por supuesto, en cualquier situación, los ingenieros individuales todavía necesitan una forma de filtrar el ruido y las interrupciones, razón por la cual la mayoría de los equipos que he visto han desarrollado una forma de comunicar que están ocupados y que deben limitar las interrupciones. Algunos de nosotros solíamos trabajar en un equipo con un protocolo de interrupción vocal: si querías hablar, decías "Breakpoint Mary", donde Mary era el nombre de la persona con la que querías hablar. Si Mary estuviera en un punto en el que pudiera detenerse, haría girar su silla y escucharía. Si Mary estaba demasiado ocupada, solo decía "ack" y tú continuabas con otras cosas hasta que terminara con su estado mental actual.

Otros equipos tienen fichas o animales de peluche que los miembros del equipo colocan en su monitor para indicar que deben ser interrumpidos solo en caso de emergencia. Otros equipos entregan audífonos con cancelación de ruido a los ingenieros para que sea más fácil lidiar con el ruido de fondo; de hecho, en muchas empresas, el mero acto de usar audífonos es una señal común que significa "no me molesten a menos que sea un problema". realmente importante." Muchos ingenieros tienden a usar el modo de solo auriculares cuando codifican, lo que puede ser útil para períodos breves pero, si se usa todo el tiempo, puede ser tan malo para la colaboración como encerrarse en una oficina.

No nos malinterprete: seguimos pensando que los ingenieros necesitan tiempo ininterrumpido para concentrarse en escribir código, pero creemos que también necesitan una conexión de gran ancho de banda y baja fricción con su equipo. Si las personas con menos conocimientos de su equipo sienten que hay una barrera para hacerle una pregunta, es un problema: encontrar el equilibrio adecuado es un arte.

⁴ Sin embargo, reconozco que los introvertidos serios probablemente necesiten más paz, tranquilidad y tiempo a solas que la mayoría de las personas y podrían beneficiarse de un entorno más tranquilo, si no de su propia oficina.

En resumen, no te escondas

Entonces, a lo que "esconderse" se reduce a esto: trabajar solo es inherentemente más riesgoso que trabajar con otros. Aunque tengas miedo de que alguien robe tu idea o piense que no eres inteligente, deberías estar mucho más preocupado por perder una gran parte de tu tiempo esforzándote en lo incorrecto.

No te conviertas en una estadística más.

Se trata del equipo

Entonces, retrocedamos ahora y pongamos todas estas ideas juntas.

El punto en el que hemos estado insistiendo es que, en el ámbito de la programación, los artesanos solitarios son extremadamente raros, e incluso cuando existen, no realizan logros sobrehumanos en el vacío; su logro que cambia el mundo es casi siempre el resultado de una chispa de inspiración seguida de un heroico esfuerzo de equipo.

Un gran equipo hace un uso brillante de sus superestrellas, pero el todo siempre es mayor que la suma de sus partes. Pero crear un equipo de superestrellas es diabólicamente difícil.

Pongamos esta idea en palabras más simples:*la ingeniería de software es un esfuerzo de equipo*.

Este concepto contradice directamente la fantasía interna del Programador Genio que muchos de nosotros tenemos, pero no es suficiente para ser brillante cuando estás solo en la guarida de tu hacker. No cambiarás el mundo ni harás las delicias de millones de usuarios de computadoras escondiendo y preparando tu invento secreto. Necesitas trabajar con otras personas. Comparte tu visión. Divide el trabajo. Aprende de los demás. Crea un equipo brillante.

Considere esto: ¿cuántas piezas de software exitosas y ampliamente utilizadas puede nombrar que realmente fueron escritas por una sola persona? (Algunas personas pueden decir "LaTeX", pero difícilmente es "ampliamente utilizado", ja menos que considere que la cantidad de personas que escriben artículos científicos es una porción estadísticamente significativa de todos los usuarios de computadoras!)

Los equipos de alto funcionamiento son oro y la verdadera clave del éxito. Deberías apuntar a esta experiencia como puedas.

Los tres pilares de la interacción social

Entonces, si el trabajo en equipo es la mejor ruta para producir un gran software, ¿cómo se crea (o encuentra) un gran equipo?

Para alcanzar el nirvana colaborativo, primero debe aprender y adoptar lo que yo llamo los "tres pilares" de las habilidades sociales. Estos tres principios no se tratan solo de engrasar las ruedas de las relaciones; son la base sobre la que se basan todas las interacciones y colaboraciones saludables:

Pilar 1: Humildad

No eres el centro del universo (¡tampoco lo es tu código!). No eres ni omnisciente ni infalible. Estás abierto a la superación personal.

Pilar 2: Respeto

Te preocupas genuinamente por las personas con las que trabajas. Los trata con amabilidad y aprecia sus habilidades y logros.

Pilar 3: Confianza

Usted cree que los demás son competentes y harán lo correcto, y está de acuerdo con dejarlos conducir cuando sea apropiado.⁵

Si realiza un análisis de causa raíz en casi cualquier conflicto social, en última instancia puede rastrearlo hasta la falta de humildad, respeto y/o confianza. Eso puede parecer inverosímil al principio, pero pruébalo. Piense en alguna situación social desagradable o incómoda que se encuentre actualmente en su vida. En el nivel más bajo, ¿todos están siendo apropiadamente humildes? ¿Las personas realmente se respetan unas a otras? ¿Hay confianza mutua?

¿Por qué importan estos pilares?

Cuando comenzó este capítulo, probablemente no planeaba inscribirse en algún tipo de grupo de apoyo semanal. Empatizamos. Lidiar con problemas sociales puede ser difícil: las personas son desordenadas, impredecibles y, a menudo, es molesto interactuar con ellas. En lugar de poner energía en analizar situaciones sociales y hacer movimientos estratégicos, es tentador descartar todo el esfuerzo. Es mucho más fácil pasar el rato con un compilador predecible, ¿no? ¿Por qué molestarse con las cosas sociales en absoluto?

Aquí hay una cita de [una famosa conferencia de Richard Hamming](#):

Al tomarme la molestia de contar chistes a las secretarias y ser un poco amigable, obtuve una ayuda secretarial excelente. Por ejemplo, una vez, por alguna estúpida razón, todos los servicios de reproducción de Murray Hill quedaron paralizados. No me pregunté cómo, pero lo fueron. Quería que se hiciera algo. Mi secretaria llamó a alguien en Holmdel, se montó [en] el automóvil de la empresa, hizo el viaje de una hora y lo reprodujo, y luego regresó. Fue una recompensa por las veces que me había esforzado por animarla, contarle chistes y ser amable; fue ese pequeño trabajo extra lo que luego valió la pena para mí. Al darse cuenta de que tiene que usar el sistema y estudiar cómo hacer que el sistema haga su trabajo, aprende a adaptar el sistema a sus deseos.

La moraleja es esta: no subestimes el poder de jugar el juego social. No se trata de engañar o manipular a las personas; se trata de crear relaciones para hacer las cosas. Las relaciones siempre duran más que los proyectos. Cuando tenga relaciones más ricas con sus compañeros de trabajo, estarán más dispuestos a hacer un esfuerzo adicional cuando los necesite.

⁵ Esto es increíblemente difícil si te has quemado en el pasado al delegar en personas incompetentes.

Humildad, respeto y confianza en la práctica

Toda esta predica sobre la humildad, el respeto y la confianza suena como un sermón. Salgamos de las nubes y pensemos cómo aplicar estas ideas en situaciones de la vida real. Vamos a examinar una lista de comportamientos específicos y ejemplos con los que puede comenzar. Muchos de ellos pueden sonar obvios al principio, pero después de que comience a pensar en ellos, notará con qué frecuencia usted (y sus compañeros) son culpables de no seguirlos. ¡Ciertamente nos hemos dado cuenta de esto!

perder el ego

OK, esta es una forma más simple de decirle a alguien sin la suficiente humildad que pierda su 'tudeza'. Nadie quiere trabajar con alguien que constantemente se comporta como si fuera la persona más importante de la sala. Incluso si sabes que eres la persona más sabia en la discusión, no lo agites en la cara de las personas. Por ejemplo, ¿siempre siente que necesita tener la primera o la última palabra en cada tema? ¿Sientes la necesidad de comentar cada detalle en una propuesta o discusión? ¿O conoces a alguien que haga estas cosas?

Aunque es importante ser humilde, eso no significa que tengas que ser un felpudo; no hay nada malo con la confianza en uno mismo. Simplemente no parezcas un sabelotodo. Aún mejor, piense en optar por un ego "colectivo", en su lugar; en lugar de preocuparse por si usted es personalmente increíble, trate de crear un sentido de logro en equipo y orgullo grupal. Por ejemplo, Apache Software Foundation tiene una larga historia de creación de comunidades en torno a proyectos de software. Estas comunidades tienen identidades increíblemente fuertes y rechazan a las personas que están más preocupadas por la autopromoción.

El ego se manifiesta de muchas maneras y, muchas veces, puede obstaculizar su productividad y ralentizarlo. Aquí hay otra gran historia de la conferencia de Hamming que ilustra este punto perfectamente (énfasis nuestro):

John Tukey casi siempre vestía de manera muy informal. Iría a una oficina importante y pasaría mucho tiempo antes de que el otro compañero se diera cuenta de que se trata de un hombre de primera clase y que sería mejor que escuchara. Durante mucho tiempo, John ha tenido que superar este tipo de hostilidad. ¡Es esfuerzo perdido! No dije que deberías conformarte; Le dije: "La apariencia de conformismo te lleva muy lejos". Si opta por hacer valer su ego de varias maneras, "lo haré a mi manera", paga un pequeño precio constante a lo largo de toda su carrera profesional. Y esto, durante toda una vida, se suma a una enorme cantidad de problemas innecesarios. [...] Al darte cuenta de que tienes que usar el sistema y estudiar cómo hacer que el sistema haga tu trabajo, aprendes a adaptar el sistema a tus deseos. *O puedes pelearla constantemente, como una pequeña guerra no declarada, durante toda tu vida.*

aprender a daryacepta críticas

Hace unos años, Joe comenzó un nuevo trabajo como programador. Después de su primera semana, realmente comenzó a profundizar en el código base. Debido a que le importaba lo que estaba pasando, comenzó a cuestionar amablemente a otros compañeros de equipo sobre sus contribuciones. Envío revisiones de código simples por correo electrónico, preguntando cortésmente sobre suposiciones de diseño o señalando lugares donde la lógica podría mejorarse. Después de un par de semanas, fue convocado a la oficina de su director. "¿Cuál es el problema?" preguntó Joe. "¿Hice algo mal?" El director parecía preocupado: "Hemos recibido muchas quejas sobre tu comportamiento, Joe. Aparentemente, has sido muy duro con tus compañeros de equipo, criticándolos a diestro y siniestro. Están molestos. Tienes que bajar el tono". Joe estaba completamente desconcertado. Seguramente, pensó, sus revisiones de código deberían haber sido bien recibidas y apreciadas por sus compañeros. En este caso, sin embargo,

En un entorno de ingeniería de software profesional, la crítica casi nunca es personal; por lo general, es solo parte del proceso de hacer un mejor proyecto. El truco consiste en asegurarse de que usted (y quienes lo rodean) entiendan la diferencia entre una crítica constructiva de la producción creativa de alguien y un ataque directo contra el carácter de alguien. Este último es inútil, es insignificante y casi imposible de actuar. El primero puede (¡y debería!) ser útil y brindar orientación sobre cómo mejorar. Y, lo que es más importante, está imbuido de respeto: la persona que hace la crítica constructiva se preocupa genuinamente por la otra persona y quiere que se mejore a sí misma o a su trabajo. Aprenda a respetar a sus compañeros y brinde críticas constructivas de manera cortés. Si realmente respetas a alguien, te sentirás motivado a elegir con tacto, fraseo útil—una habilidad que se adquiere con mucha práctica. Cubrimos esto mucho más en [Capítulo 9](#).

Del otro lado de la conversación, también debes aprender a aceptar las críticas. Esto significa no solo ser humilde acerca de tus habilidades, sino confiar en que la otra persona tiene tus mejores intereses (¡y los de tu proyecto!) en el fondo y no piensa que eres un idiota. La programación es una habilidad como cualquier otra cosa: mejora con la práctica. Si un compañero señalara formas en las que podrías mejorar tu malabarismo, ¿lo tomarías como un ataque a tu carácter y valor como ser humano? Esperamos que no. De la misma manera, su autoestima no debe estar conectada con el código que escribe, ni con ningún proyecto creativo que construya. Para repetirnos:*tu no eres tu código*. Di eso una y otra vez. No eres lo que haces. No solo debe creerlo usted mismo, sino también hacer que sus compañeros de trabajo lo crean.

Por ejemplo, si tiene un colaborador inseguro, esto es lo que no debe decir: "Hombre, te equivocaste totalmente en el flujo de control en ese método. Deberías usar el patrón de código xyzzy estándar como todos los demás". Esta retroalimentación está llena de antipatrones: le estás diciendo a alguien que está "equivocado" (como si el mundo fuera en blanco y negro), exigiéndole que cambie algo y acusándolo de crear algo que

va en contra de lo que hacen los demás (haciéndolos sentir estúpidos). Tu compañero de trabajo inmediatamente se sentirá ofendido y su respuesta seguramente será demasiado emocional.

Una mejor manera de decir lo mismo podría ser, "Oye, estoy confundido por el flujo de control en esta sección aquí. Me pregunto si el patrón de código xyzzy podría hacer esto más claro y más fácil de mantener". Observe cómo está usando la humildad para hacer la pregunta sobre usted, no sobre ellos. No están equivocados; solo tienes problemas para entender el código. La sugerencia se ofrece simplemente como una forma de aclarar las cosas para el pobrecito mientras posiblemente ayude a los objetivos de sostenibilidad a largo plazo del proyecto. Tampoco estás exigiendo nada, le estás dando a tu colaborador la capacidad de rechazar pacíficamente la sugerencia. La discusión se mantiene enfocada en el código en sí, no en el valor o las habilidades de codificación de nadie.

Fallar rápido e iterar

Hay una leyenda urbana bien conocida en el mundo de los negocios sobre un gerente que comete un error y pierde la impresionante cantidad de \$10 millones. Él va abatido a la oficina al día siguiente y comienza a empacar su escritorio, y cuando recibe la inevitable llamada de "el director ejecutivo quiere verte en su oficina", camina penosamente hasta la oficina del director ejecutivo y desliza silenciosamente una hoja de papel. al otro lado del escritorio.

"¿Qué es esto?" pregunta el director ejecutivo.

"Mi renuncia", dice el ejecutivo. "Supongo que me llamaste aquí para despedirme".

"¿Despedirte?" responde el CEO, incrédulo. "¿Por qué te despediría? ¡Acabo de gastar \$10 millones entrenándote!"⁶

Es una historia extrema, sin duda, pero el CEO en esta historia entiende que despedir al ejecutivo no desharía la pérdida de \$10 millones, y la agravaría al perder a un ejecutivo valioso que puede estar seguro de que no logrará eso. tipo de error de nuevo.

En Google, uno de nuestros lemas favoritos es que "El fracaso es una opción". Es ampliamente reconocido que si no está fallando de vez en cuando, no está siendo lo suficientemente innovador o tomando suficientes riesgos. El fracaso se ve como una oportunidad de oro para aprender y mejorar para el próximo intento.⁷ De hecho, a menudo se cita a Thomas Edison diciendo: "Si encuentro 10,000 formas en que algo no funcionará, no he fallado. No me desanimo, porque cada mal intento descartado es un paso más".

En Google X, la división que trabaja en "lanzamientos a la luna", como automóviles autónomos y acceso a Internet entregado por globos, el fracaso está integrado deliberadamente en su sistema de incentivos. A las personas se les ocurren ideas extravagantes y se alienta activamente a los compañeros de trabajo.

6 Puedes encontrar una docena de variantes de esta leyenda en la web, atribuidas a diferentes gerentes famosos.

7 De la misma manera, si haces lo mismo una y otra vez y sigues fallando, no es un fracaso, es incompetencia.

para derribarlos lo más rápido posible. Las personas son recompensadas (e incluso compiten) para ver cuántas ideas pueden refutar o invalidar en un período de tiempo determinado. Solo cuando un concepto realmente no puede ser desacreditado en una pizarra por todos los compañeros, se procede a un prototipo inicial.

Cultura post-mortem sin culpa

La clave para aprender de sus errores es documentar sus fracasos realizando un análisis de la causa raíz y redactando una "autopsia", como se le llama en Google (y en muchas otras empresas). Tenga mucho cuidado para asegurarse de que el documento post mórtem no sea solo una lista inútil de disculpas o excusas o acusaciones, ese no es su propósito. Una autopsia adecuada siempre debe contener una explicación de lo que se aprendió y lo que va a cambiar como resultado de la experiencia de aprendizaje. Luego, asegúrese de que la autopsia sea fácilmente accesible y que el equipo realmente cumpla con los cambios propuestos.

Documentar adecuadamente las fallas también facilita que otras personas (presentes y futuras) sepan qué sucedió y evitan que se repita la historia. No borre sus huellas, ¡ilumínelas como una pista para quienes lo siguen!

Una buena autopsia debe incluir lo siguiente:

- Un breve resumen del evento
- Una cronología del evento, desde el descubrimiento hasta la resolución, pasando por la investigación.
- La causa principal del evento
- Evaluación de impactos y daños
- Un conjunto de elementos de acción (con propietarios) para solucionar el problema de inmediato
- Un conjunto de elementos de acción para evitar que el evento vuelva a ocurrir
- Lecciones aprendidas

aprender paciencia

Hace años, estaba escribiendo una herramienta para convertir repositorios CVS a Subversion (y más tarde, a Git). Debido a los caprichos de CVS, seguí descubriendo errores extraños. Debido a que mi viejo amigo y compañero de trabajo, Karl, conocía muy bien CVS, decidimos que debíamos trabajar juntos para corregir estos errores.

Surgió un problema cuando empezamos a programar en pareja: soy un ingeniero de abajo hacia arriba que se contenta con zambullirse en el lodo y abrirse camino probando muchas cosas rápidamente y hojeando los detalles. Karl, sin embargo, es un ingeniero de arriba hacia abajo que quiere conocer el terreno completo y sumergirse en la implementación de casi todos los métodos en la pila de llamadas antes de proceder a abordar el error. Esto resultó en algunos conflictos interpersonales épicos, desacuerdos y discusiones acaloradas ocasionales. Llegó a la

punto en el que los dos simplemente no podíamos programar juntos: era demasiado frustrante para los dos.

Dicho esto, teníamos una larga historia de confianza y respeto mutuo. Combinado con la paciencia, esto nos ayudó a desarrollar un nuevo método de colaboración. Nos sentábamos juntos frente a la computadora, identificábamos el error y luego nos dividíamos y atacábamos el problema desde dos direcciones a la vez (de arriba hacia abajo y de abajo hacia arriba) antes de regresar con nuestros hallazgos. Nuestra paciencia y disposición para improvisar nuevos estilos de trabajo no solo salvaron el proyecto, sino también nuestra amistad.

Estar abierto a la influencia

Cuanto más abierto esté a la influencia, más podrá influir; cuanto más vulnerable eres, más fuerte pareces. Estas declaraciones suenan como extrañas contradicciones. Pero todos pueden pensar en alguien con quien han trabajado que es enloquecedoramente terco; no importa cuánto traten de persuadirlos las personas, se obstinan aún más. ¿Qué sucede eventualmente con esos miembros del equipo? En nuestra experiencia, las personas dejan de escuchar sus opiniones u objeciones; en cambio, terminan “enrutándolos” como un obstáculo que todos dan por sentado. Ciertamente no quieres ser esa persona, así que mantén esta idea en tu cabeza: está bien que otra persona cambie de opinión. En el capítulo inicial de este libro, dijimos que la ingeniería se trata inherentemente de compensaciones. Es imposible que tenga razón en todo todo el tiempo a menos que tenga un entorno inmutable y un conocimiento perfecto, por lo que, por supuesto, debe cambiar de opinión cuando se le presenten nuevas pruebas. Elige tus batallas con cuidado: para que te escuchen correctamente, primero debes escuchar a los demás. Es mejor hacer esto escuchando *antes* declarar una estaca en el suelo o anunciar con firmeza una decisión: si cambia constantemente de opinión, la gente pensará que es insípido.

La idea de vulnerabilidad también puede parecer extraña. Si alguien admite ignorar el tema en cuestión o la solución a un problema, ¿qué tipo de credibilidad tendrá en un grupo? La vulnerabilidad es una muestra de debilidad, y eso destruye la confianza, ¿no?

No es verdad. Admitir que cometiste un error o que simplemente estás fuera de tu liga puede mejorar tu estatus a largo plazo. De hecho, la voluntad de expresar vulnerabilidad es una muestra externa de humildad, demuestra responsabilidad y la voluntad de asumir la responsabilidad, y es una señal de que confías en las opiniones de los demás. A cambio, la gente termina respetando tu honestidad y fortaleza. A veces, lo mejor que puedes hacer es simplemente decir: "No lo sé".

Los políticos profesionales, por ejemplo, son notorios por no admitir nunca el error o la ignorancia, incluso cuando es evidente que están equivocados o desconocen un tema. Este comportamiento existe principalmente porque los políticos están constantemente bajo el ataque de sus oponentes, y es por eso que la mayoría de la gente no cree ni una palabra de lo que dicen los políticos. Sin embargo, cuando está escribiendo software, no necesita estar continuamente en el

a la defensiva: sus compañeros de equipo son colaboradores, no competidores. Todos ustedes tienen el mismo objetivo.

ser google

En Google, tenemos nuestra propia versión interna de los principios de "humildad, respeto y confianza" cuando se trata de comportamiento e interacciones humanas.

Desde los primeros días de nuestra cultura, a menudo nos referimos a las acciones como "Googley" o "no Googley". La palabra nunca se definió explícitamente; más bien, todos lo interpretaron como "no seas malo" o "haz lo correcto" o "sé bueno con los demás". Con el tiempo, las personas también comenzaron a usar el término "Googley" como una prueba informal de adecuación cultural cada vez que entrevistamos a un candidato para un trabajo de ingeniería, o cuando escribimos evaluaciones internas de desempeño de unos y otros. La gente a menudo expresaba opiniones sobre otros usando el término; por ejemplo, "la persona codificó bien, pero no parecía tener una actitud muy similar a la de Google".

Por supuesto, finalmente nos dimos cuenta de que el término "Googley" estaba sobrecargado de significado; peor aún, podría convertirse en una fuente de sesgo inconsciente en la contratación o las evaluaciones. Si "Googley" significa algo diferente para cada empleado, corremos el riesgo de que el término empiece a significar "*es como yo*". Obviamente, esa no es una buena prueba para contratar — no queremos contratar personas "*como yo*", sino personas de diversos orígenes y con diferentes opiniones y experiencias. El deseo personal de un entrevistador de tomar una cerveza con un candidato (o compañero de trabajo) debe *Nunca* considerarse una señal válida sobre el rendimiento o la capacidad de alguien más para prosperar en Google.

Google finalmente solucionó el problema definiendo explícitamente una rúbrica para lo que entendemos por "Googleyness", un conjunto de atributos y comportamientos que buscamos que representan un liderazgo sólido y ejemplifican "humildad, respeto y confianza":

Prospera en la ambigüedad

Puede lidiar con mensajes o instrucciones contradictorias, generar consenso y avanzar en la solución de un problema, incluso cuando el entorno cambia constantemente.

Retroalimentación de valores

Tiene humildad tanto para recibir como para dar retroalimentación con gracia y comprende cuán valiosa es la retroalimentación para el desarrollo personal (y del equipo).

Desafía el statu quo

Es capaz de establecer metas ambiciosas y perseguirlas incluso cuando puede haber resistencia o inercia por parte de los demás.

Pone al usuario primero

Tiene empatía y respeto por los usuarios de los productos de Google y lleva a cabo las acciones que más les convienen.

Se preocupa por el equipo

Tiene empatía y respeto por los compañeros de trabajo y trabaja activamente para ayudarlos sin que se lo pidan, mejorando la cohesión del equipo.

hace lo correcto

Tiene un fuerte sentido de la ética en todo lo que hace; dispuesto a tomar decisiones difíciles o inconvenientes para proteger la integridad del equipo y el producto.

Ahora que tenemos estos comportamientos de mejores prácticas mejor definidos, hemos comenzado a evitar usar el término "Googley". ¡Siempre es mejor ser específico sobre las expectativas!

Conclusión

La base de casi cualquier empresa de software, de casi cualquier tamaño, es un equipo que funcione bien. Aunque el mito del genio del desarrollador de software en solitario todavía persiste, la verdad es que nadie lo hace solo. Para que una organización de software supere la prueba del tiempo, debe tener una cultura saludable, arraigada en la humildad, la confianza y el respeto que gira en torno al equipo, en lugar del individuo. Además, la naturaleza creativa del desarrollo de software requiere que las personas toman riesgos y ocasionalmente fracasan; para que las personas acepten ese fracaso, debe existir un ambiente de equipo saludable.

TL; DR

- Sea consciente de las ventajas y desventajas de trabajar en forma aislada.
- Reconozca la cantidad de tiempo que usted y su equipo pasan comunicándose y en conflicto interpersonal. Una pequeña inversión en la comprensión de las personalidades y los estilos de trabajo de usted y los demás puede contribuir en gran medida a mejorar la productividad.
- Si desea trabajar eficazmente con un equipo o una organización grande, sea consciente de su estilo de trabajo preferido y el de los demás.

El intercambio de conocimientos

***Escrito por Nina Chen y Mark Barolak
Editado por Riona MacNamara***

Su organización entiende el dominio de su problema mejor que cualquier persona al azar en Internet; su organización debería ser capaz de responder a la mayoría de sus propias preguntas. Para lograrlo, necesita expertos que sepan las respuestas a esas preguntas y mecanismos para distribuir su conocimiento, que es lo que exploraremos en este capítulo. Estos mecanismos van desde los más simples (haga preguntas, escriba lo que sabe) hasta los mucho más estructurados, como tutoriales y clases. Sin embargo, lo más importante es que su organización necesita una cultura del aprendizaje, y eso requiere crear la seguridad psicológica que permita a las personas admitir su falta de conocimiento.

Desafíos para el aprendizaje

Compartir la experiencia en una organización no es una tarea fácil. Sin una fuerte cultura de aprendizaje, pueden surgir desafíos. Google ha experimentado varios de estos desafíos, especialmente a medida que la empresa ha escalado:

Falta de seguridad psicológica.

Un entorno en el que las personas tienen miedo de correr riesgos o cometer errores frente a los demás porque temen ser castigados por ello. Esto a menudo se manifiesta como una cultura del miedo o una tendencia a evitar la transparencia.

Islas de información

Fragmentación del conocimiento que ocurre en diferentes partes de una organización que no se comunican entre sí ni utilizan recursos compartidos. En tal

entorno, cada grupo desarrolla su propia manera de hacer las cosas.¹ Esto a menudo conduce a lo siguiente:

Fragmentación de la información

Cada isla tiene una imagen incompleta del todo más grande.

Duplicación de información

Cada isla ha reinventado su propia manera de hacer algo.

sesgo de información

Cada isla tiene sus propias formas de hacer lo mismo, y estas pueden o no entrar en conflicto.

Punto único de falla (SPOF)

Un cuello de botella que ocurre cuando la información crítica está disponible de una sola persona. Esto está relacionado con *factor de autobús*, que se analiza con más detalle en [Capítulo 2](#).

Los SPOF pueden surgir de buenas intenciones: puede ser fácil caer en el hábito de "Déjame encargarme de eso por ti". Pero este enfoque optimiza la eficiencia a corto plazo ("Es más rápido para mí hacerlo") a costa de una pobre escalabilidad a largo plazo (el equipo nunca aprende cómo hacer lo que sea necesario). Esta mentalidad también tiende a conducir *apericia de todo o nada*.

Experiencia en todo o nada

Un grupo de personas que se divide entre los que saben "todo" y los novatos, con poco término medio. Este problema a menudo se refuerza si los expertos siempre hacen todo por sí mismos y no se toman el tiempo para desarrollar nuevos expertos a través de tutorías o documentación. En este escenario, el conocimiento y las responsabilidades continúan acumulándose en aquellos que ya tienen experiencia, y los nuevos miembros del equipo o los novatos deben valerse por sí mismos y avanzar más lentamente.

loro

Mimetismo sin comprensión. Esto se caracteriza típicamente por copiar patrones o código sin pensar sin entender su propósito, a menudo bajo la suposición de que dicho código es necesario por razones desconocidas.

cementerios embrujados

Lugares, a menudo en código, que las personas evitan tocar o cambiar porque temen que algo salga mal. A diferencia de lo mencionado *loro*, los cementerios embrujados se caracterizan por personas que evitan la acción por miedo y superstición.

¹ En otras palabras, en lugar de desarrollar un único máximo global, tenemos un montón de máximos locales.

En el resto de este capítulo, profundizaremos en las estrategias que las organizaciones de ingeniería de Google han encontrado exitosas para abordar estos desafíos.

Filosofía

La ingeniería de software se puede definir como el desarrollo multipersona de programas multiversión.² Las personas son el núcleo de la ingeniería de software: el código es un resultado importante, pero solo una pequeña parte de la construcción de un producto. Fundamentalmente, el código no surge espontáneamente de la nada, y tampoco la experiencia. Todo experto alguna vez fue un novato: el éxito de una organización depende del crecimiento y la inversión en su gente.

El asesoramiento personalizado de un experto siempre es invaluable. Los diferentes miembros del equipo tienen diferentes áreas de especialización, por lo que variará el mejor compañero de equipo para hacer una pregunta determinada. Pero si el experto se va de vacaciones o cambia de equipo, el equipo puede quedarse en la estacada. Y aunque una persona podría brindar ayuda personalizada para uno a muchos, esto no se escala y se limita a un pequeño número de "muchos".

El conocimiento documentado, por otro lado, puede escalar mejor no solo al equipo sino a toda la organización. Mecanismos como un wiki de equipo permiten que muchos autores compartan su experiencia con un grupo más grande. Pero a pesar de que la documentación escrita es más escalable que las conversaciones uno a uno, esa escalabilidad viene con algunas ventajas y desventajas: puede ser más generalizada y menos aplicable a las situaciones de los estudiantes individuales, y viene con el costo adicional de mantenimiento requerido para mantener información relevante y actualizada en el tiempo.

El conocimiento tribal existe en la brecha entre lo que saben los miembros individuales del equipo y lo que está documentado. Los expertos humanos saben estas cosas que no están escritas. Si documentamos ese conocimiento y lo mantenemos, ahora está disponible no solo para alguien con acceso directo uno a uno al experto hoy, sino para cualquiera que pueda encontrar y ver la documentación.

Entonces, en un mundo mágico en el que todo está siempre perfectamente e inmediatamente documentado, ya no necesitaríamos consultar a una persona, ¿verdad? No exactamente. El conocimiento escrito tiene ventajas de escala, pero también lo hace la ayuda humana dirigida. Un experto humano puede sintetizar su extensión de conocimiento. Pueden evaluar qué información es aplicable al caso de uso del individuo, determinar si la documentación sigue siendo relevante y saber dónde encontrarla. O, si no saben dónde encontrar las respuestas, es posible que sepan quién las sabe.

2 David Lorge Parnas, *Ingeniería de Software: Desarrollo Multipersonal de Programas Multiversión* (Heidelberg: Springer-Verlag Berlín, 2011).

El conocimiento tribal y el escrito se complementan entre sí. Incluso un equipo perfectamente experto con una documentación perfecta necesita comunicarse entre sí, coordinarse con otros equipos y adaptar sus estrategias a lo largo del tiempo. Ningún enfoque único de intercambio de conocimientos es la solución correcta para todos los tipos de aprendizaje, y los detalles de una buena combinación probablemente variarán según su organización. El conocimiento institucional evoluciona con el tiempo, y los métodos de intercambio de conocimientos que funcionan mejor para su organización probablemente cambiarán a medida que crezca. Capacítense en el aprendizaje y el crecimiento, y construya su propio grupo de expertos: no existe demasiada experiencia en ingeniería.

Preparando el escenario: seguridad psicológica

La seguridad psicológica es fundamental para promover un entorno de aprendizaje.

Para aprender, primero debes reconocer que hay cosas que no entiendes. Deberíamos **bienvenida tanta honestidad** en lugar de castigarlo. (Google hace esto bastante bien, pero a veces los ingenieros son reacios a admitir que no entienden algo).

Una gran parte del aprendizaje es poder probar cosas y sentirse seguro de fallar. En un ambiente saludable, las personas se sienten cómodas haciendo preguntas, equivocándose y aprendiendo cosas nuevas. Esta es una expectativa básica para todos los equipos de Google; Por supuesto, **nuestra investigación** ha demostrado que la seguridad psicológica es la parte más importante de un equipo eficaz.

Tutoría

En Google, tratamos de establecer el tono tan pronto como un ingeniero "Noogler" (nuevo Googler) se une a la empresa. Una forma importante de desarrollar la seguridad psicológica es asignar a los Nooglers un mentor, alguien que sea **nos** u **su** miembro del equipo, gerente o líder técnico, cuyas responsabilidades incluyen explícitamente responder preguntas y ayudar a Noogler a aumentar. Tener un mentor asignado oficialmente para pedir ayuda hace que sea más fácil para el recién llegado y significa que no necesita preocuparse por tomar demasiado tiempo de sus compañeros de trabajo.

Un mentor es un voluntario que ha estado en Google durante más de un año y que está disponible para asesorar sobre cualquier cosa, desde el uso de la infraestructura de Google hasta la navegación por la cultura de Google. Fundamentalmente, el mentor está allí para ser una red de seguridad con la que hablar si el aprendiz no sabe a quién más pedir consejo. Este mentor no está en el mismo equipo que el aprendiz, lo que puede hacer que el aprendiz se sienta más cómodo para pedir ayuda en situaciones difíciles.

La tutoría formaliza y facilita el aprendizaje, pero el aprendizaje en sí mismo es un proceso continuo. Siempre habrá oportunidades para que los compañeros de trabajo aprendan unos de otros, ya sea un nuevo empleado que se une a la organización o un ingeniero experimentado que aprende una nueva tecnología. Con un equipo saludable, los compañeros de equipo estarán abiertos no solo a

responder sino también *apidiendo* preguntas: mostrando que no saben algo y aprendiendo unos de otros.

Seguridad Psicológica en Grupos Grandes

Pedir ayuda a un compañero de equipo cercano es más fácil que acercarse a un gran grupo de extraños en su mayoría. Pero como hemos visto, las soluciones uno a uno no escalan bien. Las soluciones grupales son más escalables, pero también dan más miedo. Puede ser intimidante para los novatos formular una pregunta y hacérsela a un grupo grande, sabiendo que su pregunta podría quedar archivada durante muchos años. La necesidad de seguridad psicológica se amplifica en grupos grandes. Cada miembro del grupo tiene un papel que desempeñar en la creación y el mantenimiento de un entorno seguro que garantice que los recién llegados se sientan seguros al hacer preguntas y que los nuevos expertos se sientan capacitados para ayudar a esos recién llegados sin temor a que sus respuestas sean atacadas por expertos establecidos.

La forma más importante de lograr este entorno seguro y acogedor es que las interacciones grupales sean cooperativas, no antagónicas. **Tabla 3-1** enumera algunos ejemplos de patrones recomendados (y sus antipatrones correspondientes) de interacciones grupales.

Tabla 3-1. Patrones de interacción grupal

Patrones recomendados (cooperativo)	Antipatrones (adversario)
Las preguntas básicas o los errores se guían en la dirección correcta.	Se critican las preguntas básicas o los errores y se regaña a la persona que hace la pregunta.
Las explicaciones se dan con la intención de ayudar a la persona que hace la pregunta a aprender.	Las explicaciones se dan con la intención de mostrar el propio conocimiento.
Las respuestas son amables, pacientes y útiles.	Las respuestas son condescendientes, sarcásticas y poco constructivas.
Las interacciones son discusiones compartidas para encontrar soluciones.	Las interacciones son discusiones con "ganadores" y "perdedores"

Estos antipatrones pueden surgir de manera no intencional: alguien podría estar tratando de ayudar, pero accidentalmente es condescendiente y poco acogedor. Encontramos el [Reglas sociales del Recurso Center](#) para ser útil aquí:

Sin sorpresa fingida ("¡Qué?! ¡No puedo creer que no sepas cuál es la pila!")

La sorpresa fingida es una barrera para la seguridad psicológica y hace que los miembros del grupo tengan miedo de admitir su falta de conocimiento.

Sin "bueno, en realidad"

Correcciones pedantes que tienden a ser más grandiosas que precisas.

No conducir en el asiento trasero

Interrumpir una discusión existente para ofrecer opiniones sin comprometerse con la conversación.

Sin "-ismos" sutiles ("¡Es tan fácil que mi abuela podría hacerlo!")

Pequeñas expresiones de prejuicio (racismo, discriminación por edad, homofobia) que pueden hacer que las personas se sientan indeseables, irrespetadas o inseguras.

Creciendo tu conocimiento

El intercambio de conocimientos comienza contigo mismo. Es importante reconocer que siempre tienes algo que aprender. Las siguientes pautas le permiten aumentar su propio conocimiento personal.

Hacer preguntas

Si le quitas una sola cosa a este capítulo, es esta: estar siempre aprendiendo; estar siempre haciendo preguntas.

Le decimos a Nooglers que la puesta en marcha puede llevar alrededor de seis meses. Este período prolongado es necesario para aumentar la infraestructura grande y compleja de Google, pero también refuerza la idea de que el aprendizaje es un proceso iterativo continuo. Uno de los mayores errores que cometan los principiantes es no pedir ayuda cuando están atascados. Es posible que sienta la tentación de luchar solo o tenga miedo de que sus preguntas sean "demasiado simples". "Solo necesito esforzarme más antes de pedirle ayuda a alguien", piensas. ¡No caigas en esta trampa! Sus compañeros de trabajo suelen ser la mejor fuente de información: aproveche este valioso recurso.

No existe un día mágico en el que de repente sepas exactamente qué hacer en cada situación; siempre hay más que aprender. Los ingenieros que han estado en Google durante años todavía tienen áreas en las que sienten que no saben lo que están haciendo, ¡y eso está bien! No tenga miedo de decir "No sé qué es eso; ¿Podrías explicarlo? Acepte el no saber las cosas como un área de oportunidad en lugar de uno a temer.³

No importa si es nuevo en un equipo o un líder senior: siempre debe estar en un entorno en el que haya algo que aprender. Si no, te estancas (y deberías encontrar un nuevo entorno).

Es especialmente crítico que aquellos en roles de liderazgo modelen este comportamiento: es importante no equiparar erróneamente "antigüedad" con "saberlo todo". De hecho, cuanto más sabes,**cuanto más sabes que no sabes**. Hacer preguntas abiertamente⁴ o expresar lagunas en el conocimiento refuerza que está bien que otros hagan lo mismo.

³Síndrome impostor es poco común entre los grandes triunfadores, y los Googlers no son una excepción; de hecho, un La mayoría de los autores de este libro tienen el síndrome del impostor. Reconocemos que el miedo al fracaso puede ser difícil para las personas con síndrome del impostor y puede reforzar la inclinación a evitar la ramificación.

⁴ Véase "Cómo hacer buenas preguntas."

En el extremo receptor, la paciencia y la amabilidad al responder preguntas fomentan un entorno en el que las personas se sienten seguras al buscar ayuda. Hacer que sea más fácil superar la vacilación inicial para hacer una pregunta establece el tono desde el principio: comuníquese para solicitar preguntas y facilite que incluso las preguntas "triviales" obtengan una respuesta. Aunque los ingenieros *podrían* probablemente descubran el conocimiento tribal por su cuenta, no están aquí para trabajar en el vacío. La ayuda dirigida permite a los ingenieros ser productivos más rápido, lo que a su vez hace que todo su equipo sea más productivo.

Comprender el contexto

Aprender no se trata solo de comprender cosas nuevas; también incluye desarrollar una comprensión de las decisiones detrás del diseño y la implementación de las cosas existentes. Suponga que su equipo hereda un código base heredado para una pieza crítica de infraestructura que ha existido durante muchos años. Los autores originales ya no están y el código es difícil de entender. Puede ser tentador volver a escribir desde cero en lugar de dedicar tiempo a aprender el código existente. Pero en lugar de pensar "No lo entiendo" y terminar tus pensamientos ahí, profundiza más: ¿qué preguntas deberías estar haciendo?

Considere el principio de "**valla de chesterson**": antes de quitar o cambiar algo, primero entienda por qué está ahí.

En el asunto de reformar las cosas, a diferencia de deformarlas, hay un principio claro y simple; un principio que probablemente se llamará una paradoja. Existe en tal caso cierta institución o ley; digamos, en aras de la sencillez, una valla o puerta erigida a través de un camino. El tipo más moderno de reformador se acerca alegremente y dice: "No veo el uso de esto; aclarémoslo. A lo que el tipo de reformador más inteligente hará bien en responder: "Si no ves el uso de esto, ciertamente no dejaré que lo elimines. Vete y piensa. Entonces, cuando puedas volver y decirme que ves el uso de eso, puedo permitirte destruirlo".

Esto no significa que el código no pueda carecer de claridad o que los patrones de diseño existentes no puedan estar equivocados, pero los ingenieros tienden a buscar "¡esto es malo!" mucho más rápido de lo que a menudo se justifica, especialmente para códigos, lenguajes o paradigmas desconocidos. Google no es inmune a esto. Busque y comprenda el contexto, especialmente para las decisiones que parecen inusuales. Una vez que haya entendido el contexto y el propósito del código, considere si su cambio aún tiene sentido. Si lo hace, adelante y hazlo; si no es así, documente su razonamiento para futuros lectores.

Muchas guías de estilo de Google incluyen explícitamente el contexto para ayudar a los lectores a comprender la lógica detrás de las pautas de estilo en lugar de simplemente memorizar una lista de reglas arbitrarias. Más sutilmente, la comprensión de la lógica detrás de una directriz determinada permite a los autores tomar decisiones informadas sobre cuándo no se debe aplicar la directriz o si es necesario actualizarla. Ver [Capítulo 8](#).

Escalando sus preguntas: pregunte a la comunidad

Obtener ayuda personalizada requiere un gran ancho de banda, pero necesariamente tiene una escala limitada. Y como estudiante, puede ser difícil recordar cada detalle. Hágase un favor a su futuro yo: cuando aprenda algo de una conversación uno a uno,*escribelo*.

Lo más probable es que los futuros recién llegados tengan las mismas preguntas que usted tenía. Hágales un favor a ellos también, *ycomparte lo que escribes*.

Aunque compartir las respuestas que recibe puede ser útil, también es beneficioso buscar ayuda no de individuos sino de la comunidad en general. En esta sección, examinamos diferentes formas de aprendizaje basado en la comunidad. Cada uno de estos enfoques (chats grupales, listas de correo y sistemas de preguntas y respuestas) tienen diferentes ventajas y desventajas y se complementan entre sí. Pero cada uno de ellos permite que el buscador de conocimientos obtenga ayuda de una comunidad más amplia de pares y expertos y también garantiza que las respuestas estén ampliamente disponibles para los miembros actuales y futuros de esa comunidad.

Chats grupales

Cuando tiene una pregunta, a veces puede ser difícil obtener ayuda de la persona adecuada. Tal vez no estés seguro de quién sabe la respuesta, o la persona a la que quieras preguntar está ocupada. En estas situaciones, los chats grupales son excelentes, porque puede hacer su pregunta a muchas personas a la vez y tener una conversación rápida de ida y vuelta con quien esté disponible. Como beneficio adicional, otros miembros del chat grupal pueden aprender de la pregunta y la respuesta, y muchas formas de chat grupal se pueden archivar y buscar automáticamente más adelante.

Los chats grupales tienden a estar dedicados a temas o equipos. Los chats grupales basados en temas generalmente están abiertos para que cualquiera pueda ingresar y hacer una pregunta. Tienden a atraer a expertos y pueden crecer bastante, por lo que las preguntas generalmente se responden rápidamente. Los chats orientados a equipos, por otro lado, tienden a ser más pequeños y restringen la membresía. Como resultado, es posible que no tengan el mismo alcance que un chat basado en temas, pero su tamaño más pequeño puede resultar más seguro para un recién llegado.

Aunque los chats grupales son excelentes para preguntas rápidas, no brindan mucha estructura, lo que puede dificultar la extracción de información significativa de una conversación en la que no participa activamente. Tan pronto como necesite compartir información fuera del grupo, o ponerla a disposición para consultarla más tarde, debe escribir un documento o enviar un correo electrónico a una lista de correo.

Listas de correo

La mayoría de los temas en Google tienen una lista de correo de grupos de Google topic-users@ o topic-discuss@ a la que cualquier persona de la empresa puede unirse o enviar un correo electrónico. Hacer una pregunta en una lista de correo pública es muy parecido a hacer una conversación grupal: la pregunta llega a muchas personas que podrían

potencialmente respóndala y cualquier persona que siga la lista puede aprender de la respuesta. Sin embargo, a diferencia de los chats grupales, las listas de correo públicas son fáciles de compartir con una audiencia más amplia: están empaquetadas en archivos que permiten realizar búsquedas y los hilos de correo electrónico brindan más estructura que los chats grupales. En Google, las listas de correo también están indexadas y pueden ser descubiertas por Moma, el motor de búsqueda de la intranet de Google.

Cuando encuentra una respuesta a una pregunta que hizo en una lista de correo, puede ser tentador continuar con su trabajo. ¡No lo hagas! Nunca se sabe cuándo alguien necesitará la misma información en el futuro, por lo que se recomienda volver a publicar la respuesta en la lista.

Las listas de correo no están exentas de compensaciones. Son adecuados para preguntas complicadas que requieren mucho contexto, pero son torpes para los intercambios rápidos de ida y vuelta en los que sobresalen los chats grupales. Un hilo sobre un problema en particular es generalmente más útil mientras está activo. Los archivos de correo electrónico son inmutables y puede ser difícil determinar si una respuesta descubierta en un antiguo hilo de discusión sigue siendo relevante para una situación actual. Además, la relación señal-ruido puede ser menor que la de otros medios, como la documentación formal, porque el problema que alguien tiene con su flujo de trabajo específico podría no ser aplicable a usted.

Correo electrónico en Google

La cultura de Google es infamemente centrada en el correo electrónico y cargada de correo electrónico. Los ingenieros de Google reciben cientos de correos electrónicos (si no más) cada día, con diversos grados de capacidad de acción. Los nooglers pueden pasar días simplemente configurando filtros de correo electrónico para manejar el volumen de notificaciones provenientes de grupos a los que se han suscrito automáticamente; algunas personas simplemente se dan por vencidas y no intentan seguir el ritmo. Algunos grupos copian grandes listas de correo en cada discusión de forma predeterminada, sin tratar de dirigir la información a aquellos que probablemente estén específicamente interesados en ella; como resultado, la relación señal-ruido puede ser un verdadero problema.

Google tiende hacia los flujos de trabajo basados en correo electrónico de forma predeterminada. Esto no se debe necesariamente a que el correo electrónico sea un medio mejor que otras opciones de comunicación (a menudo no lo es), sino a que es a lo que está acostumbrada nuestra cultura. Tenga esto en cuenta cuando su organización considere qué formas de comunicación fomentar o invertir.

YAQS: Plataforma de Preguntas y Respuestas

YAQS ("Yet Another Question System") es una versión interna de Google de un Desbordamiento de pila—como un sitio web, lo que facilita a los Googlers vincular a un código existente o en proceso, así como discutir información confidencial.

Al igual que Stack Overflow, YAQS comparte muchas de las mismas ventajas de las listas de correo y agrega mejoras: las respuestas marcadas como útiles se promocionan en la interfaz de usuario y

los usuarios pueden editar preguntas y respuestas para que sigan siendo precisas y útiles a medida que cambian el código y los hechos. Como resultado, algunas listas de correo han sido reemplazadas por YAQS, mientras que otras se han convertido en listas de discusión más generales que están menos enfocadas en la resolución de problemas.

Escalando su conocimiento: Siempre tienes algo que enseñar

La enseñanza no se limita a los expertos, ni la experiencia es un estado binario en el que eres un novato o un experto. La experiencia es un vector multidimensional de lo que sabes: todos tienen diferentes niveles de experiencia en diferentes áreas. Esta es una de las razones por las que la diversidad es fundamental para el éxito organizacional: diferentes personas aportan diferentes perspectivas y conocimientos a la mesa (ver Capítulo 4). Los ingenieros de Google enseñan a otros de diversas maneras, como durante el horario de oficina, dando charlas técnicas, dando clases, escribiendo documentación y revisando código.

Horas de oficina

A veces es muy importante tener una persona con quien hablar y, en esos casos, el horario de oficina puede ser una buena solución. El horario de oficina es un evento programado regularmente (generalmente semanal) durante el cual una o más personas están disponibles para responder preguntas sobre un tema en particular. El horario de oficina casi nunca es la primera opción para compartir conocimientos: si tiene una pregunta urgente, puede ser doloroso esperar la respuesta a la próxima sesión; y si está organizando horas de oficina, toman tiempo y deben promocionarse regularmente. Dicho esto, brindan una manera para que las personas hablen con un experto en persona.

Charlas y clases de tecnología

Google tiene una sólida cultura tanto interna como externa.⁵ Charlas y clases de tecnología. Nuestro equipo de engEDU (Educación en Ingeniería) se enfoca en brindar educación en Ciencias de la Computación a muchas audiencias, desde ingenieros de Google hasta estudiantes de todo el mundo. A un nivel más básico, nuestro programa g2g (Googler2Googler) permite a los Googlers

⁵<https://talksat.withgoogle.com> y <https://www.youtube.com/GoogleTechTalks>, para nombrar unos pocos.

Regístrate para dar o asistir a charlas y clases de otros Googlers.⁶ El programa tiene un gran éxito, con miles de Googlers participantes enseñando temas desde lo técnico (p. ej., "Comprender la vectorización en las CPU modernas") hasta simplemente por diversión (p. ej., "Baile swing para principiantes").

Las charlas técnicas generalmente consisten en un orador que presenta directamente a una audiencia. Las clases, por otro lado, pueden tener un componente de lectura, pero a menudo se centran en ejercicios en clase y, por lo tanto, requieren una participación más activa de los asistentes. Como resultado, las clases dirigidas por un instructor suelen ser más exigentes y costosas de crear y mantener que las charlas técnicas y se reservan para los temas más importantes o difíciles. Dicho esto, una vez que se ha creado una clase, se puede escalar con relativa facilidad porque muchos instructores pueden enseñar una clase con los mismos materiales del curso. Hemos encontrado que las clases tienden a funcionar mejor cuando existen las siguientes circunstancias:

- El tema es lo suficientemente complicado como para que sea una fuente frecuente de malentendidos. Las clases requieren mucho trabajo para crearse, por lo que deben desarrollarse solo cuando aborden necesidades específicas.
- El tema es relativamente estable. Actualizar los materiales de clase es mucho trabajo, por lo que si el tema está evolucionando rápidamente, otras formas de compartir conocimientos tendrán una mejor relación calidad-precio.
- El tema se beneficia de tener maestros disponibles para responder preguntas y brindar ayuda personalizada. Si los estudiantes pueden aprender fácilmente sin ayuda dirigida, los medios de autoservicio como la documentación o las grabaciones son más eficientes. Varias clases introductorias en Google también tienen versiones de autoaprendizaje.
- Hay suficiente demanda para ofrecer la clase regularmente. De lo contrario, los posibles alumnos obtendrán la información que necesitan de otras maneras en lugar de esperar a la clase. En Google, esto es particularmente un problema para las oficinas pequeñas y geográficamente remotas.

Documentación

La documentación es conocimiento escrito cuyo objetivo principal es ayudar a sus lectores a aprender algo. No todo el conocimiento escrito es necesariamente documentación, aunque puede ser útil como prueba en papel. Por ejemplo, es posible encontrar una respuesta a un problema en el hilo de una lista de correo, pero el objetivo principal de la pregunta original en el hilo era buscar respuestas y solo secundariamente documentar la discusión para otros.

6 El programa g2g se detalla en: Laszlo Bock, *Reglas de trabajo: Perspectivas internas de Google que transformarán Cómo vives y lideras* (Nueva York: Doce libros, 2015). Incluye descripciones de diferentes aspectos del programa, así como también cómo evaluar el impacto y recomendaciones sobre en qué enfocarse al establecer programas similares.

En esta sección, nos enfocamos en detectar oportunidades para contribuir y crear documentación formal, desde cosas pequeñas como corregir un error tipográfico hasta esfuerzos más grandes como documentar el conocimiento tribal.



Para una discusión más completa de la documentación, consulte [Capítulo 10](#).

Actualización de documentación

La primera vez que aprende algo es el mejor momento para ver formas de mejorar la documentación y los materiales de capacitación existentes. Para cuando haya absorbido y comprendido un nuevo proceso o sistema, es posible que haya olvidado lo que era difícil o los pasos simples que faltaban en la documentación de "Primeros pasos". En esta etapa, si encuentra un error u omisión en la documentación, jarréglelo! Deja el campamento más limpio de lo que lo encontraste,⁷ e intente actualizar los documentos usted mismo, incluso cuando esa documentación sea propiedad de otra parte de la organización.

En Google, los ingenieros se sienten facultados para actualizar la documentación independientemente de quién sea el propietario, y lo hacemos con frecuencia, incluso si la solución es tan pequeña como corregir un error tipográfico. Este nivel de mantenimiento de la comunidad aumentó notablemente con la introducción de g3doc,⁸ lo que hizo mucho más fácil para los Googlers encontrar un propietario de documentación para revisar su sugerencia. También deja un rastro auditável del historial de cambios que no es diferente al del código.

Creación de documentación

A medida que aumente su competencia, escriba su propia documentación y actualice los documentos existentes. Por ejemplo, si configura un nuevo flujo de desarrollo, documente los pasos. Luego, puede hacer que sea más fácil para otros seguir su camino al señalarles su documento. Aún mejor, haga que sea más fácil para las personas encontrar el documento por sí mismas. Es posible que no exista ninguna documentación suficientemente imposible de encontrar o que no se pueda buscar. Esta es otra área en la que g3doc brilla porque la documentación está ubicada justo al lado del código fuente, a diferencia de un documento o página web (no localizable) en alguna parte.

Finalmente, asegúrese de que haya un mecanismo de retroalimentación. Si no hay una manera fácil y directa para que los lectores indiquen que la documentación está desactualizada o es inexacta, es probable que no se molesten en decírselo a nadie, y el próximo recién llegado se encontrará con el mismo problema.

7 Ver "La regla de los boy scouts" y Kevin Henney, *97 cosas que todo programador debe saber* (Boston: O'Reilly, 2010).

8 g3doc significa "documentación de google3". google3 es el nombre de la encarnación actual del monodepósito de fuentes líticas.

lema Las personas están más dispuestas a aportar cambios si sienten que alguien realmente notará y considerará sus sugerencias. En Google, puede presentar un error de documentación directamente desde el propio documento.

Además, los empleados de Google pueden dejar comentarios fácilmente en las páginas de g3doc. Otros Googlers pueden ver y responder a estos comentarios y, debido a que dejar un comentario automáticamente genera un error para el propietario de la documentación, el lector no necesita averiguar a quién contactar.

Promoción de la documentación

Tradicionalmente, animar a los ingenieros a documentar su trabajo puede ser difícil. Escribir documentación requiere tiempo y esfuerzo que podría dedicarse a la codificación, y los beneficios que resultan de ese trabajo no son inmediatos y en su mayoría son cosechados por otros. Las compensaciones asimétricas como estas son buenas para la organización en su conjunto dado que muchas personas pueden beneficiarse de la inversión de tiempo de unos pocos, pero sin buenos incentivos, puede ser un desafío fomentar ese comportamiento. Discutimos algunos de estos incentivos estructurales en la sección "[Incentivos y reconocimiento](#)" en la página 57.

Sin embargo, el autor de un documento a menudo puede beneficiarse directamente de escribir documentación. Suponga que los miembros del equipo siempre le piden ayuda para depurar ciertos tipos de fallas de producción. La documentación de sus procedimientos requiere una inversión inicial de tiempo, pero una vez que se haya realizado ese trabajo, puede ahorrar tiempo en el futuro al señalar a los miembros del equipo la documentación y brindar ayuda práctica solo cuando sea necesario.

Escribir documentación también ayuda a escalar a su equipo y organización. Primero, la información en la documentación se canoniza como referencia: los miembros del equipo pueden consultar el documento compartido e incluso actualizarlo ellos mismos. En segundo lugar, la canonización puede extenderse fuera del equipo. Quizás algunas partes de la documentación no sean exclusivas de la configuración del equipo y se vuelvan útiles para otros equipos que buscan resolver problemas similares.

Código

A un nivel meta, el código es conocimiento, por lo que el acto mismo de escribir código puede considerarse una forma de transcripción del conocimiento. Si bien el intercambio de conocimientos puede no ser una intención directa del código de producción, a menudo es un efecto secundario emergente, que puede facilitarse mediante la legibilidad y la claridad del código.

La documentación del código es una forma de compartir conocimientos; la documentación clara no solo beneficia a los consumidores de la biblioteca, sino también a los futuros mantenedores. Del mismo modo, los comentarios de implementación transmiten conocimiento a través del tiempo: estás escribiendo estos comentarios expresamente por el bien de los futuros lectores (¡incluido Future You!). En términos de compensaciones, los comentarios del código están sujetos a las mismas desventajas que la documentación general: deben mantenerse activamente o pueden quedar obsoletos rápidamente, como puede atestiguar cualquiera que haya leído un comentario que contradiga directamente el código.

Revisões de código (ver [Capítulo 9](#)) son a menudo una oportunidad de aprendizaje tanto para los autores como para los revisores. Por ejemplo, la sugerencia de un revisor podría presentarle al autor un nuevo patrón de prueba, o un revisor podría enterarse de una nueva biblioteca al ver que el autor la usa en su código. Google estandariza la tutoría a través de la revisión de código con *el proceso de legibilidad*, como se detalla en el estudio de caso al final de este capítulo.

Escalando el conocimiento de su organización

Garantizar que la experiencia se comparte adecuadamente en toda la organización se vuelve más difícil a medida que la organización crece. Algunas cosas, como la cultura, son importantes en cada etapa de crecimiento, mientras que otras, como establecer fuentes canónicas de información, pueden ser más beneficiosas para organizaciones más maduras.

Cultivar una cultura de intercambio de conocimientos

La cultura organizacional es la cosa humana blanda que muchas empresas tratan como una ocurrencia tardía. Pero en Google, creemos que centrarse primero en la cultura y el medio ambiente⁹ genera mejores resultados que centrarse solo en la salida, como el código de ese entorno.

Hacer cambios organizacionales importantes es difícil y se han escrito innumerables libros sobre el tema. No pretendemos tener todas las respuestas, pero podemos compartir los pasos específicos que ha tomado Google para crear una cultura que promueva el aprendizaje.

ver el libro; *Reglas de trabajo*¹⁰ para un examen más profundo de la cultura de Google.

⁹ Laszló Bock, *Reglas de trabajo: Información desde dentro de Google que transformará su forma de vivir y liderar* (Nueva York: Doce Libros, 2015).

¹⁰ Ibíd.

El respeto

El mal comportamiento de unos pocos individuos **puede hacer que todo un equipo o comunidad sea poco acogedor**. En tal entorno, los novatos aprenden a llevar sus preguntas a otra parte y los nuevos expertos potenciales dejan de intentarlo y no tienen espacio para crecer. En el peor de los casos, el grupo se reduce a sus miembros más tóxicos. Puede ser difícil recuperarse de este estado.

El intercambio de conocimientos puede y debe hacerse con amabilidad y respeto. En tecnología, la tolerancia (o peor aún, la reverencia) hacia el "imbécil brillante" es omnipresente y dañina, pero ser un experto y ser amable no se excluyen mutuamente. La sección de Liderazgo de la escalera laboral de ingeniería de software de Google describe esto claramente:

Aunque se espera una medida de liderazgo técnico en los niveles superiores, no todo el liderazgo está dirigido a los problemas técnicos. Los líderes mejoran la calidad de las personas que los rodean, mejoran la seguridad psicológica del equipo, crean una cultura de trabajo en equipo y colaboración, calman las tensiones dentro del equipo, dan ejemplo de la cultura y los valores de Google y hacen de Google un lugar de trabajo más vibrante y emocionante.. Los idiotas no son buenos líderes.

Esta expectativa está modelada por el liderazgo senior: Urs Hözlle (Vicepresidente senior de Infraestructura Técnica) y Ben Treynor Sloss (Vicepresidente, Fundador de Google SRE) escribieron un documento interno citado con frecuencia ("No idiotas") sobre por qué los Googlers deberían preocuparse sobre el comportamiento respetuoso en el trabajo y qué hacer al respecto.

Incentivos y reconocimiento

La buena cultura debe fomentarse activamente, y fomentar una cultura de intercambio de conocimientos requiere el compromiso de reconocerla y recompensarla a nivel sistémico. Es un error común para las organizaciones hablar de boquilla sobre un conjunto de valores mientras recompensan activamente el comportamiento que no hace cumplir esos valores. Las personas reaccionan a los incentivos en lugar de los lugares comunes, por lo que es importante poner su dinero donde está su boca al establecer un sistema de compensación y premios.

Google utiliza una variedad de mecanismos de reconocimiento, desde los estándares de toda la empresa, como la revisión del desempeño y los criterios de promoción, hasta los premios de igual a igual entre los Googlers.

Nuestra escalera de ingeniería de software, que usamos para calibrar recompensas como compensación y promoción en toda la empresa, alienta a los ingenieros a compartir conocimientos al señalar estas expectativas explícitamente. En los niveles superiores, la escalera explícitamente destaca la importancia de una influencia más amplia, y esta expectativa aumenta a medida que aumenta la antigüedad. En los niveles más altos, los ejemplos de liderazgo incluyen lo siguiente:

- Desarrollar futuros líderes sirviendo como mentores para el personal subalterno, ayudándolos a desarrollarse tanto técnicamente como en su función de Google

- Sostener y desarrollar la comunidad de software en Google a través de revisiones de código y diseño, educación y desarrollo de ingeniería y orientación experta para otros en el campo.



Ver Capítulos 5 y 6 para obtener más información sobre el liderazgo.

Las expectativas de la escalera laboral son una forma de arriba hacia abajo para dirigir una cultura, pero la cultura también se forma de abajo hacia arriba. En Google, el programa de bonificación entre pares es una forma de adoptar la cultura de abajo hacia arriba. Los bonos de pares son un premio monetario y un reconocimiento formal que cualquier Googler puede otorgar a cualquier otro Googler por un trabajo superior.¹¹ Por ejemplo, cuando Ravi envía un bono de compañeros a Julia por ser una de las principales colaboradoras de una lista de correo (respondiendo regularmente preguntas que benefician a muchos lectores), está reconociendo públicamente su trabajo de intercambio de conocimientos y su impacto más allá de su equipo. Debido a que las bonificaciones entre compañeros son impulsadas por los empleados, no por la gerencia, pueden tener un efecto de base importante y poderoso.

Similares a las bonificaciones entre pares son las felicitaciones: el reconocimiento público de las contribuciones (generalmente de menor impacto o esfuerzo que aquellos que merecen una bonificación entre pares) que aumentan la visibilidad de las contribuciones entre pares.

Cuando un Googler le da a otro Googler una bonificación o felicitaciones, puede optar por copiar grupos o personas adicionales en el correo electrónico del premio, lo que aumenta el reconocimiento del trabajo del compañero. También es común que el gerente del destinatario reenvíe el correo electrónico del premio al equipo para celebrar los logros de los demás.

Un sistema en el que las personas puedan reconocer formal y fácilmente a sus compañeros es una herramienta poderosa para animar a los compañeros a seguir haciendo las cosas maravillosas que hacen. No es la bonificación lo que importa: es el reconocimiento de los compañeros.

Establecimiento de fuentes canónicas de información

Las fuentes canónicas de información son corpus de información centralizados en toda la empresa que proporcionan una forma de estandarizar y propagar el conocimiento experto. Funcionan mejor con información que es relevante para todos los ingenieros dentro de la organización, que de otro modo es propensa a islas de información. Por ejemplo, una guía para configurar un

¹¹ Los bonos de pares incluyen un premio en efectivo y un certificado, además de ser una parte permanente del premio de un Googler. registro en una herramienta interna llamada gThanks.

El flujo de trabajo básico del desarrollador debe hacerse canónico, mientras que una guía para ejecutar una instancia local de Frobber es más relevante solo para los ingenieros que trabajan en Frobber.

Establecer fuentes de información canónicas requiere una mayor inversión que mantener información más localizada, como la documentación del equipo, pero también tiene beneficios más amplios.

Proporcionar referencias centralizadas para toda la organización hace que la información ampliamente requerida sea más fácil y más predecible para encontrar y contrarresta los problemas de fragmentación de la información que pueden surgir cuando varios equipos que se enfrentan a problemas similares producen sus propias guías, a menudo contradictorias.

Debido a que la información canónica es muy visible y tiene la intención de proporcionar una comprensión compartida a nivel de la organización, es importante que expertos en la materia mantengan y examinen activamente el contenido. Cuanto más complejo es un tema, más crítico es que el contenido canónico tenga propietarios explícitos. Los lectores bien intencionados pueden ver que algo está desactualizado pero carecen de la experiencia para realizar los cambios estructurales significativos necesarios para solucionarlo, incluso si las herramientas facilitan la sugerencia de actualizaciones.

Crear y mantener fuentes de información canónicas y centralizadas es costoso y requiere mucho tiempo, y no es necesario compartir todo el contenido a nivel organizacional. Al considerar cuánto esfuerzo invertir en este recurso, tenga en cuenta a su audiencia. ¿Quién se beneficia de esta información? ¿Tú? ¿Tu equipo? ¿Su área de productos? ¿Todos los ingenieros?

Guías para desarrolladores

Google tiene un conjunto amplio y profundo de orientación oficial para ingenieros, que incluye [guías de estilo](#), mejores prácticas oficiales de ingeniería de software,¹² [guías para la revisión de código](#),¹³ [pruebas](#),¹⁴ y [Consejos de la Semana \(TotW\)](#).¹⁵

El corpus de información es tan grande que no es práctico esperar que los ingenieros lo lean de principio a fin, y mucho menos que puedan absorber tanta información a la vez. En cambio, un experto humano que ya esté familiarizado con una directriz puede enviar un enlace a un compañero ingeniero, quien luego puede leer la referencia y obtener más información. El experto ahorra tiempo al no tener que explicar personalmente una práctica de toda la empresa, y el alumno ahora sabe que existe una fuente canónica de información confiable a la que puede acceder cuando sea necesario. Dicho proceso escala el conocimiento porque permite que los expertos humanos reconozcan y resuelvan una necesidad de información específica aprovechando recursos escalables comunes.

12 Tales como libros sobre ingeniería de software en Google.

13 Ver [Capítulo 9](#).

14 Ver [Capítulo 11](#).

15 Disponible para varios idiomas. Disponible externamente para C++ en <https://abseil.io/tips>.

ir/enlaces

Los enlaces go/ (a veces denominados enlaces goto/) son el acortador de URL interno de Google.¹⁶

La mayoría de las referencias internas de Google tienen al menos un enlace go/ interno. Por ejemplo, “go/ spanner” proporciona información sobre Spanner, “go/python” es la guía para desarrolladores de Python de Google. El contenido puede vivir en cualquier repositorio (g3doc, Google Drive, Google Sites, etc.), pero tener un enlace go/ que apunte a él proporciona una forma predecible y memorable de acceder a él. Esto produce algunos beneficios agradables:

- Los enlaces go/ son tan cortos que es fácil compartirlos en una conversación (“¡Deberías ver go/frobber!”). Esto es mucho más fácil que tener que ir a buscar un enlace y luego enviar un mensaje a todas las partes interesadas. Tener una forma sencilla de compartir referencias hace que sea más probable que ese conocimiento se comparta en primer lugar.
- Los enlaces go/ proporcionan un enlace permanente al contenido, incluso si cambia la URL subyacente. Cuando un propietario mueve contenido a un repositorio diferente (por ejemplo, al mover contenido de un documento de Google a g3doc), simplemente puede actualizar la URL de destino del enlace go/. El enlace go/ en sí permanece sin cambios.

Los enlaces go/ están tan arraigados en la cultura de Google que ha surgido un círculo virtuoso: un Googler que busque información sobre Frobber probablemente primero consulte go/frobber. Si el enlace go/ no apunta a la Guía para desarrolladores de Frobber (como se esperaba), Googler generalmente configurará el enlace por sí mismo. Como resultado, los Googlers generalmente pueden adivinar el enlace go/ correcto en el primer intento.

Laboratorios de código

Los codelabs de Google son tutoriales prácticos y guiados que enseñan a los ingenieros nuevos conceptos o procesos mediante la combinación de explicaciones, código de ejemplo de prácticas recomendadas y ejercicios de código.¹⁷ Una colección canónica de codelabs para tecnologías ampliamente utilizadas en Google está disponible en go/codelab. Estos codelabs pasan por varias rondas de revisión y prueba formales antes de su publicación. Los Codelabs son un punto intermedio interesante entre la documentación estática y las clases dirigidas por un instructor, y comparten las mejores y las peores características de cada una. Su naturaleza práctica los hace más atractivos que la documentación tradicional, pero los ingenieros aún pueden acceder a ellos a pedido y completarlos por su cuenta; pero su mantenimiento es costoso y no se adaptan a las necesidades específicas del alumno.

16 enlaces go/ no están relacionados con el idioma Go.

17 Codelabs externos están disponibles en <https://codelabs.developers.google.com>.

Análisis estático

Las herramientas de análisis estático son una forma poderosa de compartir las mejores prácticas que se pueden verificar mediante programación. Cada lenguaje de programación tiene sus propias herramientas de análisis estático particulares, pero tienen el mismo propósito general: alertar a los autores y revisores de código sobre las formas en que se puede mejorar el código para seguir el estilo y las mejores prácticas. Algunas herramientas van un paso más allá y ofrecen aplicar automáticamente esas mejoras al código.



Vea [capítulo 20](#) para obtener detalles sobre las herramientas de análisis estático y cómo se utilizan en Google.

La configuración de herramientas de análisis estático requiere una inversión inicial, pero tan pronto como están en su lugar, se escalan de manera eficiente. Cuando se agrega una verificación de una mejor práctica a una herramienta, cada ingeniero que usa esa herramienta se da cuenta de esa mejor práctica. Esto también libera a los ingenieros para que enseñen otras cosas: el tiempo y el esfuerzo que habrían invertido en enseñar manualmente las mejores prácticas (ahora automatizadas) se pueden usar para enseñar otra cosa. Las herramientas de análisis estático aumentan el conocimiento de los ingenieros. Permiten que una organización aplique más mejores prácticas y las aplique de manera más consistente de lo que sería posible de otro modo.

Mantenerse en el bucle

Cierta información es crítica para hacer el trabajo de uno, como saber cómo hacer un flujo de trabajo de desarrollo típico. Otra información, como actualizaciones sobre herramientas de productividad populares, es menos crítica pero sigue siendo útil. Para este tipo de conocimiento, la formalidad del medio de intercambio de información depende de la importancia de la información que se entrega. Por ejemplo, los usuarios esperan que la documentación oficial se mantenga actualizada, pero normalmente no tienen esa expectativa para el contenido del boletín, que por lo tanto requiere menos mantenimiento por parte del propietario.

Boletines

Google tiene una serie de boletines informativos para toda la empresa que se envían a todos los ingenieros, incluidos *EspNoticias* (noticias de ingeniería), *propio* (Noticias de privacidad/seguridad), y *Grandes éxitos de Google* (informe de los cortes más interesantes del trimestre). Son una buena manera de comunicar información que es de interés para los ingenieros pero que no es de misión crítica. Para este tipo de actualización, descubrimos que los boletines obtienen una mayor participación cuando se envían con menos frecuencia y contienen contenido más útil e interesante. De lo contrario, los boletines pueden percibirse como spam.

Aunque la mayoría de los boletines de Google se envían por correo electrónico, algunos son más creativos en su distribución. *Prueba en el inodoro* (consejos de prueba) y *Aprendiendo en el baño* (producción

consejos de actividad) son boletines de una sola página publicados dentro de los retretes. Este medio de entrega único ayuda al *Prueba en el inodoro* y *Aprendiendo en el baño* se destacan de otros boletines, y todos los números se archivan en línea.



Ver [Capítulo 11](#) para una historia de cómo *Prueba en el inodoro* vino a ser.

Comunidades

A los empleados de Google les gusta formar comunidades entre organizaciones en torno a diversos temas para compartir conocimientos. Estos canales abiertos facilitan aprender de otras personas fuera de su círculo inmediato y evitan islas de información y duplicación. Los Grupos de Google son especialmente populares: Google tiene miles de grupos internos con diferentes niveles de formalidad. Algunos están dedicados a la resolución de problemas; otros, como el grupo Code Health, son más para discusión y orientación. El Google+ interno también es popular entre los usuarios de Google como fuente de información informal porque las personas publican desgloses técnicos interesantes o detalles sobre los proyectos en los que están trabajando.

Legibilidad: Tutoría estandarizada a través de la revisión del código

En Google, la "legibilidad" se refiere a algo más que la legibilidad del código; es un proceso de tutoría estandarizado en todo Google para difundir las mejores prácticas del lenguaje de programación. La legibilidad cubre una amplia gama de conocimientos, incluidos, entre otros, expresiones idiomáticas, estructura de código, diseño de API, uso adecuado de bibliotecas comunes, documentación y cobertura de pruebas.

La legibilidad comenzó como un esfuerzo de una sola persona. En los primeros días de Google, Craig Silverstein (identificación de empleado n.º 3) se sentaba en persona con cada nuevo empleado y hacía una "revisión de legibilidad" línea por línea de su primera confirmación de código importante. Fue una revisión minuciosa que cubrió todo, desde las formas en que el código podría mejorarse hasta las convenciones de espacios en blanco. Esto le dio a la base de código de Google una apariencia uniforme pero, lo que es más importante, enseñó las mejores prácticas, destacó qué infraestructura compartida estaba disponible y mostró a los nuevos empleados cómo es escribir código en Google.

Inevitablemente, la tasa de contratación de Google creció más allá de lo que una persona podía seguir. Tantos ingenieros encontraron valioso el proceso que ofrecieron su propio tiempo para escalar el programa. En la actualidad, alrededor del 20 % de los ingenieros de Google participan en el proceso de legibilidad en un momento dado, ya sea como revisores o autores de código.

¿Qué es el proceso de legibilidad?

La revisión del código es obligatoria en Google. Cada lista de cambios (CL)¹⁸ requiere *aprobación de legibilidad*, lo que indica que alguien que tiene *certificación de legibilidad* para ese idioma ha aprobado la CL. Los autores certificados proporcionan implícitamente la aprobación de la legibilidad de sus propias CL; de lo contrario, uno o más revisores calificados deben aprobar explícitamente la legibilidad de la CL. Este requisito se agregó después de que Google creció hasta un punto en el que ya no era posible exigir que todos los ingenieros recibieran revisiones de código que enseñaran las mejores prácticas con el rigor deseado.



Ver [Capítulo 9](#) para obtener una descripción general del proceso de revisión del código de Google y lo que significa Aprobación en este contexto.

Dentro de Google, tener una certificación de legibilidad se conoce comúnmente como "tener legibilidad" para un idioma. Los ingenieros con legibilidad han demostrado que constantemente escriben código claro, idiomático y mantenible que ejemplifica las prácticas recomendadas y el estilo de codificación de Google para un idioma determinado. Lo hacen enviando CL a través del proceso de legibilidad, durante el cual un grupo centralizado de revisores de legibilidad revisa los CL y brinde comentarios sobre cuánto demuestra las diversas áreas de dominio. A medida que los autores internalizan las pautas de legibilidad, reciben cada vez menos comentarios en sus CL hasta que finalmente se gradúan del proceso y reciben formalmente la legibilidad. La legibilidad conlleva una mayor responsabilidad: se confía en los ingenieros con legibilidad para que sigan aplicando sus conocimientos a su propio código y actúen como revisores del código de otros ingenieros.

Alrededor del 1 al 2% de los ingenieros de Google son revisores de legibilidad. Todos los revisores son voluntarios, y cualquier persona con legibilidad puede autonominarse para convertirse en revisor de legibilidad. Los revisores de legibilidad están sujetos a los más altos estándares porque se espera que no solo tengan una gran experiencia en el lenguaje, sino también aptitudes para enseñar a través de la revisión de código. Se espera que traten la legibilidad ante todo como un proceso cooperativo y de tutoría, no como un proceso de control o confrontación. Se alienta a los revisores de legibilidad y a los autores de CL a tener discusiones durante el proceso de revisión. Los revisores proporcionan citas relevantes para sus comentarios para que los autores puedan aprender sobre los fundamentos que se incluyeron en las pautas de estilo ("valla de Chesterton"). Si la justificación de una directriz dada no está clara,

¹⁸A *lista de cambios* es una lista de archivos que componen un cambio en un sistema de control de versiones. Una lista de cambios es sinónimo con un *conjunto de cambios*.

La legibilidad es deliberadamente un proceso impulsado por humanos que tiene como objetivo escalar el conocimiento de una manera estandarizada pero personalizada. Como una combinación complementaria de conocimiento escrito y tribal, la legibilidad combina las ventajas de la documentación escrita, a la que se puede acceder con referencias citables, con las ventajas de revisores humanos expertos, que saben qué pautas citar. Las pautas canónicas y las recomendaciones de lenguaje están ampliamente documentadas, ¡lo cual es bueno!, pero el corpus de información es tan grande¹⁹que puede ser abrumador, especialmente para los recién llegados.

¿Por qué tener este proceso?

El código se lee mucho más de lo que se escribe, y este efecto se magnifica a la escala de Google y en nuestro (muy grande) monorepo.²⁰Cualquier ingeniero puede mirar y aprender de la gran cantidad de conocimiento que es el código de otros equipos y herramientas poderosas como Kythé facilitar la búsqueda de referencias en todo el código base (vercapítulo 17). Una característica importante de las mejores prácticas documentadas (verCapítulo 8) es que proporcionan estándares consistentes para que los siga todo el código de Google. La legibilidad es tanto un mecanismo de aplicación como de propagación de estos estándares.

Una de las principales ventajas del programa de legibilidad es que expone a los ingenieros a algo más que el conocimiento tribal de su propio equipo. Para obtener legibilidad en un idioma determinado, los ingenieros deben enviar CL a través de un conjunto centralizado de revisores de legibilidad que revisan el código en toda la empresa. Centralizar el proceso genera una compensación significativa: el programa se limita a escalar linealmente en lugar de sublinealmente con el crecimiento de la organización, pero hace que sea más fácil hacer cumplir la coherencia, evitar islas y evitar (a menudo involuntario) desviarse de las normas establecidas.

El valor de la coherencia en toda la base de código no se puede subestimar: incluso con decenas de miles de ingenieros escribiendo código durante décadas, garantiza que el código en un idioma determinado se verá similar en todo el corpus. Esto permite a los lectores concentrarse en lo que hace el código en lugar de distraerse pensando en por qué se ve diferente al código al que están acostumbrados. Autores de cambios a gran escala (vercapítulo 22) puede realizar cambios más fácilmente en todo el monorepo, cruzando los límites de miles de equipos. Las personas pueden cambiar de equipo y estar seguros de que la forma en que el nuevo equipo usa un idioma determinado no es muy diferente a la del equipo anterior.

19 A partir de 2019, solo la guía de estilo de Google C++ tiene 40 páginas. El material secundario que constituye el completo corpus de mejores prácticas es muchas veces más largo.

20 Para saber por qué Google usa un monorepo, consulte <https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billones-de-lneas-de-código-en-un-único-repositorio/texto-completo>. Tenga en cuenta también que no todo el código de Google vive dentro del monorepo; la legibilidad como se describe aquí se aplica solo al monorepo porque es una noción de consistencia dentro del repositorio.

Estos beneficios tienen algunos costos: la legibilidad es un proceso pesado en comparación con otros medios, como la documentación y las clases, porque es obligatorio y lo aplican las herramientas de Google (ver[capítulo 19](#)). Estos costos no son triviales e incluyen lo siguiente:

- Mayor fricción para los equipos que no tienen ningún miembro del equipo con legibilidad, porque necesitan encontrar revisores externos a su equipo para aprobar la legibilidad de las CL.
- Potencial para rondas adicionales de revisión de código para autores que necesitan revisión de legibilidad.
- Desventajas de escala de ser un proceso impulsado por humanos. Limitado a escalar linealmente al crecimiento de la organización porque depende de revisores humanos que realizan revisiones de código especializadas.

La pregunta, entonces, es si los beneficios superan los costos. También está el factor del tiempo: el efecto total de los beneficios versus los costos no están en la misma escala de tiempo. El programa hace una compensación deliberada de mayor latencia de revisión de código a corto plazo y costos iniciales por los beneficios a largo plazo de código de mayor calidad, consistencia de código en todo el repositorio y mayor experiencia de ingeniería. La escala de tiempo más larga de los beneficios viene con la expectativa de que el código se escriba con una vida útil potencial de años, si no décadas.²¹

Como ocurre con la mayoría de los procesos de ingeniería, o quizás con todos, siempre hay espacio para mejorar. Algunos de los costos se pueden mitigar con herramientas. Varios comentarios de legibilidad abordan problemas que podrían detectarse estáticamente y comentarse automáticamente mediante herramientas de análisis estático. A medida que continuamos invirtiendo en análisis estático, los revisores de legibilidad pueden centrarse cada vez más en áreas de orden superior, como si un bloque de código en particular es comprensible para lectores externos que no están íntimamente familiarizados con el código base en lugar de detecciones automatizables como si una línea tiene espacios en blanco al final.

Pero las aspiraciones no son suficientes. La legibilidad es un programa controvertido: algunos ingenieros se quejan de que es un obstáculo burocrático innecesario y un mal uso del tiempo del ingeniero. ¿Valen la pena las ventajas y desventajas de la legibilidad? Para obtener la respuesta, recurrimos a nuestro confiable equipo de Investigación de productividad de ingeniería (EPR).

21 Por esta razón, el código que se sabe que tiene un período de tiempo corto está exento de los requisitos de legibilidad. Examen-por favor incluya el [experimental](#) (explicítamente designado para código experimental y no puede pasar a producción) y el [Programa del área 120](#), un taller para los productos experimentales de Google.

El equipo de EPR realizó estudios profundos de legibilidad, que incluyeron, entre otros, si las personas se vieron obstaculizadas por el proceso, aprendieron algo o cambiaron su comportamiento después de graduarse. Estos estudios demostraron que la legibilidad tiene un impacto neto positivo en la velocidad de ingeniería. Las CL de autores con legibilidad tardan estadísticamente significativamente menos tiempo en revisarse y enviarse que las CL de autores que no tienen legibilidad.²² La satisfacción de los ingenieros autoinformados con la calidad de su código, al carecer de medidas más objetivas para la calidad del código, es mayor entre los ingenieros que tienen legibilidad en comparación con los que no. Una gran mayoría de los ingenieros que completan el programa informan estar satisfechos con el proceso y encuentran que vale la pena. Informan que aprendieron de los revisores y cambiaron su propio comportamiento para evitar problemas de legibilidad al escribir,y código de revisión.



Para ver en profundidad este estudio y la investigación de productividad de ingeniería interna de Google, consulte[Capítulo 7](#).

Google tiene una cultura muy fuerte de revisión de código y la legibilidad es una extensión natural de esa cultura. La legibilidad pasó de la pasión de un solo ingeniero a un programa formal de expertos humanos que asesoraban a todos los ingenieros de Google. Evolucionó y cambió con el crecimiento de Google, y seguirá evolucionando a medida que cambien las necesidades de Google.

Conclusión

El conocimiento es, de alguna manera, el capital más importante (aunque intangible) de una organización de ingeniería de software, y compartir ese conocimiento es crucial para hacer que una organización sea resistente y redundante frente al cambio. Una cultura que promueve el intercambio de conocimientos abierto y honesto distribuye ese conocimiento de manera eficiente en toda la organización y permite que la organización crezca con el tiempo. En la mayoría de los casos, las inversiones para facilitar el intercambio de conocimientos cosechan muchos dividendos a lo largo de la vida de una empresa.

²² Esto incluye el control de una variedad de factores, incluida la permanencia en Google y el hecho de que las CL de los autores quienes no tienen legibilidad normalmente necesitan rondas adicionales de revisión en comparación con los autores que ya tienen legibilidad.

TL; DR

- *Seguridad psicológica*es la base para fomentar un entorno de intercambio de conocimientos.
- Comience poco a poco: haga preguntas y escriba las cosas.
- Facilite que las personas obtengan la ayuda que necesitan tanto de expertos humanos como de referencias documentadas.
- A nivel sistémico, anime y recompense a aquellos que se toman el tiempo para enseñar y ampliar su experiencia más allá de ellos mismos, su equipo o su organización.
- No existe una bala de plata: empoderar una cultura de intercambio de conocimientos requiere una combinación de múltiples estrategias, y la combinación exacta que funcione mejor para su organización probablemente cambiará con el tiempo.

Ingeniería para la Equidad

*Escrito por Demma Rodriguez
Editado por Riona MacNamara*

En capítulos anteriores, hemos explorado el contraste entre la programación como la producción de código que aborda el problema del momento y la ingeniería de software como la aplicación más amplia de código, herramientas, políticas y procesos a un problema dinámico y ambiguo que puede abarcar décadas o incluso vidas. En este capítulo, discutiremos las responsabilidades únicas de un ingeniero al diseñar productos para una amplia base de usuarios. Además, evaluamos cómo una organización, al aceptar la diversidad, puede diseñar sistemas que funcionen para todos y evitar perpetuar el daño contra nuestros usuarios.

Tan nuevo como es el campo de la ingeniería de software, somos aún más nuevos en la comprensión del impacto que tiene en las personas subrepresentadas y las sociedades diversas. No escribimos este capítulo porque sabemos todas las respuestas. Nosotros no. De hecho, Google está aprendiendo a comprender cómo diseñar productos que empoderen y respeten a todos nuestros usuarios. Hemos tenido muchas fallas públicas en la protección de nuestros usuarios más vulnerables, por lo que escribimos este capítulo porque el camino hacia productos más equitativos comienza con la evaluación de nuestras propias fallas y el fomento del crecimiento.

También estamos escribiendo este capítulo debido al creciente desequilibrio de poder entre aquellos que toman decisiones de desarrollo que impactan al mundo y aquellos que simplemente deben aceptar y vivir con esas decisiones que a veces ponen en desventaja a comunidades ya marginadas a nivel mundial. Es importante compartir y reflexionar sobre lo que hemos aprendido hasta ahora con la próxima generación de ingenieros de software. Es aún más importante que ayudemos a influir en la próxima generación de ingenieros para que sean mejores de lo que somos hoy.

El simple hecho de leer este libro significa que probablemente aspire a ser un ingeniero excepcional. Quiere resolver problemas. Aspira a crear productos que generen resultados positivos para la base más amplia de personas, incluidas las personas a las que es más difícil llegar. Para hacer esto, deberá considerar cómo se aprovecharán las herramientas que construya para cambiar la trayectoria de la humanidad, con suerte para mejor.

El sesgo es el predeterminado

Cuando los ingenieros no se enfocan en usuarios de diferentes nacionalidades, etnias, razas, géneros, edades, niveles socioeconómicos, habilidades y sistemas de creencias, incluso el personal más talentoso fallará inadvertidamente a sus usuarios. Tales fallas a menudo no son intencionales; todas las personas tienen ciertos prejuicios, y los científicos sociales han reconocido durante las últimas décadas que la mayoría de las personas exhiben prejuicios inconscientes, que refuerzan y promulgan los estereotipos existentes. El sesgo inconsciente es insidioso y, a menudo, más difícil de mitigar que los actos intencionales de exclusión. Incluso cuando queremos hacer lo correcto, es posible que no reconozcamos nuestros propios prejuicios. De la misma manera, nuestras organizaciones también deben reconocer que tal sesgo existe y trabajar para abordarlo en sus fuerzas laborales, desarrollo de productos y alcance a los usuarios.

Debido al sesgo, Google a veces no ha logrado representar a los usuarios de manera equitativa dentro de sus productos, con lanzamientos en los últimos años que no se enfocaron lo suficiente en grupos subrepresentados. Muchos usuarios atribuyen nuestra falta de conocimiento en estos casos al hecho de que nuestra población de ingenieros es mayoritariamente masculina, mayoritariamente blanca o asiática, y ciertamente no es representativa de todas las comunidades que utilizan nuestros productos. La falta de representación de dichos usuarios en nuestra plantilla significa que a menudo no tenemos la diversidad necesaria para comprender cómo el uso de nuestros productos puede afectar a los usuarios vulnerables o subrepresentados.

Estudio de caso: Google pierde la marca en la inclusión racial

En 2015, el ingeniero de software Jacky Alciné señaló¹ que los algoritmos de reconocimiento de imágenes de Google Photos clasificaban a sus amigos negros como "gorilas". Google fue lento en responder a estos errores e incompleto al abordarlos.

¿Qué causó un fracaso tan monumental? Varias cosas:

- Los algoritmos de reconocimiento de imágenes dependen de que se les suministre un conjunto de datos "adecuado" (que a menudo significa "completo"). Los datos de las fotos introducidos en el algoritmo de reconocimiento de imágenes de Google estaban claramente incompletos. En resumen, los datos no representaban a la población.

¹Informe de diversidad de Google 2019.

² @jackyalcine. 2015. "Google Photos, todos jodidos. Mi amigo no es un gorila". Twitter, 29 de junio de 2015. <https://twitter.com/jackyalcine/status/615329515909156865>.

- Google mismo (y la industria tecnológica en general) no tenía (y no tiene) mucha representación negra,³ y eso afecta las decisiones subjetivas en el diseño de dichos algoritmos y la recopilación de dichos conjuntos de datos. El sesgo inconsciente de la propia organización probablemente condujo a que se dejara sobre la mesa un producto más representativo.
- El mercado objetivo de Google para el reconocimiento de imágenes no incluía adecuadamente a estos grupos subrepresentados. Las pruebas de Google no detectaron estos errores; como resultado, nuestros usuarios lo hicieron, lo que avergonzó a Google y perjudicó a nuestros usuarios.

Todavía en 2018, Google aún no había abordado adecuadamente el problema subyacente.⁴

En este ejemplo, nuestro producto se diseñó y ejecutó de manera inadecuada, al no considerar adecuadamente a todos los grupos raciales y, como resultado, falló a nuestros usuarios y causó mala prensa a Google. Otra tecnología sufre fallas similares: el autocompletado puede arrojar resultados ofensivos o racistas. El sistema de anuncios de Google podría manipularse para mostrar anuncios racistas u ofensivos. Es posible que YouTube no capte el discurso de odio, aunque técnicamente está prohibido en esa plataforma.

En todos estos casos, la tecnología en sí no es realmente la culpable. Autocompletar, por ejemplo, no fue diseñado para dirigirse a los usuarios o para discriminar. Pero tampoco fue lo suficientemente resistente en su diseño para excluir el lenguaje discriminatorio que se considera discurso de odio. Como resultado, el algoritmo arrojó resultados que causaron daño a nuestros usuarios. El daño a Google también debería ser obvio: reducción de la confianza del usuario y del compromiso con la empresa. Por ejemplo, los solicitantes negros, latinos y judíos podrían perder la fe en Google como plataforma o incluso como un entorno inclusivo en sí mismo, lo que socavaría el objetivo de Google de mejorar la representación en la contratación.

¿Cómo pudo pasar esto? Después de todo, Google contrata tecnólogos con una educación impecable y/o experiencia profesional: programadores excepcionales que escriben el mejor código y prueban su trabajo. "Construir para todos" es una declaración de marca de Google, pero la verdad es que todavía tenemos un largo camino por recorrer antes de poder afirmar que lo hacemos. Una forma de abordar estos problemas es ayudar a la propia organización de ingeniería de software a parecerse a las poblaciones para las que construimos productos.

3 Muchos informes en 2018-2019 señalaron una falta de diversidad en la tecnología. Algunos notables incluyen el [Nacional Centro para la Mujer y Tecnología de la Información](#), y [Diversidad en tecnología](#).

4 Tom Simonite, "Cuando se trata de gorilas, Google Photos permanece ciego" *cableado*, 11 de enero de 2018.

Entender la necesidad de la diversidad

En Google, creemos que ser un ingeniero excepcional requiere que también se centre en incorporar diversas perspectivas al diseño y la implementación de productos. También significa que los Googlers responsables de contratar o entrevistar a otros ingenieros deben contribuir a crear una fuerza laboral más representativa. Por ejemplo, si entrevista a otros ingenieros para puestos en su empresa, es importante saber cómo se producen los resultados sesgados en la contratación. Existen prerequisitos significativos para entender cómo anticipar el daño y prevenirlo. Para llegar al punto en que podamos construir para todos, primero debemos comprender a nuestras poblaciones representativas. Necesitamos alentar a los ingenieros a tener un alcance más amplio de capacitación educativa.

La primera orden del día es desbaratar la noción de que, como persona con un título en informática y/o experiencia laboral, tienes todas las habilidades que necesitas para convertirte en un ingeniero excepcional. Un título en informática es a menudo una base necesaria. Sin embargo, el título por sí solo (incluso cuando se combina con la experiencia laboral) no lo convertirá en ingeniero. También es importante desbaratar la idea de que solo las personas con títulos en informática pueden diseñar y construir productos. Hoy, *la mayoría de los programadores tienen un título en informática*; tienen éxito en la construcción de códigos, el establecimiento de teorías de cambio y la aplicación de metodologías para la resolución de problemas. Sin embargo, como demuestran los ejemplos anteriores, *este enfoque es insuficiente para una ingeniería inclusiva y equitativa*.

Los ingenieros deben comenzar por enfocar todo el trabajo dentro del marco del ecosistema completo en el que buscan influir. Como mínimo, necesitan comprender la demografía de la población de sus usuarios. Los ingenieros deben centrarse en las personas que son diferentes a ellos, especialmente las personas que podrían intentar usar sus productos para causar daño. Los usuarios más difíciles de considerar son aquellos que están privados de sus derechos por los procesos y el entorno en el que acceden a la tecnología. Para abordar este desafío, los equipos de ingeniería deben ser representativos de sus usuarios actuales y futuros. En ausencia de una representación diversa en los equipos de ingeniería, los ingenieros individuales deben aprender a construir para todos los usuarios.

Creación de capacidad multicultural

Una marca de un ingeniero excepcional es la capacidad de comprender cómo los productos pueden beneficiar y perjudicar a diferentes grupos de seres humanos. Se espera que los ingenieros tengan aptitudes técnicas, pero también deben tener la *discernimiento* saber cuándo construir algo y cuándo no. El discernimiento incluye desarrollar la capacidad para identificar y rechazar características o productos que generan resultados adversos. Este es un objetivo elevado y difícil, porque hay una enorme cantidad de individualismo que implica ser un ingeniero de alto rendimiento. Sin embargo, para tener éxito, debemos extender nuestra

centrarse más allá de nuestras propias comunidades a los próximos mil millones de usuarios o a los usuarios actuales que podrían verse privados de sus derechos o dejados atrás por nuestros productos.

Con el tiempo, puede crear herramientas que miles de millones de personas usan a diario: herramientas que influyen en cómo piensan las personas sobre el valor de las vidas humanas, herramientas que monitorean la actividad humana y herramientas que capturan y conservan datos confidenciales, como imágenes de sus hijos. - hijos y seres queridos, así como otro tipo de datos sensibles. Como ingeniero, es posible que ejerza más poder del que cree: el poder de cambiar literalmente la sociedad. Es fundamental que en su viaje para convertirse en un ingeniero excepcional, comprenda la responsabilidad innata necesaria para ejercer el poder sin causar daño. El primer paso es reconocer el estado predeterminado de su sesgo causado por muchos factores sociales y educativos. Una vez que reconozca esto, podrá considerar los casos de uso a menudo olvidados o los usuarios que pueden beneficiarse o verse perjudicados por los productos que crea.

La industria continúa avanzando, creando nuevos casos de uso para la inteligencia artificial (IA) y el aprendizaje automático a una velocidad cada vez mayor. Para seguir siendo competitivos, nos dirigimos hacia la escala y la eficacia en la creación de una fuerza laboral de ingeniería y tecnología de gran talento. Sin embargo, debemos hacer una pausa y considerar el hecho de que hoy en día, algunas personas tienen la capacidad de diseñar el futuro de la tecnología y otras no. Necesitamos entender si los sistemas de software que construimos eliminarán el potencial de que poblaciones enteras experimenten una prosperidad compartida y brinden igualdad de acceso a la tecnología.

Históricamente, las empresas que se enfrentan a una decisión entre completar un objetivo estratégico que impulsa el dominio del mercado y los ingresos y uno que potencialmente ralentiza el impulso hacia ese objetivo han optado por la velocidad y el valor para los accionistas. Esta tendencia se ve exacerbada por el hecho de que muchas empresas valoran el desempeño y la excelencia individuales, pero a menudo no logran impulsar de manera efectiva la responsabilidad sobre la equidad del producto en todas las áreas. Centrarse en los usuarios subrepresentados es una clara oportunidad para promover la equidad. Para seguir siendo competitivos en el sector de la tecnología, debemos aprender a diseñar para la equidad global.

Hoy en día, nos preocupamos cuando las empresas diseñan tecnología para escanear, capturar e identificar a las personas que caminan por la calle. Nos preocupa la privacidad y cómo los gobiernos podrían usar esta información ahora y en el futuro. Sin embargo, la mayoría de los tecnólogos no tienen la perspectiva necesaria de los grupos subrepresentados para comprender el impacto de la variación racial en el reconocimiento facial o para comprender cómo la aplicación de IA puede generar resultados dañinos e inexactos.

Actualmente, el software de reconocimiento facial impulsado por IA continúa perjudicando a las personas de color o minorías étnicas. Nuestra investigación no es lo suficientemente exhaustiva y no incluye una gama lo suficientemente amplia de diferentes tonos de piel. No podemos esperar que el resultado sea válido si tanto los datos de capacitación como los que crean el software representan solo una pequeña subsección de personas. En esos casos, deberíamos estar dispuestos a retrasar el desarrollo para tratar de obtener datos más completos y precisos, y un producto más completo e inclusivo.

Sin embargo, la ciencia de datos en sí misma es un desafío para que los humanos la evalúen. Incluso cuando tenemos representación, un conjunto de entrenamiento aún puede estar sesgado y producir resultados no válidos. Un estudio completado en 2016 encontró que más de 117 millones de adultos estadounidenses están en una base de datos de reconocimiento facial de las fuerzas del orden.⁵ Debido a la vigilancia policial desproporcionada de las comunidades negras y los resultados dispares en los arrestos, podría haber tasas de error sesgadas racialmente al utilizar dicha base de datos en el reconocimiento facial. Aunque el software se está desarrollando y desplegando a un ritmo cada vez mayor, las pruebas independientes no lo están. Para corregir este paso en falso atroz, debemos tener la integridad para reducir la velocidad y asegurarnos de que nuestras entradas contengan la menor parcialidad posible. Google ahora ofrece capacitación estadística en el contexto de la IA para ayudar a garantizar que los conjuntos de datos no estén intrínsecamente sesgados.

Por lo tanto, cambiar el enfoque de su experiencia en la industria para incluir una educación más integral, multicultural, de estudios de raza y género no solo *essuresponsabilidad*, sino también *la responsabilidad de su empleador*. Las empresas tecnológicas deben asegurarse de que sus empleados estén recibiendo un desarrollo profesional continuo y que este desarrollo sea integral y multidisciplinar. El requisito no es que un individuo se encargue de aprender sobre otras culturas u otros grupos demográficos por sí solo. El cambio requiere que cada uno de nosotros, individualmente o como líderes de equipos, invierta en un desarrollo profesional continuo que construya no solo nuestras habilidades de liderazgo y desarrollo de software, sino también nuestra capacidad para comprender las diversas experiencias a lo largo de la humanidad.

Hacer que la diversidad sea accionable

La equidad y la justicia sistémicas son alcanzables si estamos dispuestos a aceptar que todos somos responsables de la discriminación sistémica que vemos en el sector de la tecnología. Somos responsables de las fallas en el sistema. Aplazar o abstraerse de la responsabilidad personal es ineficaz y, dependiendo de su función, podría ser irresponsable. También es irresponsable atribuir completamente la dinámica en su empresa específica o dentro de su equipo a los problemas sociales más amplios que contribuyen a la inequidad. Una línea favorita entre los defensores y detractores de la diversidad es algo así: "Estamos trabajando arduamente para solucionarlo (inserte el tema de la discriminación sistémica), pero la rendición de cuentas es difícil. ¿Cómo combatimos (inserte cientos de años) de discriminación histórica?" Esta línea de investigación es un desvío hacia una conversación más filosófica o académica y se aleja de los esfuerzos enfocados en mejorar las condiciones laborales o los resultados. Parte del desarrollo de la capacidad multicultural requiere una comprensión más integral de cómo los sistemas de desigualdad en la sociedad impactan el lugar de trabajo, especialmente en el sector de la tecnología.

5 Stephen Gaines y Sara Williams. "La alineación perpetua: la policía no regulada se enfrenta al reconocimiento en Estados Unidos".

Centro de Privacidad y Tecnología en Georgetown Law, 18 de octubre de 2016.

Si usted es un gerente de ingeniería que trabaja para contratar a más personas de grupos subrepresentados, considerar el impacto histórico de la discriminación en el mundo es un ejercicio académico útil. Sin embargo, es fundamental ir más allá de la conversación académica y centrarse en los pasos cuantificables y procesables que puede tomar para impulsar la equidad y la justicia. Por ejemplo, como gerente de contratación de ingenieros de software, usted es responsable de garantizar que sus listas de candidatos estén equilibradas. ¿Hay mujeres u otros grupos subrepresentados en el conjunto de revisiones de candidatos? Después de contratar a alguien, ¿qué oportunidades de crecimiento ha brindado? ¿La distribución de oportunidades es equitativa? Cada líder de tecnología o gerente de ingeniería de software tiene los medios para aumentar la equidad en sus equipos. Es importante que reconozcamos que, aunque existen importantes desafíos sistémicos, todos somos parte del sistema. Es nuestro problema arreglar.

Rechazar enfoques singulares

No podemos perpetuar soluciones que presenten una sola filosofía o metodología para corregir la inequidad en el sector tecnológico. Nuestros problemas son complejos y multifactoriales. Por lo tanto, debemos interrumpir los enfoques singulares para promover la representación en el lugar de trabajo, incluso si son promovidos por personas que admiramos o que tienen poder institucional.

Una narrativa singular apreciada en la industria de la tecnología es que la falta de representación en la fuerza laboral puede abordarse únicamente arreglando las fuentes de contratación. Sí, ese es un paso fundamental, pero ese no es el problema inmediato que debemos solucionar. Necesitamos reconocer la inequidad sistémica en la progresión y la retención mientras nos enfocamos simultáneamente en la contratación más representativa y las disparidades educativas a través de las líneas de raza, género y estatus socioeconómico y de inmigración, por ejemplo.

En la industria de la tecnología, muchas personas de grupos subrepresentados son pasadas por alto diariamente en busca de oportunidades y avances. Deserción entre los empleados negros+ de Google **superó la deserción de todos los demás grupos** confunde el progreso en las metas de representación. Si queremos impulsar el cambio y aumentar la representación, debemos evaluar si estamos creando un ecosistema en el que todos los aspirantes a ingenieros y otros profesionales de la tecnología puedan prosperar.

Comprender completamente un espacio problemático completo es fundamental para determinar cómo solucionarlo. Esto es válido para todo, desde una migración de datos críticos hasta la contratación de una fuerza laboral representativa. Por ejemplo, si usted es un gerente de ingeniería que quiere contratar a más mujeres, no se concentre solo en construir una tubería. Concéntrese en otros aspectos del ecosistema de contratación, retención y progresión y qué tan inclusivo podría ser o no para las mujeres. Considere si sus reclutadores están demostrando la capacidad de identificar candidatos fuertes que sean tanto mujeres como hombres. Si administra un equipo de ingeniería diverso, concéntrese en la seguridad psicológica e invierta en aumentar la capacidad multicultural en el equipo para que los nuevos miembros se sientan bienvenidos.

Una metodología común hoy en día es construir primero para el caso de uso mayoritario, dejando las mejoras y características que abordan los casos extremos para más adelante. Pero este enfoque es defectuoso; les da a los usuarios que ya tienen ventaja en el acceso a la tecnología una ventaja, lo que aumenta la inequidad. Relegar la consideración de todos los grupos de usuarios hasta el punto en que el diseño está casi terminado es bajar el listón de lo que significa ser un excelente ingeniero. En cambio, al incorporar un diseño inclusivo desde el principio y elevar los estándares de desarrollo para que las herramientas sean agradables y accesibles para las personas que luchan por acceder a la tecnología, mejoraremos la experiencia para *todos* los usuarios.

Diseñar para el usuario que menos se parece a usted no solo es sabio, es una buena práctica. Hay próximos pasos pragmáticos e inmediatos que todos los tecnólogos, independientemente del dominio, deben tener en cuenta al desarrollar productos que eviten perjudicar o subrepresentar a los usuarios. Comienza con una investigación más completa de la experiencia del usuario. Esta investigación debe realizarse con grupos de usuarios que sean multilingües y multiculturales y que abarquen múltiples países, clases socioeconómicas, habilidades y rangos de edad. Concéntrese primero en el caso de uso más difícil o menos representado.

Desafiar Procesos Establecidos

Desafiar a sí mismo para construir sistemas más equitativos va más allá de diseñar especificaciones de productos más inclusivas. Construir sistemas equitativos a veces significa desafiar los procesos establecidos que conducen a resultados no válidos.

Considere un caso reciente evaluado por implicaciones de equidad. En Google, varios equipos de ingeniería trabajaron para construir un sistema de solicitud de contratación global. El sistema admite tanto la contratación externa como la movilidad interna. Los ingenieros y gerentes de producto involucrados hicieron un gran trabajo al escuchar las solicitudes de lo que consideraban que era su grupo principal de usuarios: los reclutadores. Los reclutadores se enfocaron en minimizar el tiempo perdido para contratar gerentes y solicitantes, y le presentaron al equipo de desarrollo casos de uso enfocados en la escala y la eficiencia para esas personas. Para impulsar la eficiencia, los reclutadores le pidieron al equipo de ingeniería que incluyera una característica que destacaría las calificaciones de desempeño (específicamente calificaciones más bajas) para el gerente de contratación y el reclutador tan pronto como una transferencia interna expresara interés en un trabajo.

A primera vista, acelerar el proceso de evaluación y ayudar a los solicitantes de empleo a ahorrar tiempo es un gran objetivo. Entonces, ¿dónde está la posible preocupación por la equidad? Se plantearon las siguientes cuestiones de equidad:

- ¿Son las evaluaciones del desarrollo una medida predictiva del desempeño?
- ¿Las evaluaciones de desempeño que se presentan a los gerentes potenciales están libres de prejuicios individuales?
- ¿Están estandarizados los puntajes de evaluación del desempeño en todas las organizaciones?

Si la respuesta a cualquiera de estas preguntas es "no", la presentación de calificaciones de desempeño aún podría generar resultados no equitativos y, por lo tanto, no válidos.

Cuando un ingeniero excepcional cuestionó si el desempeño pasado predecía el desempeño futuro, el equipo de revisión decidió realizar una revisión exhaustiva. Al final, se determinó que los candidatos que habían recibido una calificación de desempeño deficiente probablemente superarían la calificación deficiente si encontraban un nuevo equipo. De hecho, tenían la misma probabilidad de recibir una calificación de desempeño satisfactoria o ejemplar que los candidatos que nunca habían recibido una calificación baja. En resumen, las calificaciones de desempeño son solo indicativas de cómo una persona se está desempeñando en su rol dado. *en el momento en que están siendo evaluados*. Las calificaciones, aunque son una forma importante de medir el desempeño durante un período específico, no predicen el desempeño futuro y no deben usarse para medir la preparación para un puesto futuro o calificar a un candidato interno para un equipo diferente. (Sin embargo, pueden usarse para evaluar si un empleado está colocado de manera adecuada o incorrecta en su equipo actual; por lo tanto, pueden brindar la oportunidad de evaluar cómo apoyar mejor a un candidato interno en el futuro).

Este análisis definitivamente tomó mucho tiempo del proyecto, pero la compensación positiva fue un proceso de movilidad interna más equitativo.

Valores versus resultados

Google tiene un sólido historial de inversión en contratación. Como ilustra el ejemplo anterior, también evaluamos continuamente nuestros procesos para mejorar la equidad y la inclusión. En términos más generales, nuestros valores fundamentales se basan en el respeto y un compromiso inquebrantable con una fuerza laboral diversa e inclusiva. Sin embargo, año tras año, también hemos fallado en la contratación de una fuerza laboral representativa que refleje a nuestros usuarios en todo el mundo. La lucha por mejorar nuestros resultados equitativos persiste a pesar de las políticas y los programas implementados para ayudar a respaldar las iniciativas de inclusión y promover la excelencia en la contratación y el progreso. El punto de falla no está en los valores, intenciones o inversiones de la empresa, sino en la aplicación de esas políticas en el *implementación* nivel.

Los viejos hábitos son difíciles de romper. Los usuarios para los que podría estar acostumbrado a diseñar hoy en día, aquellos de los que está acostumbrado a recibir comentarios, pueden no ser representativos de todos los usuarios a los que necesita llegar. Vemos que esto ocurre con frecuencia en todo tipo de productos, desde dispositivos portátiles que no funcionan para el cuerpo de la mujer hasta software de videoconferencia que no funciona bien para las personas con tonos de piel más oscuros.

Entonces, ¿cuál es la salida?

1. Mírate bien en el espejo. En Google, tenemos el eslogan de la marca, "Crear para todos". ¿Cómo podemos construir para todos cuando no tenemos una fuerza laboral representativa o un modelo de compromiso que centralice primero los comentarios de la comunidad? Nosotros

no puede. La verdad es que, en ocasiones, hemos fallado públicamente en proteger a nuestros usuarios más vulnerables del contenido racista, antisemita y homofóbico.

2.No construyas para todos. Construir *contando* el mundo.Todavía no estamos construyendo para todos. Ese trabajo no sucede en el vacío, y ciertamente no sucede cuando la tecnología aún no es representativa de la población en su conjunto. Dicho esto, no podemos empacar e irnos a casa. Entonces, ¿cómo construimos para todos? Construimos con nuestros usuarios. Necesitamos involucrar a nuestros usuarios en todo el espectro de la humanidad y tener la intención de poner a las comunidades más vulnerables en el centro de nuestro diseño. No deberían ser una ocurrencia tardía.

3.Diseñe para el usuario que tendrá más dificultades para usar su producto.

Construir para aquellos con desafíos adicionales hará que el producto sea mejor para todos. Otra forma de pensar sobre esto es: no intercambie acciones por velocidad a corto plazo.

4.No asuma equidad; mida la equidad en todos sus sistemas.Reconozca que los tomadores de decisiones también están sujetos a prejuicios y pueden no estar bien informados sobre las causas de la inequidad. Es posible que no tenga la experiencia para identificar o medir el alcance de un problema de capital. Atender a una sola base de usuarios podría significar privar de derechos a otra; estas compensaciones pueden ser difíciles de detectar e imposibles de revertir. Asóciese con personas o equipos que sean expertos en diversidad, equidad e inclusión.

5.El cambio es posible.Los problemas que enfrentamos con la tecnología hoy en día, desde la vigilancia hasta la desinformación y el acoso en línea, son realmente abrumadores. No podemos resolverlos con los enfoques fallidos del pasado o solo con las habilidades que ya tenemos. Necesitamos cambiar.

Mantente curioso, avanza

El camino hacia la equidad es largo y complejo. Sin embargo, podemos y debemos hacer la transición de simplemente construir herramientas y servicios a aumentar nuestra comprensión de cómo los productos que diseñamos impactan a la humanidad. Desafiar nuestra educación, influir en nuestros equipos y gerentes, y hacer una investigación más exhaustiva de los usuarios son formas de progresar. Aunque el cambio es incómodo y el camino hacia el alto rendimiento puede ser doloroso, es posible a través de la colaboración y la creatividad.

Por último, como futuros ingenieros excepcionales, debemos centrarnos primero en los usuarios más afectados por el sesgo y la discriminación. Juntos, podemos trabajar para acelerar el progreso centrándonos en la mejora continua y asumiendo nuestros errores. Convertirse en ingeniero es un proceso complicado y continuo. El objetivo es hacer cambios que impulsen a la humanidad hacia adelante sin privar más de sus derechos a los desfavorecidos. Como futuros ingenieros excepcionales, tenemos fe en que podemos prevenir futuras fallas en el sistema.

Conclusión

Desarrollar software y desarrollar una organización de software es un esfuerzo de equipo. A medida que una organización de software escala, debe responder y diseñar adecuadamente para su base de usuarios, que en el mundo interconectado de la informática actual involucra a todos, localmente y en todo el mundo. Se debe hacer un mayor esfuerzo para que tanto los equipos de desarrollo que diseñan el software como los productos que producen reflejen los valores de un conjunto de usuarios tan diverso y abarcador. Y, si una organización de ingeniería quiere escalar, no puede ignorar a los grupos subrepresentados; Los ingenieros de estos grupos no solo aumentan la organización en sí, sino que brindan perspectivas únicas y necesarias para el diseño y la implementación de software que es verdaderamente útil para el mundo en general.

TL; DR

- El sesgo es el predeterminado.
- La diversidad es necesaria para diseñar adecuadamente para una base de usuarios completa.
- La inclusión es fundamental no solo para mejorar la canalización de contratación para los grupos subrepresentados, sino también para proporcionar un entorno de trabajo verdaderamente solidario para todas las personas.
- La velocidad del producto debe evaluarse frente a la provisión de un producto que sea verdaderamente útil para todos los usuarios. Es mejor reducir la velocidad que lanzar un producto que podría causar daño a algunos usuarios.

Cómo liderar un equipo

*Escrito por Brian Fitzpatrick
Editado por Riona MacNamara*

Hemos cubierto mucho terreno hasta ahora sobre la cultura y la composición de los equipos que escriben software y, en este capítulo, echaremos un vistazo a la persona responsable en última instancia de hacer que todo funcione.

Ningún equipo puede funcionar bien sin un líder, especialmente en Google, donde la ingeniería es casi exclusivamente un esfuerzo de equipo. En Google, reconocemos dos roles de liderazgo diferentes. A *Gerente* es un líder de personas, mientras que un *Líder técnico* lidera los esfuerzos tecnológicos. Aunque las responsabilidades de estos dos roles requieren habilidades de planificación similares, requieren habilidades personales bastante diferentes.

Un barco sin capitán no es más que una sala de espera flotante: a menos que alguien agarre el timón y arranque el motor, simplemente va a la deriva sin rumbo con la corriente. Una pieza de software es como ese barco: si nadie lo pilota, te quedas con un grupo de ingenieros que gastan un tiempo valioso, simplemente sentados esperando que suceda algo (o peor aún, siguen escribiendo código que tú no sabes). No es necesario. Aunque este capítulo trata sobre la gestión de personas y el liderazgo técnico, vale la pena leerlo si es un colaborador individual porque probablemente lo ayudará a comprender un poco mejor a sus propios líderes.

Gerentes y líderes tecnológicos (y ambos)

Mientras que cada equipo de ingeniería generalmente tiene un líder, adquieren esos líderes de diferentes maneras. Esto es ciertamente cierto en Google; a veces, un gerente experimentado entra para dirigir un equipo y, a veces, un colaborador individual es ascendido a una posición de liderazgo (generalmente de un equipo más pequeño).

En los equipos nacientes, ambos roles a veces serán ocupados por la misma persona: un *Gerente líder de tecnología* (TLM). En equipos más grandes, un administrador de personas con experiencia intervendrá para asumir el rol de administración, mientras que un ingeniero senior con amplia experiencia asumirá el rol de líder tecnológico. Aunque el gerente y el líder tecnológico juegan un papel importante en el crecimiento y la productividad de un equipo de ingeniería, las habilidades de las personas necesarias para tener éxito en cada rol son muy diferentes.

El Gerente de Ingeniería

Muchas empresas contratan gerentes de personas capacitadas que pueden saber poco o nada sobre ingeniería de software para dirigir sus equipos de ingeniería. Sin embargo, Google decidió desde el principio que sus gerentes de ingeniería de software deberían tener experiencia en ingeniería. Esto significaba contratar gerentes experimentados que solían ser ingenieros de software, o capacitar a ingenieros de software para que fueran gerentes (más sobre esto más adelante).

Al más alto nivel, un gerente de ingeniería es responsable del desempeño, la productividad y la felicidad de cada persona en su equipo, incluido su líder tecnológico, mientras se asegura de que el producto del que es responsable satisfaga las necesidades del negocio. . Debido a que las necesidades del negocio y las necesidades de los miembros individuales del equipo no siempre se alinean, esto a menudo puede colocar a un gerente en una posición difícil.

El líder tecnológico

El líder tecnológico (TL) de un equipo, que a menudo informará al gerente de ese equipo, es responsable (*¡sorpresa!*) de los aspectos técnicos del producto, incluidas las decisiones y opciones tecnológicas, la arquitectura, las prioridades, la velocidad y el proyecto general. gestión (aunque en equipos más grandes pueden tener gerentes de programa que ayuden con esto). El TL generalmente trabajará de la mano con el gerente de ingeniería para garantizar que el equipo cuente con el personal adecuado para su producto y que los ingenieros estén listos para trabajar en las tareas que mejor se adapten a sus conjuntos de habilidades y niveles de habilidad. La mayoría de los TL también son colaboradores individuales, lo que a menudo los obliga a elegir entre hacer algo rápidamente ellos mismos o delegarlo a un miembro del equipo para que lo haga (a veces) más lentamente.

El gerente líder de tecnología

En equipos pequeños e incipientes para los que los gerentes de ingeniería necesitan un sólido conjunto de habilidades técnicas, el valor predeterminado suele ser tener un TLM: una sola persona que pueda manejar tanto las necesidades técnicas como las de personas de su equipo. A veces, un TLM es una persona de mayor rango, pero la mayoría de las veces, el rol lo asume alguien que, hasta hace poco, era un colaborador individual.

En Google, es habitual que los equipos más grandes y bien establecidos tengan un par de líderes, un TL y un gerente de ingeniería, que trabajan juntos como socios. La teoría es que es realmente difícil hacer ambos trabajos al mismo tiempo (bueno) sin agotarse por completo, por lo que es mejor tener dos especialistas que aplasten cada rol con un enfoque dedicado.

El trabajo de TLM es complicado y, a menudo, requiere que TLM aprenda a equilibrar el trabajo individual, la delegación y la gestión de personas. Como tal, por lo general requiere un alto grado de tutoría y asistencia de los TLM más experimentados. (De hecho, recomendamos que además de tomar una serie de clases que Google ofrece sobre este tema, un TLM recién acuñado busque un mentor senior que pueda asesorarlo regularmente a medida que crece en el puesto).

Estudio de caso: influenciar sin autoridad

En general, se acepta que puede hacer que las personas que se reportan a usted hagan el trabajo que necesita para sus productos, pero es diferente cuando necesita que las personas estén fuera de su organización, o diablos, incluso fuera de su área de productos a veces. —hacer algo que crees que debe hacerse. Esta “influencia sin autoridad” es uno de los rasgos de liderazgo más poderosos que puede desarrollar.

Por ejemplo, durante años, Jeff Dean, ingeniero senior y posiblemente el Googler más conocido *en el interior* de Google, dirigió solo una fracción del equipo de ingeniería de Google, pero su influencia en las decisiones técnicas y la dirección llega hasta los extremos de toda la organización de ingeniería y más allá (gracias a que escribe y habla fuera de la empresa).

Otro ejemplo es un equipo que comencé llamado The Data Liberation Front: con un equipo de menos de media docena de ingenieros, logramos que más de 50 productos de Google exportaran sus datos a través de un producto que lanzamos llamado Google Takeout. . En ese momento, no había una directiva formal del nivel ejecutivo en Google para que todos los productos fueran parte de Takeout, entonces, ¿cómo logramos que cientos de ingenieros contribuyeran a este esfuerzo? Identificando una necesidad estratégica para la empresa, mostrando cómo se relacionaba con la misión y las prioridades existentes de la empresa, y trabajando con un pequeño grupo de ingenieros para desarrollar una herramienta que permitiera a los equipos integrarse rápida y fácilmente con Takeout.

Pasar de un rol de colaborador individual a un rol de liderazgo

Ya sea que estén designados oficialmente o no, alguien debe tomar el asiento del conductor si su producto alguna vez va a llegar a alguna parte, y si usted es del tipo motivado e impaciente, esa persona podría ser usted. Podrías encontrarte absorbido por ayudar

tu equipo resuelve conflictos, toma decisiones y coordina personas. Ocurre todo el tiempo, ya menudo por accidente. Tal vez nunca tuvo la intención de convertirse en un "líder", pero de alguna manera sucedió de todos modos. Algunas personas se refieren a esta aflicción como "manageritis".

Incluso si se ha jurado a sí mismo que nunca se convertirá en gerente, en algún momento de su carrera, es probable que se encuentre en una posición de liderazgo, especialmente si ha tenido éxito en su función. El resto de este capítulo pretende ayudarlo a comprender qué hacer cuando esto sucede.

No estamos aquí para intentar convencerlo de que se convierta en gerente, sino para ayudarlo a mostrar por qué los mejores líderes trabajan para servir a su equipo utilizando los principios de humildad, respeto y confianza. Comprender los entresijos del liderazgo es una habilidad vital para influir en la dirección de su trabajo. Si desea dirigir el barco para su proyecto y no solo seguir el viaje, necesita saber cómo navegar, o se ejecutará (y su proyecto) en un banco de arena.

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

Aparte de la sensación general de malestar que siente la mayoría de las personas cuando escuchan la palabra "gerente", hay varias razones por las que la mayoría de las personas no quieren convertirse en gerentes. La principal razón que escuchará en el mundo del desarrollo de software es que dedica mucho menos tiempo a escribir código. Esto es cierto ya sea que se convierta en TL o en gerente de ingeniería, y hablaré más sobre esto más adelante en el capítulo, pero primero, cubramos algunas razones más por las que la mayoría de nosotros evitamos convertirnos en gerentes.

Si ha pasado la mayor parte de su carrera escribiendo código, normalmente termina el día con algo que puede señalar, ya sea un código, un documento de diseño o un montón de errores que acaba de cerrar, y dice: "Eso es lo que hice". Hoy." Pero al final de un ajetreado día de "gestión", por lo general te encontrarás pensando: "Hoy no hice absolutamente nada". Es el equivalente a pasar años contando la cantidad de manzanas que recogiste cada día y cambiar a un trabajo cultivando bananas, solo para decirte a ti mismo al final de cada día: "No recogí ninguna manzana", felizmente ignorando la floreciente banana. árboles sentados a tu lado. Cuantificar el trabajo de administración es más difícil que contar los widgets que produjo, pero hacer posible que su equipo sea feliz y productivo es una gran medida de su trabajo.¹

Otra gran razón para no convertirse en gerente a menudo no se expresa, pero tiene sus raíces en el famoso "Principio de Peter", que establece que "En una jerarquía, cada empleado tiende a elevarse a su nivel de incompetencia". Google generalmente evita esto al exigir que una persona realice el trabajo *encima* su nivel actual por un período de tiempo (es decir, para "superar

¹ Otra diferencia a la que hay que acostumbrarse es que las cosas que hacemos como gerentes suelen dar sus frutos en una línea de tiempo más larga.

expectativas" en su nivel actual) antes de ser promovido a ese nivel. La mayoría de las personas han tenido un gerente que era incapaz de hacer su trabajo o simplemente era muy malo para administrar personas,²y conocemos a algunas personas que han trabajado solo para malos gerentes. Si ha estado expuesto solo a gerentes de mierda durante toda su carrera, ¿por qué *siempre*¿quieres ser gerente? ¿Por qué querrías ser ascendido a un puesto que no te sientes capaz de hacer?

Sin embargo, hay buenas razones para considerar convertirse en TL o gerente. En primer lugar, es una forma de escalar a ti mismo. Incluso si eres bueno escribiendo código, todavía hay un límite superior en la cantidad de código que puedes escribir. ¡Imagínese cuánto código podría escribir un equipo de grandes ingenieros bajo su liderazgo! En segundo lugar, es posible que sea realmente bueno en eso: muchas personas que se ven absorbidas por el vacío de liderazgo de un proyecto descubren que son excepcionalmente hábiles para brindar el tipo de orientación, ayuda y cobertura aérea que necesita un equipo o una empresa. Alguien tiene que liderar, así que ¿por qué no tú?

Liderazgo de servicio

Parece haber una especie de enfermedad que afecta a los gerentes en la que se olvidan de todas las cosas horribles que sus gerentes les hicieron y de repente comienzan a hacer estas mismas cosas para "administrar" a las personas que les reportan. Los síntomas de esta enfermedad incluyen, pero no se limitan a, la microgestión, ignorar a los empleados de bajo rendimiento y contratar a los incautos. Sin un tratamiento oportuno, esta enfermedad puede matar a todo un equipo. El mejor consejo que recibí cuando me convertí en gerente en Google fue de Steve Vinter, un director de ingeniería en ese momento. Él dijo: "Sobre todo, resista la tentación de administrar". Uno de los mayores impulsos del nuevo gerente es "administrar" activamente a sus empleados porque eso es lo que hace un gerente, ¿no? Esto suele tener consecuencias desastrosas.

La cura para la enfermedad de la "gerencia" es una aplicación liberal del "liderazgo de servicio", que es una buena manera de decir que lo más importante que puede hacer como líder es servir a su equipo, al igual que un mayordomo o un mayordomo atiende a los demás. salud y bienestar de un hogar. Como líder servidor, debe esforzarse por crear una atmósfera de humildad, respeto y confianza. Esto podría significar eliminar obstáculos burocráticos que un miembro del equipo no puede eliminar por sí mismo, ayudar a un equipo a lograr un consenso o incluso comprar la cena para el equipo cuando trabajan hasta tarde en la oficina. El líder de servicio llena las grietas para allanar el camino para su equipo y les aconseja cuando es necesario, pero aún así no tiene miedo de ensuciarse las manos. La única gestión que hace un líder servidor es gestionar tanto la salud técnica como social del equipo; tan tentador

2 Otra razón más por la que las empresas no deberían forzar a las personas a la gestión como parte de una trayectoria profesional: si un ingeniero es capaz de escribir montones de código excelente y no tiene ningún deseo de administrar personas o liderar un equipo, al obligarlos a asumir un rol de administración o TL, está perdiendo a un gran ingeniero y ganando a un gerente de mierda. Esto no solo es una mala idea, sino que es activamente dañino.

como podría ser centrarse únicamente en la salud técnica del equipo, la salud social del equipo es igual de importante (pero a menudo infinitamente más difícil de manejar).

El Gerente de Ingeniería

Entonces, ¿qué se espera realmente de un gerente en una empresa de software moderna? Antes de la era de la informática, la "gestión" y el "trabajo" podrían haber asumido roles casi antagónicos, con el gerente ejerciendo todo el poder y el trabajo que requería la acción colectiva para lograr sus propios fines. Pero no es así como funcionan las empresas de software modernas.

Gerente es una palabra de cuatro letras

Antes de hablar sobre las responsabilidades principales de un gerente de ingeniería en Google, repasemos la historia de los gerentes. El concepto actual del gerente de pelo puntiagudo es parcialmente un remanente, primero de la jerarquía militar y luego adoptado por la Revolución Industrial, ¡hace más de cien años! Las fábricas comenzaron a aparecer en todas partes y requerían trabajadores (generalmente no calificados) para mantener las máquinas en funcionamiento. En consecuencia, estos trabajadores requerían supervisores para administrarlos, y debido a que era fácil reemplazar a estos trabajadores con otras personas que estaban desesperadas por un trabajo, los gerentes tenían poca motivación para tratar bien a sus empleados o mejorar sus condiciones. Sea humano o no, este método funcionó bien durante muchos años cuando los empleados no tenían nada más que hacer que realizar tareas rutinarias.

Los gerentes frecuentemente trataban a los empleados de la misma manera que los conductores de carretas tratarían a sus mulas: los motivaban llevándolos alternativamente con una zanahoria y, cuando eso no funcionaba, azotándolos con un palo. Este método de gestión del palo y la zanahoria sobrevivió a la transición de la fábrica³ a la oficina moderna, donde el estereotipo del gerente duro como un arriero floreció a mediados del siglo XX, cuando los empleados trabajaban en el mismo trabajo durante años y años.

Esto continúa hoy en algunas industrias, incluso en industrias que requieren pensamiento creativo y resolución de problemas, a pesar de numerosos estudios que sugieren que el palo y la zanahoria anacrónicos son ineficaces y perjudiciales para la productividad de las personas creativas. Mientras que el trabajador de la línea de ensamblaje de años pasados podía capacitarse en días y reemplazarse a voluntad, los ingenieros de software que trabajan en grandes bases de código pueden tardar meses en ponerse al día en un nuevo equipo. A diferencia del trabajador reemplazable de la línea de montaje, estas personas necesitan cuidados, tiempo y espacio para pensar y crear.

³ Para obtener más información fascinante sobre cómo optimizar los movimientos de los trabajadores de fábrica, lea en *Scientific Management o taylorismo, especialmente sus efectos en la moral de los trabajadores.*

Gerente de ingeniería de hoy

La mayoría de la gente todavía usa el título “gerente” a pesar de que a menudo es un anacronismo. El título mismo a menudo alienta a los nuevos gerentes a gestionar sus informes. Los gerentes pueden terminar actuando como padres,⁴ y en consecuencia los empleados reaccionan como niños. Para enmarcar esto en el contexto de la humildad, el respeto y la confianza: si un gerente deja en claro que confía en su empleado, el empleado siente una presión positiva para estar a la altura de esa confianza. Es así de simple. Un buen gerente abre el camino para un equipo, velando por su seguridad y bienestar, al mismo tiempo que se asegura de que se satisfagan sus necesidades. Si hay algo que recuerdas de este capítulo, que sea esto:

Los gerentes tradicionales se preocupan por cómo hacer las cosas, mientras que los grandes gerentes se preocupan por las cosas que se hacen (y confían en que su equipo descubra cómo hacerlo).

Un nuevo ingeniero, Jerry, se unió a mi equipo hace unos años. El último gerente de Jerry (en una empresa diferente) insistía en que estuviera en su escritorio de 9:00 a 5:00 todos los días, y asumió que si no estaba allí, no estaba trabajando lo suficiente (lo cual es, por supuesto, una suposición ridícula). En su primer día trabajando conmigo, Jerry vino a verme a las 4:40 pm y tartamudeó una disculpa diciendo que tenía que irse 15 minutos antes porque tenía una cita que no había podido reprogramar. Lo miré, sonréí y le dije rotundamente: “Mira, mientras termines tu trabajo, no me importa a qué hora salgas de la oficina”. Jerry me miró sin comprender durante unos segundos, asintió y siguió su camino. Traté a Jerry como un adulto; siempre hacía su trabajo y nunca tuve que preocuparme de que él estuviera en su escritorio, porque no necesitaba una niñera para hacer su trabajo. *eso es tu verdadero problema.*

El fracaso es una opción

Otra forma de catalizar a su equipo es hacer que se sientan seguros y protegidos para que puedan asumir mayores riesgos mediante la creación de seguridad psicológica, lo que significa que los miembros de su equipo sienten que pueden ser ellos mismos sin temor a las repercusiones negativas de usted o de los miembros de su equipo. El riesgo es algo fascinante; la mayoría de los humanos son terribles para evaluar el riesgo, y la mayoría de las empresas intentan evitar el riesgo a toda costa. Como resultado, el modus operandi habitual es trabajar de forma conservadora y centrarse en los éxitos menores, incluso cuando asumir un riesgo mayor podría significar un éxito exponencialmente mayor. Un dicho común en Google es que si intenta alcanzar un objetivo imposible, es muy probable que fracase, pero si fracasa al intentar alcanzar lo imposible, lo más probable es que logre mucho más de lo que habría logrado si lo hubiera hecho. simplemente intentó algo que usted

⁴ Si tiene hijos, es muy probable que pueda recordar con sorprendente claridad la primera vez que dijo algo. algo a su hijo que lo hizo detenerse y exclamar (quizás incluso en voz alta): “Mierda, me he convertido en mi madre”.

sabía que podía completar. Una buena forma de crear una cultura en la que se acepte la asunción de riesgos es hacerle saber a su equipo que está bien fracasar.

Entonces, dejemos eso de lado: está bien fallar. De hecho, nos gusta pensar en el fracaso como una forma de aprender mucho muy rápido (siempre que no estés fallando repetidamente en lo mismo). Además, es importante ver el fracaso como una oportunidad para aprender y no señalar con el dedo o culpar. Fallar rápido es bueno porque no hay mucho en juego. Fallar lentamente también puede enseñar una lección valiosa, pero es más doloroso porque hay más en riesgo y se puede perder más (generalmente tiempo de ingeniería). Fracasar de una manera que afecta a los clientes es probablemente el fracaso menos deseable que encontramos, pero también es uno en el que tenemos la mayor cantidad de estructura para aprender de los fracasos. Como se mencionó anteriormente, cada vez que hay una falla importante en la producción de Google, realizamos una autopsia. Este procedimiento es una forma de documentar los eventos que llevaron a la falla real y desarrollar una serie de pasos que evitarán que suceda en el futuro. Esta no es una oportunidad para señalar con el dedo ni pretende introducir controles burocráticos innecesarios; más bien, el objetivo es centrarse fuertemente en el núcleo del problema y solucionarlo de una vez por todas. Es muy difícil, pero bastante efectivo (y catártico).

Los éxitos y fracasos individuales son un poco diferentes. Una cosa es alabar los éxitos individuales, pero buscar culpar a los individuos en caso de fracaso es una excelente manera de dividir un equipo y desalentar la toma de riesgos en todos los ámbitos. Está bien fallar, pero falla como equipo y aprende de tus fallas. Si un individuo tiene éxito, elógielo frente al equipo. Si un individuo falla, haga una crítica constructiva en privado.⁵ Cualquiera sea el caso, aproveche la oportunidad y aplique una generosa dosis de humildad, respeto y confianza para ayudar a su equipo a aprender de sus fallas.

antipatrones

Antes de repasar una letanía de "patrones de diseño" para TL y gerentes de ingeniería exitosos, vamos a revisar una colección de patrones que usted/noquieres seguir si quieras ser un gerente exitoso. Hemos observado estos patrones destructivos en un puñado de malos gerentes que hemos encontrado en nuestras carreras y, en más de unos pocos casos, en nosotros mismos.

5 La crítica pública de un individuo no solo es ineficaz (pone a la gente en defensa), sino que rara vez es necesaria, y la mayoría de las veces es simplemente malo o cruel. Puede estar seguro de que el resto del equipo ya sabe cuándo un individuo ha fallado, por lo que no es necesario restregárselo.

Antipatrón: contratar empujones

Si es gerente y se siente inseguro en su función (por el motivo que sea), una forma de asegurarse de que nadie cuestione su autoridad o amenace su trabajo es contratar personas a las que pueda presionar. Puedes lograr esto contratando personas que no sean tan inteligentes o ambiciosas como tú, o simplemente personas que sean más inseguras que tú. Aunque esto consolidará su posición como líder del equipo y tomador de decisiones, significará mucho más trabajo para usted. Tu equipo no podrá hacer ningún movimiento sin que los guíes como perros con correa. Si construyes un equipo de pusilánimes, probablemente no puedas tomarte unas vacaciones; en el momento en que sale de la habitación, la productividad se detiene. Pero seguramente este es un pequeño precio a pagar por sentirse seguro en su trabajo, ¿verdad?

En su lugar, debe esforzarse por contratar personas que sean más inteligentes que usted y que puedan reemplazarlo. Esto puede ser difícil porque estas mismas personas lo desafiarán regularmente (además de avisarle cuando cometa un error). Estas mismas personas también lo impresionarán constantemente y harán que sucedan grandes cosas. Podrán dirigirse a sí mismos en una medida mucho mayor, y algunos también estarán ansiosos por liderar el equipo. No deberías ver esto como un intento de usurpar tu poder; en su lugar, míralo como una oportunidad para liderar un equipo adicional, investigar nuevas oportunidades o incluso tomar unas vacaciones sin preocuparte por controlar al equipo todos los días para asegurarte de que está haciendo su trabajo.

Antipatrón: ignorar los de bajo rendimiento

Al principio de mi carrera como gerente en Google, llegó el momento de entregar cartas de bonificación a mi equipo, y sonréi cuando le dije a mi gerente: "¡Me encanta ser gerente!" Sin perder el ritmo, mi gerente, un veterano de la industria desde hace mucho tiempo, respondió: "A veces puedes ser el hada de los dientes, otras veces tienes que ser el dentista".

Nunca es divertido sacar dientes. Hemos visto a los líderes de equipo hacer todo lo correcto para construir equipos increíblemente fuertes solo para que estos equipos no sobresalgan (y eventualmente se desmoronen) debido a solo uno o dos de bajo rendimiento. Entendemos que el aspecto humano es la parte más desafiante de la creación de software, pero la parte más difícil de tratar con humanos es manejar a alguien que no cumple con las expectativas. A veces, las personas pierden las expectativas porque no están trabajando lo suficiente o lo suficientemente duro, pero los casos más difíciles son cuando alguien simplemente no es capaz de hacer su trabajo sin importar cuánto tiempo o duro trabaje.

El equipo de ingeniería de confiabilidad del sitio (SRE) de Google tiene un lema: "La esperanza no es una estrategia". Y en ninguna parte se abusa más de la esperanza como estrategia que en el trato con un trabajador de bajo rendimiento. La mayoría de los líderes de equipo aprietan los dientes, desvían la mirada y simplemente esperanzan que el bajo

el ejecutante mejora mágicamente o simplemente desaparece. Sin embargo, es extremadamente raro que esta persona lo haga.

Mientras que el líder espera y el de bajo rendimiento no mejora (o se va), los de alto rendimiento del equipo pierden un tiempo valioso arrastrando al de bajo rendimiento y la moral del equipo se desvanece. Puede estar seguro de que el equipo sabe que el jugador de bajo rendimiento está allí incluso si lo está ignorando; de hecho, el equipo está *extremadamente conscientes* de quiénes son los de bajo rendimiento, porque tienen que cargar con ellos.

Ignorar a los de bajo rendimiento no solo es una forma de evitar que nuevos empleados de alto rendimiento se unan a su equipo, sino que también es una forma de alentar a los de alto rendimiento existentes a que se vayan. Eventualmente terminas con un equipo completo de bajo rendimiento porque son los únicos que no pueden irse por su propia voluntad. Por último, ni siquiera le está haciendo ningún favor al jugador de bajo rendimiento al mantenerlo en el equipo; a menudo, alguien a quien no le iría bien en su equipo en realidad podría tener mucho impacto en otro lugar.

El beneficio de tratar con un jugador de bajo rendimiento lo más rápido posible es que puedes ponerte en la posición de ayudarlos hacia arriba o hacia afuera. Si trata de inmediato con un trabajador de bajo rendimiento, a menudo encontrará que simplemente necesita un poco de aliento o dirección para pasar a un estado más alto de productividad. Si espera demasiado para tratar con un trabajador de bajo rendimiento, su relación con el equipo será tan amarga y usted se sentirá tan frustrado que no podrá ayudarlo.

¿Cómo entrenar de manera efectiva a un trabajador de bajo rendimiento? La mejor analogía es imaginar que estás ayudando a una persona que cojea a aprender a caminar de nuevo, luego a trotar y luego a correr junto con el resto del equipo. Casi siempre requiere una microgestión temporal, pero aun así mucha humildad, respeto y confianza, particularmente respeto. Establezca un marco de tiempo específico (digamos, dos meses) y algunas metas muy específicas que espera que logren en ese período. Haga que las metas sean pequeñas, incrementales y medibles para que haya una oportunidad de muchos pequeños éxitos. Reúnase con el miembro del equipo todas las semanas para verificar el progreso y asegúrese de establecer expectativas realmente explícitas en torno a cada próximo hito para que sea fácil medir el éxito o el fracaso. Si el de bajo rendimiento no puede seguir el ritmo, será bastante obvio para ambos al principio del proceso. En este punto, la persona a menudo reconocerá que las cosas no van bien y decidirá renunciar; en otros casos, la determinación se activará y "mejorarán su juego" para cumplir con las expectativas. De cualquier manera, al trabajar directamente con el de bajo rendimiento, está catalizando cambios importantes y necesarios.

Antipatrón: ignorar problemas humanos

Un gerente tiene dos áreas principales de enfoque para su equipo: la social y la técnica. Es bastante común que los gerentes sean más fuertes en el aspecto técnico de Google, y debido a que la mayoría de los gerentes son promovidos de un trabajo técnico (cuyo objetivo principal de su trabajo era resolver problemas técnicos), tienden a ignorar los problemas humanos. Es tentador concentrar toda su energía en el aspecto técnico de su equipo.

porque, como colaborador individual, pasa la mayor parte de su tiempo resolviendo problemas técnicos. Cuando era estudiante, sus clases consistían en aprender los entresijos técnicos de su trabajo. Sin embargo, ahora que es gerente, ignora el elemento humano de su equipo bajo su propio riesgo.

Comencemos con un ejemplo de un líder que ignora el elemento humano en su equipo. Hace años, Jake tuvo su primer hijo. Jake y Katie habían trabajado juntos durante años, tanto de forma remota como en la misma oficina, por lo que en las semanas posteriores a la llegada del nuevo bebé, Jake trabajó desde casa. Esto funcionó muy bien para la pareja, y Katie estuvo totalmente de acuerdo porque ya estaba acostumbrada a trabajar de forma remota con Jake. Eran tan productivos como siempre hasta que su gerente, Pablo (que trabajaba en una oficina diferente), descubrió que Jake trabajaba desde casa la mayor parte de la semana. Pablo estaba molesto porque Jake no iba a la oficina a trabajar con Katie, a pesar de que Jake era tan productivo como siempre y que Katie estaba bien con la situación. Jake intentó explicarle a Pablo que era tan productivo como lo sería si fuera a la oficina y que era mucho más fácil para él y su esposa trabajar principalmente desde casa durante algunas semanas. Respuesta de Pablo: "Amigo, la gente tiene hijos todo el tiempo. Tienes que ir a la oficina. No hace falta decir que Jake (normalmente un ingeniero afable) se enfureció y perdió mucho respeto por Pablo.

Hay muchas maneras en que Pablo podría haber manejado esto de manera diferente: podría haber mostrado cierta comprensión de que Jake quería pasar más tiempo en casa con su esposa y, si su productividad y su equipo no se hubieran visto afectados, simplemente dejarlo seguir haciendo así por un tiempo. Podría haber negociado que Jake fuera a la oficina uno o dos días a la semana hasta que las cosas se calmaran. Independientemente del resultado final, un poco de empatía habría sido de gran ayuda para mantener feliz a Jake en esta situación.

Antipatrón: Sea amigo de todos

La primera incursión que la mayoría de las personas tienen en el liderazgo de cualquier tipo es cuando se convierten en gerentes o TL de un equipo del que anteriormente eran miembros. Muchos líderes no quieren perder las amistades que han cultivado con sus equipos, por lo que a veces trabajarán más duro para mantener la amistad con los miembros de su equipo después de convertirse en líderes de equipo. Esta puede ser una receta para el desastre y para muchas amistades rotas. No confunda la amistad con liderar con un toque suave: cuando tiene poder sobre la carrera de alguien, es posible que se sienta presionado para corresponder artificialmente gestos de amistad.

Recuerde que puede liderar un equipo y generar consenso sin ser un amigo cercano de su equipo (o un monumental duro). Del mismo modo, puede ser un líder duro sin tirar por la borda sus amistades existentes. Descubrimos que almorzar con su equipo puede ser una forma efectiva de mantenerse conectado socialmente con ellos sin

hacerlos sentir incómodos: esto le da la oportunidad de tener conversaciones informales fuera del entorno laboral normal.

A veces, puede ser complicado pasar a una función de gestión sobre alguien que ha sido un buen amigo y compañero. Si el amigo que está siendo dirigido no es autogestionario y no trabaja duro, puede ser estresante para todos. Le recomendamos que evite meterse en esta situación siempre que sea posible, pero si no puede, preste especial atención a su relación con esas personas.

Antipatrón: comprometer la barra de contratación

Steve Jobs dijo una vez: "La gente A contrata a otra gente A; La gente B contrata a la gente C". Es increíblemente fácil ser víctima de este adagio, y más aún cuando estás tratando de contratar rápidamente. Un enfoque común que he visto fuera de Google es que un equipo necesita contratar a 5 ingenieros, por lo que revisa una pila de solicitudes, entrevista a 40 o 50 personas y elige a los mejores 5 candidatos independientemente de si cumplen con la barra de contratación.

Esta es una de las formas más rápidas de construir un equipo mediocre.

El costo de encontrar a la persona adecuada, ya sea pagando a los reclutadores, pagando por publicidad o golpeando el pavimento en busca de referencias, palidece en comparación con el costo de tratar con un empleado que nunca debería haber contratado en primer lugar. Este "costo" se manifiesta en la pérdida de productividad del equipo, el estrés del equipo, el tiempo dedicado a administrar al empleado, y el papeleo y el estrés involucrados en el despido del empleado. Eso suponiendo, por supuesto, que trate de evitar el costo monumental de simplemente dejarlos en el equipo. Si está administrando un equipo para el cual no tiene voz en la contratación y no está satisfecho con las contrataciones que se realizan para su equipo, debe luchar con uñas y dientes para obtener ingenieros de mayor calidad. Si todavía tiene ingenieros por debajo del estándar, tal vez sea hora de buscar otro trabajo. Sin las materias primas para un gran equipo,

Antipatrón: trata a tu equipo como niños

La mejor manera de mostrarle a su equipo que no confía en él es tratar a los miembros del equipo como niños: las personas tienden a actuar de la manera en que las trata, así que si las trata como niños o prisioneros, no se sorprenda cuando así sea. Ellos se comportan. Puede manifestar este comportamiento al microgestionarlos o simplemente faltarle el respeto a sus habilidades y no darles la oportunidad de ser responsables de su trabajo. Si es permanentemente necesario microgestionar a las personas porque no confías en ellas, tienes un fracaso de contratación en tus manos. Bueno, es un fracaso a menos que tu objetivo sea formar un equipo al que puedas pasar el resto de tu vida cuidando niños. Si contrata a personas dignas de confianza y les demuestra que confía en ellas, por lo general estarán a la altura de las circunstancias (siguiendo la premisa básica, como mencionamos anteriormente, de que ha contratado a buenas personas).

Los resultados de este nivel de confianza llegan hasta cosas más mundanas como suministros de oficina e informática. Como otro ejemplo, Google proporciona a los empleados gabinetes llenos de diversos suministros de oficina (por ejemplo, bolígrafos, cuadernos y otros implementos de creación "heredados") que pueden tomar libremente cuando los empleados los necesiten. El departamento de TI ejecuta numerosas "paradas tecnológicas" que brindan áreas de autoservicio que son como una mini tienda de electrónica. Estos contienen una gran cantidad de accesorios de computadora y accesorios (fuentes de alimentación, cables, mouse, unidades USB, etc.) que serían fáciles de agarrar y llevarse en masa, pero debido a que se confía en que los empleados de Google verifiquen estos artículos, sentir la responsabilidad de hacer lo correcto. Muchas personas de corporaciones típicas reaccionan con horror al escuchar esto, exclamando que seguramente Google está perdiendo dinero debido a que la gente "roba" estos artículos. Eso es ciertamente posible, pero ¿qué pasa con los costos de tener una fuerza laboral que se comporta como niños o que tiene que perder un tiempo valioso solicitando formalmente suministros de oficina baratos? Seguro que es más caro que el precio de unos cuantos bolígrafos y cables USB.

Patrones positivos

Ahora que hemos cubierto los antipatrones, pasemos a los patrones positivos para el liderazgo y la gestión exitosos que hemos aprendido de nuestras experiencias en Google, al observar a otros líderes exitosos y, sobre todo, de nuestros propios mentores de liderazgo. Estos patrones no son solo los que hemos tenido mucho éxito en implementar, sino los patrones que siempre hemos respetado más en los líderes a los que seguimos.

perder el ego

Hablamos de "perder el ego" hace algunos capítulos cuando examinamos por primera vez la humildad, el respeto y la confianza, pero es especialmente importante cuando eres un líder de equipo. Con frecuencia, este patrón se malinterpreta como que alienta a las personas a ser felpudos y dejar que otros los pisoteen, pero ese no es el caso en absoluto. Por supuesto, hay una delgada línea entre ser humilde y dejar que los demás se aprovechen de ti, pero la humildad no es lo mismo que la falta de confianza. Todavía puedes tener confianza en ti mismo y opiniones sin ser un ególatra. Los grandes egos personales son difíciles de manejar en cualquier equipo, especialmente en el líder del equipo. En su lugar, debe trabajar para cultivar un fuerte ego e identidad de equipo colectivo.

Parte de "perder el ego" es la confianza: necesitas confiar en tu equipo. Eso significa respetar las habilidades y los logros anteriores de los miembros del equipo, incluso si son nuevos en su equipo.

Si no está microgestionando a su equipo, puede estar bastante seguro de que las personas que trabajan en las trincheras conocen los detalles de su trabajo mejor que usted. Esto significa que, si bien usted puede ser quien dirija al equipo hacia el consenso y ayude a establecer la dirección, las personas que están elaborando el producto deciden mejor los aspectos prácticos de cómo lograr sus objetivos. Esto les da no sólo un mayor sentido

de propiedad, sino también un mayor sentido de rendición de cuentas y responsabilidad por el éxito (o el fracaso) de su producto. Si tiene un buen equipo y deja que fije el estándar de calidad y ritmo de su trabajo, logrará más que si usted se para sobre los miembros del equipo con el palo y la zanahoria.

La mayoría de las personas nuevas en un rol de liderazgo sienten una enorme responsabilidad de hacer todo bien, de saberlo todo y de tener todas las respuestas. Te podemos asegurar que no acertarás en todo, ni tendrás todas las respuestas, y si actúas como lo haces, perderás rápidamente el respeto de tu equipo. Mucho de esto se reduce a tener un sentido básico de seguridad en su rol. Piense en cuando era un colaborador individual; se podía oler la inseguridad a una milla de distancia. Trate de apreciar la indagación: cuando alguien cuestione una decisión o declaración que haya hecho, recuerde que esta persona generalmente solo está tratando de comprenderlo mejor. Si fomenta la investigación, es mucho más probable que reciba el tipo de crítica constructiva que lo convertirá en un mejor líder de un mejor equipo. Encontrar personas que te den buenas críticas constructivas es increíblemente difícil, y es aún más difícil recibir este tipo de críticas de personas que "trabajan para ti". Piense en el panorama general de lo que está tratando de lograr como equipo y acepte comentarios y críticas abiertamente; evitar el impulso de ser territorial.

La última parte de perder el ego es simple, pero muchos ingenieros preferirían ser hervidos en aceite antes que hacerlo: discúlpate cuando cometes un error. Y no queremos decir que debas espolvorear "Lo siento" a lo largo de tu conversación como sal en las palomitas de maíz, debes decirlo sinceramente. Absolutamente vas a cometer errores, y lo admitas o no, tu equipo sabrá que has cometido un error. Los miembros de su equipo lo sabrán independientemente de si hablan con usted (y una cosa está garantizada: ellos *voluntad*hablar de ello entre sí). Disculparse no cuesta dinero. La gente tiene un enorme respeto por los líderes que se disculpán cuando cometen un error y, contrariamente a la creencia popular, disculparse no te hace vulnerable. De hecho, por lo general te ganarás el respeto de las personas cuando te disculpes, porque disculparte les dice a las personas que eres sensato, bueno para evaluar situaciones y, volviendo a la humildad, el respeto y la confianza, humilde.

ser un maestro zen

Como ingeniero, es probable que haya desarrollado un excelente sentido del escepticismo y el cinismo, pero esto puede ser una desventaja cuando intenta liderar un equipo. Esto no quiere decir que debas ser ingenuamente optimista en todo momento, pero harías bien en ser menos escéptico verbalmente y al mismo tiempo hacerle saber a tu equipo que estás al tanto de las complejidades y los obstáculos involucrados en tu trabajo. Medir sus reacciones y mantener la calma es más importante a medida que lidera a más personas, porque su equipo (tanto inconsciente como conscientemente) buscará en usted pistas sobre cómo actuar y reaccionar ante lo que sucede a su alrededor.

Una forma sencilla de visualizar este efecto es ver el organigrama de su empresa como una cadena de engranajes, con el colaborador individual como un engranaje diminuto con solo unos pocos dientes en un extremo, y cada gerente sucesivo por encima de ellos como otro engranaje, terminando con el CEO como el engranaje más grande con muchos cientos de dientes. Esto significa que cada vez que el “engranaje del gerente” de ese individuo (con tal vez unas pocas docenas de dientes) hace una sola revolución, el “engranaje del individuo” hace dos o tres revoluciones. ¡Y el CEO puede hacer un pequeño movimiento y enviar al desafortunado empleado, al final de una cadena de seis o siete marchas, girando salvajemente! Cuanto más avance en la cadena, más rápido podrá hacer girar los engranajes debajo de usted, lo intente o no.

Otra forma de pensar en esto es la máxima de que el líder siempre está en el escenario. Esto significa que si está en una posición de liderazgo abierta, siempre está siendo observado: no solo cuando organiza una reunión o da una charla, sino incluso cuando está sentado en su escritorio respondiendo correos electrónicos. Sus compañeros lo están observando en busca de pistas sutiles en su lenguaje corporal, sus reacciones a las conversaciones triviales y sus señales mientras almuerza. ¿Leen confianza o miedo? Como líder, su trabajo es inspirar, pero la inspiración es un trabajo de 24 horas al día, 7 días a la semana. Su actitud visible sobre absolutamente todo, sin importar cuán trivial sea, se nota inconscientemente y se propaga de manera infecciosa a su equipo.

Uno de los primeros gerentes de Google, Bill Coughran, vicepresidente de ingeniería, realmente había dominado la capacidad de mantener la calma en todo momento. No importa lo que estalle, no importa qué cosa loca suceda, no importa cuán grande sea la tormenta de fuego, Bill nunca entraría en pánico. La mayor parte del tiempo colocaba un brazo sobre el pecho, apoyaba la barbillla en la mano y hacía preguntas sobre el problema, por lo general a un ingeniero completamente aterrado. Esto tuvo el efecto de calmarlos y ayudarlos a concentrarse en resolver el problema en lugar de correr como un pollo sin cabeza. Algunos de nosotros solíamos bromear diciendo que si alguien entraba y le decía a Bill que 19 de las oficinas de la compañía habían sido atacadas por extraterrestres, la respuesta de Bill sería: “¿Alguna idea de por qué no llegaron a 20?”

Esto nos lleva a otro truco de gestión Zen: hacer preguntas. Cuando un miembro del equipo te pide un consejo, suele ser bastante emocionante porque finalmente tienes la oportunidad de arreglar algo. Eso es exactamente lo que hizo durante años antes de pasar a una posición de liderazgo, por lo que generalmente salta al modo de solución, pero ese es el último lugar donde debería estar. La persona que pide consejo normalmente no quiere *usted* para resolver su problema, sino para ayudarlo a resolverlo, y la forma más fácil de hacerlo es hacerle preguntas a esta persona. Esto no quiere decir que debas reemplazarte con una Magic 8 Ball, lo que sería enloquecedor e inútil. En su lugar, puede aplicar un poco de humildad, respeto y confianza y tratar de ayudar a la persona a resolver el problema por su cuenta tratando de refinar y explorar el problema. Esto generalmente llevará al empleado a la respuesta,⁶ y será la respuesta de esa persona, lo que nos lleva de regreso a la propiedad y la responsabilidad que analizamos anteriormente en este capítulo. Ya sea que tenga o no la

⁶ Véase también “Depuración de pato de goma.”

respuesta, usar esta técnica casi siempre dejará al empleado con la impresión de que lo hiciste. Difícil, ¿eh? Sócrates estaría orgulloso de ti.

ser un catalizador

En química, un catalizador es algo que acelera una reacción química, pero que no se consume en la reacción. Una de las formas en que funcionan los catalizadores (por ejemplo, las enzimas) es acercar los reactivos: en lugar de rebotar aleatoriamente en una solución, es mucho más probable que los reactivos interactúen favorablemente entre sí cuando el catalizador ayuda a unirlos. Este es un papel que a menudo deberá desempeñar como líder, y hay varias maneras de hacerlo.

Una de las cosas más comunes que hace un líder de equipo es generar consenso. Esto podría significar que diriges el proceso de principio a fin, o simplemente lo empujas suavemente en la dirección correcta para acelerarlo. Trabajar para generar consenso en el equipo es una habilidad de liderazgo que a menudo utilizan los líderes no oficiales porque es una forma de liderar sin ninguna autoridad real. Si tiene la autoridad, puede dirigir y dictar instrucciones, pero eso es menos efectivo en general que generar consenso.⁷ Si su equipo busca moverse rápidamente, a veces concederá voluntariamente autoridad y dirección a uno o más líderes de equipo. Aunque esto pueda parecer una dictadura o una oligarquía, cuando se hace voluntariamente, es una forma de consenso.

Eliminar obstáculos

A veces, su equipo ya tiene un consenso sobre lo que debe hacer, pero se topó con un obstáculo y se atascó. Esto podría ser un obstáculo técnico u organizativo, pero intervenir para ayudar al equipo a moverse nuevamente es una técnica de liderazgo común. Hay algunos obstáculos que, aunque son prácticamente imposibles de superar para los miembros de su equipo, serán fáciles de manejar para usted, y es valioso ayudar a su equipo a comprender que está contento (y es capaz) de ayudar con estos obstáculos.

Una vez, un equipo pasó varias semanas tratando de superar un obstáculo con el departamento legal de Google. Cuando el equipo finalmente llegó al final de su ingenio colectivo y acudió a su gerente con el problema, el gerente lo resolvió en menos de dos horas simplemente porque conocía a la persona adecuada para contactar para discutir el asunto. En otra ocasión, un equipo necesitaba algunos recursos del servidor y simplemente no podía asignarlos.

Afortunadamente, el gerente del equipo estaba en comunicación con otros equipos de la empresa y logró que el equipo obtuviera exactamente lo que necesitaba esa misma tarde. En otra ocasión, uno de los ingenieros estaba teniendo problemas con un bit arcano de código Java. Aunque la directora del equipo no era experta en Java, pudo conectar al ingeniero con otro

7 Intentar lograr un 100 % de consenso también puede ser perjudicial. Debe ser capaz de decidir continuar incluso si no todos están en la misma página o todavía hay cierta incertidumbre.

ingeniero que sabía exactamente cuál era el problema. No necesita saber todas las respuestas para ayudar a eliminar los obstáculos, pero por lo general es útil conocer a las personas que las conocen. En muchos casos, conocer a la persona adecuada es más valioso que saber la respuesta correcta.

Sea un maestro y un mentor

Una de las cosas más difíciles de hacer como TL es ver a un miembro más joven del equipo pasar 3 horas trabajando en algo que sabes que puedes eliminar en 20 minutos. Enseñar a las personas y darles la oportunidad de aprender por su cuenta puede ser increíblemente difícil al principio, pero es un componente vital de un liderazgo eficaz. Esto es especialmente importante para los nuevos empleados que, además de aprender la tecnología y el código base de su equipo, están aprendiendo la cultura de su equipo y el nivel adecuado de responsabilidad que deben asumir. Un buen mentor debe equilibrar las ventajas y desventajas del tiempo de aprendizaje de un aprendiz versus su tiempo contribuyendo a su producto como parte de un esfuerzo efectivo para escalar el equipo a medida que crece.

Al igual que el rol de gerente, la mayoría de las personas no solicitan el rol de mentor; por lo general, se convierten en uno cuando un líder busca a alguien para asesorar a un nuevo miembro del equipo. No se necesita mucha educación formal o preparación para ser un mentor. Principalmente, necesita tres cosas: experiencia con los procesos y sistemas de su equipo, la capacidad de explicar las cosas a otra persona y la capacidad de medir cuánta ayuda necesita su aprendiz. Lo último es probablemente lo más importante: lo que se supone que debe hacer es darle a su aprendiz suficiente información, pero si explica demasiado las cosas o divaga sin cesar, su aprendiz probablemente lo ignorará en lugar de decirle cortésmente que lo entendió.

Establecer objetivos claros

Este es uno de esos patrones que, por obvio que parezca, es ignorado por una enorme cantidad de líderes. Si va a hacer que su equipo se mueva rápidamente en una dirección, debe asegurarse de que cada miembro del equipo entienda y esté de acuerdo en cuál es la dirección. Imagina que tu producto es un gran camión (y no una serie de tubos). Cada miembro del equipo tiene en la mano una cuerda atada a la parte delantera del camión y, mientras trabajan en el producto, tirarán del camión en su propia dirección. Si su intención es llevar el camión (o el producto) hacia el norte lo más rápido posible, no puede hacer que los miembros del equipo tiren en todos los sentidos; quiere que todos tiren del camión hacia el norte. Si va a tener objetivos claros, debe establecer prioridades claras y ayudar a su equipo a decidir cómo debe hacer concesiones cuando llegue el momento.

La forma más fácil de establecer un objetivo claro y hacer que su equipo impulse el producto en la misma dirección es crear una declaración de misión concisa para el equipo. Una vez que haya ayudado al equipo a definir su dirección y objetivos, puede dar un paso atrás y darle más autonomía, revisando periódicamente para asegurarse de que todos sigan en el camino correcto. Esto no

solo libera su tiempo para manejar otras tareas de liderazgo, también aumenta drásticamente la eficiencia de su equipo. Los equipos pueden tener éxito (y lo tienen) sin objetivos claros, pero normalmente desperdician una gran cantidad de energía ya que cada miembro del equipo tira del producto en una dirección ligeramente diferente. Esto te frustra, ralentiza el progreso del equipo y te obliga a usar cada vez más tu propia energía para corregir el rumbo.

Se honesto

Esto no significa que estamos asumiendo que le está mintiendo a su equipo, pero merece una mención porque inevitablemente se encontrará en una posición en la que no puede decirle algo a su equipo o, peor aún, necesita decírselo a todos algo que no quieren oír. Un gerente que conocemos les dice a los nuevos miembros del equipo: "No les mentiré, pero les diré cuando no pueda decírselo algo o si simplemente no lo sé".

Si un miembro del equipo se le acerca por algo que no puede compartir, está bien decirle que sabe la respuesta pero que no tiene la libertad de decir nada. Aún más común es cuando un miembro del equipo te pregunta algo de lo que no sabes la respuesta: puedes decirle a esa persona que no conoces. Esta es otra de esas cosas que parece deslumbrantemente obvia cuando lo lees, pero muchas personas en un rol de gerente sienten que si no saben la respuesta a algo, prueba que son débiles o están fuera de contacto. En realidad, lo único que prueba es que son humanos.

Dar retroalimentación dura es... bueno, difícil. La primera vez que necesita decírselo a uno de sus informes que cometió un error o que no hizo su trabajo tan bien como se esperaba, puede ser increíblemente estresante. La mayoría de los textos de gestión aconsejan que use el "sándwich de cumplido" para suavizar el golpe cuando brinde comentarios duros. Un sándwich de cumplido se parece a esto:

Eres un miembro sólido del equipo y uno de nuestros ingenieros más inteligentes. Dicho esto, su código es complicado y casi imposible de entender para cualquier otra persona del equipo. Pero tienes un gran potencial y una genial colección de camisetas.

Claro, esto suaviza el golpe, pero con este tipo de andar por las ramas, la mayoría de las personas saldrán de esta reunión pensando solo: "¡Genial! ¡Tengo camisetas geniales!" Nosotros *fuertemente* desaconsejamos usar el emparedado de cumplidos, no porque creamos que debe ser innecesariamente cruel o duro, sino porque la mayoría de las personas no escucharán el mensaje crítico, que es lo que algo debe cambiar. Es posible emplear el respeto aquí: sea amable y empático al hacer una crítica constructiva sin recurrir al emparedado de elogios. De hecho, la amabilidad y la empatía son fundamentales si desea que el destinatario escuche las críticas y no se ponga inmediatamente a la defensiva.

Hace años, un colega recogió a un miembro del equipo, Tim, de otro gerente que insistió en que era imposible trabajar con Tim. Dijo que Tim nunca respondió a los comentarios o críticas y, en cambio, siguió haciendo las mismas cosas que le habían dicho que no debía hacer. Nuestro colega asistió a algunas de las reuniones del gerente con Tim para observar la interacción entre el gerente y Tim, y notó que el gerente

Hizo un uso extensivo del sándwich de cumplidos para no herir los sentimientos de Tim. Cuando trajeron a Tim a su equipo, se sentaron con él y le explicaron muy claramente que Tim necesitaba hacer algunos cambios para trabajar de manera más efectiva con el equipo:

Estamos bastante seguros de que no eres consciente de esto, pero la forma en que interactúas con el equipo los está alienando y enojando, y si quieres ser efectivo, necesitas refinar tus habilidades de comunicación, y nosotros, estamos comprometidos a ayudarlo a hacer eso.

No le dieron a Tim ningún cumplido ni trataron el tema con dulzura, pero igual de importante, no fueron malos, simplemente expusieron los hechos tal como los vieron en función del desempeño de Tim con el equipo anterior. He aquí que, en cuestión de semanas (y después de algunas reuniones más de "actualización"), el desempeño de Tim mejoró dramáticamente. Tim solo necesitaba una retroalimentación y una dirección muy claras.

Cuando proporciona comentarios o críticas directas, su entrega es clave para asegurarse de que su mensaje se escuche y no se desvíe. Si pones al destinatario a la defensiva, no pensará en cómo puede cambiar, sino en cómo puede discutir contigo para demostrar que estás equivocado. Nuestro colega Ben dirigió una vez a un ingeniero al que llamaremos Dean. Dean tenía opiniones extremadamente fuertes y discutía con el resto del equipo sobre cualquier cosa. Podría ser algo tan grande como la misión del equipo o tan pequeño como la ubicación de un widget en una página web; Dean discutiría con la misma convicción y vehemencia de cualquier manera, y se negaba a dejar pasar nada. Después de meses de este comportamiento, Ben se reunió con Dean para explicarle que estaba siendo demasiado combativo. Ahora, si Ben hubiera dicho, "Dean, deja de ser tan idiota, Puedes estar bastante seguro de que Dean lo habría ignorado por completo. Ben pensó mucho en cómo podría hacer que Dean entendiera cómo sus acciones estaban afectando negativamente al equipo, y se le ocurrió la siguiente metáfora:

Cada vez que se toma una decisión, es como un tren que atraviesa la ciudad: cuando salta frente al tren para detenerlo, reduce la velocidad del tren y potencialmente molesta al maquinista que conduce el tren. Pasa un nuevo tren cada 15 minutos, y si saltas frente a cada tren, no solo pasas mucho tiempo deteniendo los trenes, sino que eventualmente uno de los maquinistas que conducen el tren se enojará lo suficiente como para correr justo sobre ti. Por lo tanto, aunque está bien saltar frente a algunos trenes, seleccione y elija los que desea detener para asegurarse de detener solo los trenes que realmente importan.

Esta anécdota no solo inyectó un poco de humor a la situación, sino que también facilitó que Ben y Dean discutieran el efecto que la "parada del tren" de Dean estaba teniendo en el equipo, además de la energía que Dean estaba gastando en ello.

Seguimiento de la felicidad

Como líder, una forma en que puede hacer que su equipo sea más productivo (y menos probable que se vaya) a largo plazo es tomarse un tiempo para medir su felicidad. Los mejores líderes con los que hemos trabajado han sido todos psicólogos aficionados, que de vez en cuando vigilan el bienestar de los miembros de su equipo, asegurándose de que obtengan reconocimiento por lo que hacen.

hacer, y tratando de asegurarse de que estén contentos con su trabajo. Un TLM que conocemos hace una hoja de cálculo de todas las tareas sucias e ingratis que deben realizarse y se asegura de que estas tareas se distribuyan uniformemente en todo el equipo. Otro TLM observa las horas de trabajo de su equipo y utiliza el tiempo de compensación y las salidas divertidas del equipo para evitar el agotamiento y el agotamiento. Otro más inicia sesiones individuales con los miembros de su equipo al tratar sus problemas técnicos como una forma de romper el hielo, y luego se toma un tiempo para asegurarse de que cada ingeniero tenga todo lo que necesita para realizar su trabajo. Después de calentar, habla un poco con el ingeniero sobre cómo disfrutan el trabajo y qué esperan con ansias.

Una buena manera simple de hacer un seguimiento de la felicidad de su equipo.⁸ Es preguntarle al miembro del equipo al final de cada reunión uno a uno, "¿Qué necesita?" Esta simple pregunta es una excelente manera de concluir y asegurarse de que cada miembro del equipo tenga lo que necesita para ser productivo y feliz, aunque es posible que deba investigar un poco para obtener detalles. Si pregunta esto cada vez que tiene una reunión individual, descubrirá que eventualmente su equipo lo recordará y, a veces, incluso acudirá a usted con una larga lista de cosas que necesita para mejorar el trabajo de todos.

La pregunta inesperada

Poco después de comenzar en Google, tuve mi primera reunión con el entonces director ejecutivo Eric Schmidt y, al final, Eric me preguntó: "¿Hay algo que necesites?". Había preparado un millón de respuestas defensivas a preguntas o desafíos difíciles, pero no estaba preparado para esto. Así que me senté allí, estupefacto y mirando. ¡Puede estar seguro de que tenía algo listo la próxima vez que me hicieran esa pregunta!

También puede valer la pena como líder prestar atención a la felicidad de su equipo fuera de la oficina. Nuestro colega Mekka comienza su uno a uno pidiendo a sus informes que califiquen su felicidad en una escala del 1 al 10 y, a menudo, sus informes usan esto como una forma de hablar sobre la felicidad en fuera de la oficina. Tenga cuidado al asumir que las personas no tienen vida fuera del trabajo; tener expectativas poco realistas sobre la cantidad de tiempo que las personas pueden dedicar a su trabajo hará que las personas pierdan el respeto por usted o, peor aún, se agoten. No estamos sugiriendo que se entrometa en la vida personal de los miembros de su equipo, pero ser sensible a las situaciones personales por las que están pasando los miembros de su equipo puede darle mucha información sobre por qué podrían ser más o menos productivos. en cualquier momento dado. Dar un poco más de flexibilidad a un miembro del equipo que actualmente está pasando por un momento difícil en casa puede hacer que esté mucho más dispuesto a dedicar más horas cuando su equipo tiene un plazo ajustado para cumplir más adelante.

⁸ Google también realiza una encuesta anual de empleados llamada "Googlegeist" que califica la felicidad de los empleados en muchos dimensiones. Esto proporciona una buena retroalimentación, pero no es lo que llamaríamos "simple".

Una gran parte del seguimiento de la felicidad de los miembros de su equipo es el seguimiento de sus carreras. Si le pregunta a un miembro del equipo cómo ve su carrera en cinco años, la mayoría de las veces obtendrá un encogimiento de hombros y una mirada en blanco. Cuando se pone en el lugar, la mayoría de la gente no dirá mucho sobre esto, pero por lo general hay algunas cosas que a todos les gustaría hacer en los próximos cinco años: ser promovido, aprender algo nuevo, lanzar algo importante y trabajar con inteligencia. gente. Independientemente de si verbalizan esto, la mayoría de la gente está pensando en ello. Si va a ser un líder efectivo, debe pensar en cómo puede ayudar a que todas esas cosas sucedan y hacerle saber a su equipo que está pensando en esto.

El seguimiento de la felicidad se reduce no solo a monitorear carreras, sino también a brindar a los miembros de su equipo oportunidades para mejorar, ser reconocidos por el trabajo que realizan y divertirse un poco en el camino.

Otros consejos y trucos

Los siguientes son otros consejos y trucos diversos que recomendamos en Google cuando se encuentra en una posición de liderazgo:

Delega, pero ensúciate las manos

Al pasar de un rol de colaborador individual a un rol de liderazgo, lograr un equilibrio es una de las cosas más difíciles de hacer. Inicialmente, se inclina a hacer todo el trabajo usted mismo y, después de estar en un rol de liderazgo durante mucho tiempo, es fácil adquirir el hábito de no hacer nada del trabajo usted mismo. Si es nuevo en un rol de liderazgo, probablemente necesite trabajar duro para delegar el trabajo a otros ingenieros de su equipo, incluso si les llevará mucho más tiempo que a usted realizar ese trabajo. Esta no solo es una forma de mantener la cordura, sino que también es la forma en que el resto de su equipo aprenderá. Si has estado liderando equipos por un tiempo o si eliges un nuevo equipo, Una de las maneras más fáciles de ganarse el respeto del equipo y ponerse al día con lo que están haciendo es ensuciarse las manos, generalmente asumiendo una tarea sucia que nadie más quiere hacer. Puedes tener un currículum y una lista de logros de una milla de largo, pero nada le permite a un equipo saber qué tan hábil y dedicado (y humilde) eres como saltar y realmente hacer un trabajo duro.

Busca reemplazarte

A menos que quiera seguir haciendo exactamente el mismo trabajo por el resto de su carrera, busque reemplazarse a sí mismo. Esto comienza, como mencionamos anteriormente, con el proceso de contratación: si desea que un miembro de su equipo lo reemplace, debe contratar personas capaces de reemplazarlo, lo que generalmente resumimos diciendo que necesita "contratar personas". mucho más inteligente que tú. Una vez que tenga miembros del equipo capaces de hacer su trabajo, debe darles oportunidades para asumir más responsabilidades o, ocasionalmente, liderar el equipo. Si haces esto, verás rápidamente quién tiene más aptitudes para

liderar, así como quién quiere liderar el equipo. Recuerde que algunas personas prefieren ser simplemente colaboradores individuales de alto rendimiento, y eso está bien. Siempre nos han sorprendido las empresas que toman a sus mejores ingenieros y, en contra de sus deseos, colocan a estos ingenieros en funciones gerenciales. Esto generalmente resta a un gran ingeniero de su equipo y agrega un gerente mediocre.

Sepa cuándo hacer olas

Tendrás (inevitablemente y con frecuencia) situaciones difíciles en las que cada célula de tu cuerpo te gritará que no hagas nada al respecto. Podría ser el ingeniero de su equipo cuyas habilidades técnicas no están a la altura. Podría ser la persona que salta delante de cada tren. Podría ser el empleado desmotivado que trabaja 30 horas a la semana. "Solo espera un poco y mejorará", te dirás a ti mismo. "Se resolverá solo", racionalizarás. No caiga en esta trampa: estas son las situaciones en las que necesita hacer las olas más grandes y necesita hacerlo ahora. Rara vez estos problemas se resolverán por sí solos, y cuanto más espere para abordarlos, más afectarán negativamente al resto del equipo y más lo mantendrán despierto por la noche pensando en ellos. esperando, solo está retrasando lo inevitable y causando un daño incalculable en el proceso. Así que actúa, y actúa rápido.

Protege a tu equipo del caos

Cuando asume un rol de liderazgo, lo primero que generalmente descubrirá es que fuera de su equipo hay un mundo de caos e incertidumbre (o incluso locura) que nunca vio cuando era un colaborador individual. Cuando me convertí en gerente por primera vez en la década de 1990 (antes de volver a ser un colaborador individual), me sorprendió la gran cantidad de incertidumbre y el caos organizacional que estaba sucediendo en mi empresa. Le pregunté a otro gerente qué había causado esta inestabilidad repentina en la compañía tranquila, y el otro gerente se rió histéricamente de mi ingenuidad: el caos siempre había estado presente, pero mi gerente anterior nos había protegido a mí y al resto de mi equipo.

Proporcione cobertura aérea a su equipo

Si bien es importante que mantenga informado a su equipo sobre lo que sucede "por encima" de ellos en la empresa, es igual de importante que los defienda de muchas de las incertidumbres y demandas frívolas que pueden imponerle desde fuera de su equipo. Comparta tanta información como pueda con su equipo, pero no los distraiga con locuras organizativas que es muy poco probable que alguna vez los afecten.

Hágale saber a su equipo cuando lo están haciendo bien

Muchos nuevos líderes de equipo pueden quedar tan atrapados al lidiar con las deficiencias de los miembros de su equipo que se niegan a proporcionar comentarios positivos con la suficiente frecuencia. Así como le avisa a alguien cuando comete un error, asegúrate de hacérselo saber

cuando lo hagan bien, y asegúrese de avisarles (y al resto del equipo) cuando eliminan a uno del parque.

Por último, aquí hay algo que los mejores líderes saben y usan con frecuencia cuando tienen miembros de equipo aventureros que quieren probar cosas nuevas:

Es fácil decir "sí" a algo que es fácil de deshacer

Si tiene un miembro del equipo que quiere tomarse uno o dos días para intentar usar una nueva herramienta o biblioteca⁹ que podría acelerar su producto (y no tiene un plazo ajustado), es fácil decir: "Claro, pruébelo". Si, por otro lado, quieren hacer algo como lanzar un producto que tendrá que respaldar durante los próximos 10 años, es probable que desee pensar un poco más. Los líderes realmente buenos saben cuándo algo se puede deshacer, pero hay más cosas que se pueden deshacer de lo que piensas (y esto se aplica tanto a las decisiones técnicas como a las no técnicas).

Las personas son como las plantas

Mi esposa es la menor de seis hijos, y su madre se enfrentó a la difícil tarea de descubrir cómo criar a seis hijos muy diferentes, cada uno de los cuales necesitaba cosas diferentes. Le pregunté a mi suegra cómo se las arreglaba (¿ves lo que hice allí?), y me respondió que los niños son como las plantas: algunos son como los cactus y necesitan poca agua pero mucho sol, otros son como las violetas africanas y necesitan difundir la luz y la tierra húmeda, y otros son como los tomates y realmente sobresaldrán si les das un poco de fertilizante. Si tienes seis hijos y les das a cada uno la misma cantidad de agua, luz y fertilizante, todos recibirán el mismo trato, pero hay buenas probabilidades de que ninguno de ellos obtendrá lo que realmente necesitan.

Y así, los miembros de su equipo también son como las plantas: algunos necesitan más luz y otros necesitan más agua (y algunos necesitan más... fertilizante). Es su trabajo como líder determinar quién necesita qué y luego dársele, excepto que en lugar de luz, agua y fertilizante, su equipo necesita diferentes cantidades de motivación y dirección.

Para que todos los miembros de su equipo obtengan lo que necesitan, debe motivar a los que están estancados y proporcionar una dirección más firme a los que están distraídos o no saben qué hacer. Por supuesto, hay quienes están "a la deriva" y necesitan tanto motivación como dirección. Entonces, con esta combinación de motivación y dirección, puede hacer que su equipo sea feliz y productivo. Y no querrás darles demasiado porque si no necesitan motivación o dirección y tratas de dárselas, solo los vas a molestar.

9 Para obtener una mejor comprensión de cuán "deshacer" pueden ser los cambios técnicos, consulte [capítulo 22](#).

Dar instrucciones es bastante sencillo: requiere una comprensión básica de lo que se debe hacer, algunas habilidades organizativas simples y suficiente coordinación para dividirlo en tareas manejables. Con estas herramientas en la mano, puede proporcionar suficiente orientación para un ingeniero que necesita ayuda direccional. La motivación, sin embargo, es un poco más sofisticada y merece alguna explicación.

Motivación intrínseca versus extrínseca

Hay dos tipos de motivación: *extrínseco*, que se origina en fuerzas externas (como la compensación monetaria), y *intrínseco*, que viene de dentro. En su libro *Manejar*,¹⁰ Dan Pink explica que la forma de hacer que las personas sean más felices y productivas no es motivarlas extrínsecamente (p. ej., arrojarles montones de dinero en efectivo); más bien, necesita trabajar para aumentar su motivación intrínseca. Dan afirma que puede aumentar la motivación intrínseca dando a las personas tres cosas: autonomía, dominio y propósito.¹¹

Una persona tiene autonomía cuando tiene la capacidad de actuar por sí misma sin que alguien la microgestione.¹² Con empleados autónomos (y Google se esfuerza por contratar en su mayoría ingenieros autónomos), puede darles la dirección general en la que deben tomar el producto, pero dejar que ellos decidan cómo llegar allí. Esto ayuda con la motivación no solo porque tienen una relación más cercana con el producto (y probablemente saben mejor que usted cómo construirlo), sino también porque les da un sentido mucho mayor de propiedad del producto. Cuanto mayor sea su apuesta en el éxito del producto, mayor será su interés en verlo triunfar.

El dominio en su forma más básica simplemente significa que necesitas darle a alguien la oportunidad de mejorar las habilidades existentes y aprender otras nuevas. Brindar amplias oportunidades para el dominio no solo ayuda a motivar a las personas, sino que también las hace mejores con el tiempo, lo que hace que los equipos sean más fuertes.¹³ Las habilidades de un empleado son como la hoja de un cuchillo: puede gastar decenas de miles de dólares para encontrar personas con las mejores habilidades para su equipo, pero si usa ese cuchillo durante años sin afilarlo, terminará con un cuchillo desafilado. Eso es ineficiente, y en algunos casos inútil. Google brinda amplias oportunidades para que los ingenieros aprendan cosas nuevas y dominen su oficio a fin de mantenerlos alertas, eficientes y efectivos.

10 Ver [La fantástica charla TED de Danen](#) esta asignatura.

11 Esto supone que a las personas en cuestión se les paga lo suficientemente bien como para que los ingresos no sean una fuente de estrés. 12 Esto supone que tiene personas en su equipo que no necesitan microgestión.

13 Por supuesto, también significa que son empleados más valiosos y comercializables, por lo que es más fácil para ellos recoger y dejarte si no están disfrutando de su trabajo. Ver el patrón en "Hacer un seguimiento de la felicidad" en la página 99.

Por supuesto, toda la autonomía y el dominio del mundo no ayudarán a motivar a alguien si está trabajando sin ningún motivo, por lo que es necesario darle un propósito a su trabajo. Mucha gente trabaja en productos que tienen una gran importancia, pero se mantienen alejados de los efectos positivos que sus productos puedan tener en su empresa, sus clientes o incluso en el mundo. Incluso para los casos en los que el producto pueda tener un impacto mucho menor, puedes motivar a tu equipo buscando la razón de sus esfuerzos y haciéndoles entender esta razón. Si puede ayudarlos a ver este propósito en su trabajo, verá un gran aumento en su motivación y productividad.¹⁴ Un gerente que conocemos sigue de cerca los comentarios por correo electrónico que recibe Google sobre su producto (uno de los productos de "menor impacto"), y cada vez que ve un mensaje de un cliente que habla sobre cómo el producto de la empresa lo ha ayudado personalmente o ayudó a su negocio, inmediatamente lo reenvía al equipo de ingeniería. Esto no solo motiva al equipo, sino que también inspira con frecuencia a los miembros del equipo a pensar en formas en las que pueden mejorar aún más su producto.

Conclusión

Líderes de equipo son una tarea diferente a la de ser un ingeniero de software. Como resultado, los buenos ingenieros de software no siempre son buenos gerentes, y eso está bien. Las organizaciones efectivas permiten trayectorias profesionales productivas tanto para los colaboradores individuales como para los gerentes de personas. Aunque Google descubrió que la experiencia en ingeniería de software en sí misma es invaluable para los gerentes, las habilidades más importantes que un gerente eficaz aporta son las sociales. Los buenos gerentes capacitan a sus equipos de ingeniería ayudándolos a trabajar bien, manteniéndolos enfocados en los objetivos adecuados y aislándolos de problemas fuera del grupo, todo mientras siguen los tres pilares de la humildad, la confianza y el respeto.

TL; DR

- No “administre” en el sentido tradicional; concéntrese en el liderazgo, la influencia y el servicio a su equipo.
- Delegar cuando sea posible; no hagas bricolaje (hazlo tú mismo).
- Preste especial atención al enfoque, la dirección y la velocidad de su equipo.

¹⁴ Adam M. Grant, "La importancia de la importancia de la tarea: efectos en el desempeño laboral, mecanismos relacionales y Condiciones de borde," *Revista de Psicología Aplicada*, 93, núm. 1 (2018), http://bit.ly/task_significance.

Liderando a Escala

*Escrito por Ben Collins-Sussman
Editado por Riona MacNamara*

En Capítulo 5, hablamos sobre lo que significa pasar de ser un "colaborador individual" a ser un líder explícito de un equipo. Es una progresión natural pasar de liderar un equipo a liderar un conjunto de equipos relacionados, y este capítulo habla sobre cómo ser efectivo a medida que continúa por el camino del liderazgo en ingeniería.

A medida que su rol evoluciona, se siguen aplicando todas las mejores prácticas. Sigues siendo un "líder servidor"; solo estás sirviendo a un grupo más grande. Dicho esto, el alcance de los problemas que está resolviendo se vuelve más grande y más abstracto. Gradualmente te ves obligado a convertirte en un "nivel superior". Es decir, eres cada vez menos capaz de entrar en los detalles técnicos o de ingeniería de las cosas, y te empujan a ser "amplio" en lugar de "profundo". En cada paso, este proceso es frustrante: lamenta la pérdida de estos detalles y se da cuenta de que su experiencia previa en ingeniería es cada vez menos relevante para su trabajo. En cambio, su efectividad depende más que nunca de su *genera*l intuición técnica y capacidad para impulsar a los ingenieros a moverse en buenas direcciones.

El proceso suele ser desmoralizador, hasta que un día te das cuenta de que en realidad estás teniendo mucho más impacto como líder que el que nunca tuviste como colaborador individual. Es una realización satisfactoria pero agridulce.

Entonces, suponiendo que entendemos los conceptos básicos del liderazgo, ¿qué se necesita para convertirse en un *realmente bueno* líder? De eso es de lo que hablamos aquí, usando lo que llamamos "los tres Siempre del liderazgo": Siempre Decidir, Siempre Salir, Siempre Escalar.

Estar siempre decidiendo

Dirigir un equipo de equipos significa tomar cada vez más decisiones a niveles cada vez más altos. Su trabajo se centra más en la estrategia de alto nivel que en cómo resolver cualquier tarea de ingeniería específica. En este nivel, la mayoría de las decisiones que tomará tienen que ver con encontrar el conjunto correcto de compensaciones.

La parábola del avión

lindsay joneses un amigo nuestro que es diseñador de sonido teatral y compositor profesional. Se pasa la vida volando por los Estados Unidos, saltando de una producción a otra, y está lleno de historias locas (y verdaderas) sobre viajes aéreos. Esta es una de nuestras historias favoritas:

Son las 6 am, todos estamos abordados en el avión y listos para partir. El capitán entra en el sistema de megafonía y nos explica que, de alguna manera, alguien ha llenado el tanque de combustible en 10,000 galones. Ahora, he volado en aviones durante mucho tiempo y no sabía que tal cosa era posible. Quiero decir, si llené mi auto en exceso por un galón, tendré gasolina en mis zapatos, ¿verdad?

Bueno, de todos modos, el capitán dice que tenemos dos opciones: podemos esperar a que el camión venga a sacar el combustible del avión, lo que va a demorar más de una hora, o veinte personas tienen que bajarse del avión. avión ahora mismo para equilibrar el peso.

Nadie se mueve.

Ahora, hay un tipo al otro lado del pasillo en primera clase, y está absolutamente furioso. Me recuerda a Frank Burns en *MEZCLA* simplemente está súper indignado y farfullando por todas partes, exigiendo saber quién es el responsable. Es un escaparate increíble, es como si fuera Margaret Dumont en las películas de los hermanos Marx.

¡Entonces, agarra su billetera y saca este enorme fajo de efectivo! Y él dice: "¡No puedo llegar tarde a esta reunión!! ¡Le daré \$40 a cualquier persona que baje de este avión ahora mismo!".

Efectivamente, la gente lo acepta. Da \$40 a 20 personas (¡que por cierto son \$800 en efectivo!) y todos se van.

Entonces, ahora estamos listos y nos dirigimos a la pista, y el capitán regresa a la megafonía nuevamente. La computadora del avión ha dejado de funcionar. Nadie sabe por qué. Ahora tenemos que ser remolcados de regreso a la puerta.

Frank Burns está apopléjico. Quiero decir, en serio, pensé que iba a tener un derrame cerebral. Está maldiciendo y gritando. Todos los demás solo se miran entre sí.

Volvemos a la puerta de embarque y este tipo exige otro vuelo. Le ofrecen reservarlo a las 9:30, que es demasiado tarde. Es como, "¿No hay otro vuelo antes de las 9:30?"

El agente de la puerta dice: "Bueno, había otro vuelo a las 8, pero ahora está todo lleno. Ahora están cerrando las puertas".

Y él dice: "¿Lleno? ¿Qué significa que está lleno? ¡¿No hay un solo asiento libre en ese avión?!?!"

El agente de la puerta dice: "No señor, ese avión estaba completamente abierto hasta que 20 pasajeros aparecieron de la nada y ocuparon todos los asientos. Eran los pasajeros más felices que he visto en mi vida, se reían todo el camino por el puente del jet".

Fue un viaje muy tranquilo en el vuelo de las 9:30.

Esta historia trata, por supuesto, de compensaciones. Aunque la mayor parte de este libro se enfoca en varias compensaciones técnicas en los sistemas de ingeniería, resulta que las compensaciones también se aplican a los comportamientos humanos. Como líder, debe tomar decisiones sobre lo que deben hacer sus equipos cada semana. A veces, las compensaciones son obvias ("si trabajamos en este proyecto, se retrasa el otro..."); a veces las compensaciones tienen consecuencias imprevisibles que pueden volver a morderte, como en la historia anterior.

En el nivel más alto, su trabajo como líder, ya sea de un solo equipo o de una organización más grande, es guiar a las personas hacia la solución de problemas difíciles y ambiguos. Por *ambiguo*, queremos decir que el problema no tiene una solución obvia e incluso podría ser irresoluble. De cualquier manera, el problema debe ser explorado, navegado y (con suerte) llevado a un estado en el que esté bajo control. Si escribir código es similar a talar árboles, su trabajo como líder es "ver el bosque a través de los árboles" y encontrar un camino viable a través de ese bosque, dirigiendo a los ingenieros hacia los árboles importantes. Hay tres pasos principales en este proceso. En primer lugar, debe identificar las *anteojeras*; A continuación, debe identificar las *compensaciones*; y luego necesitará iterar en una solución.

Identificar las anteojeras

Cuando aborda un problema por primera vez, a menudo descubrirá que un grupo de personas ya ha estado luchando con él durante años. Estas personas han estado inmersas en el problema durante tanto tiempo que usan "anteojeras", es decir, ya no pueden ver el bosque. Hacen un montón de suposiciones sobre el problema (o la solución) sin darse cuenta. "Así es como siempre lo hemos hecho", dirán, habiendo perdido la capacidad de considerar críticamente el *statu quo*. A veces, descubrirá extraños mecanismos de afrontamiento o racionalizaciones que han evolucionado para justificar el *statu quo*. aquí es donde tu —con ojos nuevos— tienen una gran ventaja. Puede ver estas anteojeras, hacer preguntas y luego considerar nuevas estrategias. (Por supuesto, no estar familiarizado con el problema no es un requisito para un buen liderazgo, pero a menudo es una ventaja).

Identificar las compensaciones clave

Por definición, los problemas importantes y ambiguos no tienen soluciones mágicas de "bala de plata". No hay una respuesta que funcione para siempre en todas las situaciones. solo hay el *la mejor respuesta por el momento*, y es casi seguro que implica hacer concesiones en una u otra dirección. Es su trabajo mencionar las compensaciones, explicárselas a todos y luego ayudar a decidir cómo equilibrarlas.

Decidir, luego iterar

Una vez que comprenda las ventajas y desventajas y cómo funcionan, estará empoderado. Puede utilizar esta información para tomar la mejor decisión para este mes en particular. El próximo mes, es posible que deba reevaluar y reequilibrar las compensaciones nuevamente; es un proceso iterativo. Esto es lo que queremos decir cuando decimos *Estar siempre decidiendo*.

Hay un riesgo aquí. Si no enmarca su proceso como un reequilibrio continuo de las compensaciones, es probable que sus equipos caigan en la trampa de buscar la solución perfecta, lo que puede conducir a lo que algunos llaman "parálisis de análisis". Debe hacer que sus equipos se sientan cómodos con la iteración. Una forma de hacerlo es bajar las apuestas y calmar los nervios explicando: "Vamos a probar esta decisión y ver cómo va. El próximo mes, podemos deshacer el cambio o tomar una decisión diferente". Esto mantiene a la gente flexible y en un estado de aprendizaje de sus elecciones.

Estudio de caso: abordar la "latencia" de la búsqueda web

Al administrar un equipo de equipos, existe una tendencia natural a alejarse de un solo producto y, en cambio, poseer una "clase" completa de productos, o quizás un problema más amplio que cruza productos. Un buen ejemplo de esto en Google tiene que ver con nuestro producto más antiguo, la Búsqueda web.

Durante años, miles de ingenieros de Google han trabajado en el problema general de mejorar los resultados de búsqueda, mejorando la "calidad" de la página de resultados. Pero resulta que esta búsqueda de calidad tiene un efecto secundario: hace que el producto sea cada vez más lento. Érase una vez, los resultados de búsqueda de Google no eran mucho más que una página de 10 enlaces azules, cada uno de los cuales representaba un sitio web relevante. Sin embargo, durante la última década, miles de pequeños cambios para mejorar la "calidad" han dado como resultado resultados cada vez más ricos: imágenes, videos, cuadros con datos de Wikipedia, incluso elementos interactivos de la interfaz de usuario. Esto significa que los servidores necesitan hacer mucho más trabajo para generar información: se envían más bytes por cable; se le pide al cliente (generalmente un teléfono) que presente HTML y datos cada vez más complejos. La latencia ha aumentado. Esto puede no parecer un gran problema, pero la latencia de un producto tiene un efecto directo (en conjunto) en la participación de los usuarios y la frecuencia con la que lo usan. Incluso los aumentos en el tiempo de renderizado tan pequeños como 10 ms importan. La latencia aumenta lentamente. Esto no es culpa de un equipo de ingeniería específico, sino que representa un largo envenenamiento colectivo de los bienes comunes. En algún momento, la latencia general de la búsqueda web crece hasta que su efecto comienza a anular las mejoras en la participación del usuario que provienen de las mejoras en la "calidad" de los resultados.

Varios líderes lucharon con este problema a lo largo de los años, pero no lograron abordar el problema de manera sistemática. Las anteojeras que todos usaban asumían que la única forma de

tratar con la latencia fue declarar una latencia "código amarillo": cada dos o tres años, durante los cuales todo el mundo dejaba todo para optimizar el código y acelerar el producto. Aunque esta estrategia funcionaría temporalmente, la latencia comenzaría a aumentar nuevamente solo uno o dos meses después y pronto volvería a sus niveles anteriores.

Entonces, ¿qué cambió? En algún momento, dimos un paso atrás, identificamos las anteojeras e hicimos una reevaluación completa de las compensaciones. Resulta que la búsqueda de la "calidad" no tiene uno, sino *dos* diferentes costos. El primer costo es para el usuario: más calidad generalmente significa que se envían más datos, lo que significa más latencia. El segundo costo es para Google: más calidad significa hacer más trabajo para generar los datos, lo que cuesta más tiempo de CPU en nuestros servidores, lo que llamamos "capacidad de servicio". Aunque el liderazgo a menudo había tratado con cuidado el equilibrio entre calidad y capacidad, nunca había tratado la latencia como un ciudadano completo en el cálculo. Como dice el viejo chiste: "Bueno, rápido, barato: elige dos". Una forma sencilla de representar las ventajas y desventajas es dibujar un triángulo de tensión entre Bueno (Calidad), Rápido (Latencia) y Barato (Capacidad), como se ilustra en [Figura 6-1](#).

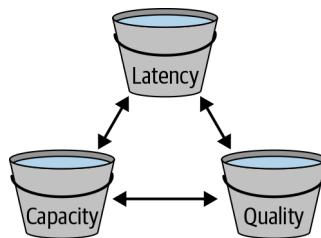


Figura 6-1. Compensaciones dentro de la búsqueda web; ¡elige dos!

Eso es exactamente lo que estaba pasando aquí. Es fácil mejorar cualquiera de estos rasgos dañando deliberadamente al menos uno de los otros dos. Por ejemplo, puede mejorar la calidad colocando más datos en la página de resultados de búsqueda, pero hacerlo afectará la capacidad y la latencia. También puede hacer un intercambio directo entre latencia y capacidad cambiando la carga de tráfico en su clúster de servicio. Si envía más consultas al clúster, obtiene una mayor capacidad en el sentido de que obtiene una mejor utilización de las CPU: más rendimiento por su inversión en hardware. Pero una mayor carga aumenta la contención de recursos dentro de una computadora, lo que empeora la latencia promedio de una consulta. Si disminuye deliberadamente el tráfico de un clúster (ejecutarlo "más fresco"), tiene menos capacidad de servicio en general, pero cada consulta se vuelve más rápida.

El punto principal aquí es que esta idea, una mejor comprensión de *todas* las compensaciones nos permitieron comenzar a experimentar con nuevas formas de equilibrio. En lugar de tratar la latencia como un efecto secundario inevitable y accidental, ahora podríamos tratarla como un primer paso.

1 "Código amarillo" es el término de Google para "hackathon de emergencia para solucionar un problema crítico". Los equipos afectados son esperados a suspender todos los trabajos y concentrar el 100% de atención en el problema hasta que se declare el estado de emergencia.

objetivo de la clase junto con nuestros otros objetivos. Esto nos llevó a nuevas estrategias. Por ejemplo, nuestros científicos de datos pudieron medir exactamente cuánto afectaba la latencia a la participación del usuario. Esto les permitió construir una métrica que comparaba las mejoras impulsadas por la calidad en la participación del usuario a corto plazo con el daño impulsado por la latencia en la participación del usuario a largo plazo. Este enfoque nos permite tomar más decisiones basadas en datos sobre cambios de productos. Por ejemplo, si un pequeño cambio mejora la calidad pero también perjudica la latencia, podemos decidir cuantitativamente si vale la pena lanzar el cambio o no. Somos *siempre decidiéndonos* nuestros cambios de calidad, latencia y capacidad están en equilibrio, e iterando nuestras decisiones cada mes.

Siempre se va

A su valor nominal, *Siempre se vas* es como un pésimo consejo. ¿Por qué un buen líder estaría tratando de irse? De hecho, esta es una cita famosa de Bharat Mediratta, exdirector de ingeniería de Google. Lo que quiso decir es que no es solo su trabajo resolver un problema ambiguo, sino lograr que su organización lo resuelva *por sí mismo*, sin ti presente. Si puede hacer eso, lo libera para pasar a un nuevo problema (o una nueva organización), dejando un rastro de éxito autosuficiente a su paso.

El antipatrón aquí, por supuesto, es una situación en la que te has configurado para ser un punto único de falla (SPOF). Como señalamos anteriormente en este libro, los Googlers tienen un término para eso, el factor autobús: *la cantidad de personas que deben ser atropelladas por un autobús antes de que su proyecto esté completamente condenado*.

Por supuesto, el "autobús" aquí es solo una metáfora. La gente se enferma; cambian de equipo o de empresa; ellos se alejan. Como prueba de fuego, piense en un problema difícil en el que su equipo esté progresando bien. Ahora imagina que tú, el líder, desapareces. ¿Tu equipo sigue adelante? ¿Sigue teniendo éxito? Aquí hay una prueba aún más simple: piensa en las últimas vacaciones que tomaste que duraron al menos una semana. ¿Seguiste revisando el correo electrónico de tu trabajo? (La mayoría de los líderes lo hacen). Pregúntese *por qué*. ¿Se desmoronarán las cosas si no prestas atención? Si es así, es muy probable que te hayas convertido en un SPOF. Tienes que arreglar eso.

Su misión: construir un equipo de "autoconducción"

Volviendo a la cita de Bharat: ser un líder exitoso significa construir una organización que sea capaz de resolver el problema difícil por sí misma. Esa organización necesita tener un conjunto sólido de líderes, procesos de ingeniería saludables y una cultura positiva que se perpetúe a sí misma y que perdure en el tiempo. Sí, esto es difícil; pero vuelve al hecho de que liderar un equipo de equipos a menudo se trata más de organizar gente en lugar de ser un mago técnico. Nuevamente, hay tres partes principales para construir este tipo de grupo autosuficiente: dividir el espacio del problema, delegar subproblemas e iterar según sea necesario.

Dividiendo el espacio del problema

Los problemas desafiantes generalmente se componen de subproblemas difíciles. Si está liderando un equipo de equipos, una opción obvia es poner un equipo a cargo de cada subproblema. El riesgo, sin embargo, es que los subproblemas pueden cambiar con el tiempo, y los límites rígidos del equipo no podrán notar ni adaptarse a este hecho. Si puede, considere una estructura organizativa que sea más flexible, una en la que los subequipos puedan cambiar de tamaño, las personas puedan migrar entre subequipos y los problemas asignados a los subequipos puedan cambiar con el tiempo. Esto implica caminar por una delgada línea entre "demasiado rígido" y "demasiado vago". Por un lado, desea que sus subequipos tengan un sentido claro del problema, el propósito y el logro constante; por otro lado, las personas necesitan la libertad de cambiar de dirección y probar cosas nuevas en respuesta a un entorno cambiante.

Ejemplo: subdividir el "problema de latencia" de la Búsqueda de Google

Al abordar el problema de la latencia de búsqueda, nos dimos cuenta de que el problema podría, como mínimo, subdividirse en dos espacios generales: trabajo que abordó la *síntomas* de latencia, y diferentes trabajos que abordaron la *causas* de latencia. Era obvio que necesitábamos personal para muchos proyectos para optimizar nuestra base de código para la velocidad, pero centrarnos en la velocidad no sería suficiente. Todavía había miles de ingenieros aumentando la complejidad y la "calidad" de los resultados de búsqueda, deshaciendo las mejoras de velocidad tan pronto como aterrizaron, por lo que también necesitábamos personas que se centraran en un problema paralelo de prevención de la latencia en primer lugar. . Descubrimos brechas en nuestras métricas, en nuestras herramientas de análisis de latencia y en nuestra educación y documentación para desarrolladores. Al asignar diferentes equipos para trabajar en las causas y los síntomas de la latencia al mismo tiempo, pudimos controlar sistemáticamente la latencia a largo plazo. (Además, observe cómo estos equipos poseían el *problems*, ¡no soluciones específicas!)

Delegar subproblemas a los líderes

Es esencialmente un cliché que los libros de administración hablen de "delegación", pero hay una razón para eso: la delegación es *realmente difícil* aprender. Va en contra de todos nuestros instintos de eficiencia y logro. Esta dificultad es la razón del dicho: "Si quieres que algo se haga bien, hazlo tú mismo".

Dicho esto, si está de acuerdo en que su misión es construir una organización autónoma, el principal mecanismo de enseñanza es a través de la delegación. Debe construir un conjunto de líderes autosuficientes, y la delegación es absolutamente la forma más efectiva de entrenarlos. Les das una tarea, los dejas fallar y luego intentas de nuevo y vuelves a intentar. Silicon Valley tiene mantras bien conocidos sobre "fallar rápido e iterar". Esta filosofía no solo se aplica al diseño de ingeniería, sino también al aprendizaje humano.

Como líder, su plato se llena constantemente con tareas importantes que deben realizarse. La mayoría de estas tareas son cosas que son bastante fáciles de hacer. Suponga que está trabajando diligentemente en su bandeja de entrada, respondiendo a los problemas y luego

decide reservar 20 minutos para solucionar un problema persistente y de larga data. Pero antes de llevar a cabo la tarea, sea consciente y deténgase. Hágase esta pregunta crítica: *¿Soy realmente el único que puede hacer este trabajo?*

Claro, podría ser la mayoría eficiente para que lo haga, pero luego está fallando en entrenar a sus líderes. No está construyendo una organización autosuficiente. A menos que la tarea sea realmente sensible al tiempo y en llamas, muerde la bala y asigna el trabajo a otra persona, presumiblemente alguien que sepas que puede hacerlo, pero que probablemente tardará mucho más en terminar. Entrénelos en el trabajo si es necesario. Necesita crear oportunidades para que sus líderes crezcan; necesitan aprender a "subir de nivel" y hacer este trabajo ellos mismos para que ya no esté en la ruta crítica.

El corolario aquí es que debe tener en cuenta su propio propósito como líder de líderes. Si se encuentra metido en la maleza, no le está haciendo ningún favor a su organización. Cuando llegue al trabajo todos los días, hágase una pregunta crítica diferente: *que puedo hacer esonadiemás en mi equipo puede hacer?*

Hay una serie de buenas respuestas. Por ejemplo, puede proteger a sus equipos de la política organizacional; puedes darles ánimo; puede asegurarse de que todos se traten bien unos a otros, creando una cultura de humildad, confianza y respeto. También es importante "administrar", asegurándose de que su cadena de gestión comprenda lo que está haciendo su grupo y se mantenga conectado con la empresa en general. Pero a menudo la respuesta más común e importante a esta pregunta es: "Puedo ver el bosque a través de los árboles". En otras palabras, puedes *definir una estrategia de alto nivel*. Su estrategia debe cubrir no solo la dirección técnica general, sino también una estrategia organizacional. Está creando un modelo de cómo se resuelve el problema ambiguo y cómo su organización puede manejar el problema a lo largo del tiempo. Estás mapeando continuamente el bosque y luego asignando la tala de árboles a otros.

Ajuste e iteración

Supongamos que ahora ha llegado al punto en el que ha construido una máquina autosuficiente. Ya no eres un SPOF. ¡Felicidades! ¿Qué haces ahora?

Antes de responder, tenga en cuenta que en realidad se ha liberado a sí mismo: ahora tiene la libertad de "siempre se va". Esto podría ser la libertad de abordar un nuevo problema adyacente, o tal vez incluso podría trasladarse a un departamento y espacio de problemas completamente nuevos, dejando espacio para las carreras de los líderes que ha capacitado. Esta es una excelente manera de evitar el agotamiento personal.

La respuesta simple a "¿y ahora qué?" Es para *directo*esta máquina y manténgala saludable. Pero a menos que haya una crisis, debe usar un toque suave. El libro *Equipos de depuración*² tiene una parábola sobre hacer ajustes conscientes:

Hay una historia sobre un maestro de todas las cosas mecánicas que se retiró hace mucho tiempo. Su antigua empresa tenía un problema que nadie podía solucionar, por lo que llamaron al Maestro para ver si podía ayudar a encontrar el problema. El Maestro examinó la máquina, la escuchó y finalmente sacó un trozo de tiza desgastado e hizo una pequeña X en el costado de la máquina. Informó al técnico que había un cable suelto que necesitaba reparación en ese mismo lugar. El técnico abrió la máquina y tensó el cable suelto, solucionando así el problema. Cuando llegó la factura del Maestro por \$10,000, el airado CEO respondió exigiendo un desglose de este cargo ridículamente alto por una simple marca de tiza. El Maestro respondió con otra factura, mostrando un costo de \$1 por la tiza para hacer la marca y \$9,999 por saber dónde colocarla.

Para nosotros, esta es una historia sobre la sabiduría: que un solo ajuste cuidadosamente considerado puede tener efectos gigantescos. Utilizamos esta técnica cuando gestionamos personas. Imaginamos a nuestro equipo como volando en un gran dirigible, dirigiéndose lenta y seguramente en cierta dirección. En lugar de microgestionar y tratar de hacer correcciones de curso continuas, pasamos la mayor parte de la semana observando y escuchando atentamente. Al final de la semana hacemos una pequeña marca con tiza en un lugar preciso en el dirigible, luego damos un pequeño pero crítico "toque" para ajustar el rumbo.

De esto se trata una buena gestión: 95% de observación y escucha, y 5% de hacer ajustes críticos en el lugar correcto. Escuche a sus líderes y salte los informes. Hable con sus clientes y recuerde que a menudo (especialmente si su equipo construye infraestructura de ingeniería), sus "clientes" no son usuarios finales en el mundo, sino sus compañeros de trabajo. La felicidad de los clientes requiere una escucha tan intensa como la felicidad de sus informes. ¿Qué funciona y qué no? ¿Este dirigible autónomo se dirige en la dirección correcta? Su dirección debe ser iterativa, pero reflexiva y mínima, haciendo los ajustes mínimos necesarios para corregir el rumbo. Si retrocede a la microgestión, corre el riesgo de convertirse en un SPOF nuevamente. "Always Be Leaving" es un llamado a *macro* administración.

Tenga cuidado al anclar la identidad de un equipo.

Un error común es poner a un equipo a cargo de un producto específico en lugar de un problema general. Un producto es *una solución* a un problema. La expectativa de vida de las soluciones puede ser corta y los productos pueden ser reemplazados por mejores soluciones. Sin embargo, un *problema* - si se elige bien - puede ser de hoja perenne. Anclar la identidad de un equipo a una solución específica ("Somos el equipo que administra los repositorios de Git") puede generar todo tipo de angustia con el tiempo. ¿Qué sucede si un gran porcentaje de sus ingenieros desea cambiar a un nuevo sistema de control de versiones? Es probable que el equipo "se involucre", defienda su solución y resista

² Brian W. Fitzpatrick y Ben Collins-Sussman, *Equipos de depuración: Mejor productividad a través de la colaboración* (Boston: O'Reilly, 2016).

cambio, aunque este no sea el mejor camino para la organización. El equipo se aferra a sus anteojeras, porque la solución se ha convertido en parte de la identidad y autoestima del equipo. Si el equipo en cambio posee el *problema*(ej., "Somos el equipo que proporciona el control de versiones a la empresa"), queda libre para experimentar con diferentes soluciones a lo largo del tiempo.

Siempre esté escalando

Muchos libros de liderazgo hablan de "escala" en el contexto de aprender a "maximizar su impacto": estrategias para hacer crecer su equipo e influencia. No vamos a discutir esas cosas aquí más allá de lo que ya hemos mencionado. Probablemente sea obvio que construir una organización autónoma con líderes fuertes ya es una gran receta para el crecimiento y el éxito.

En su lugar, vamos a discutir la escalabilidad del equipo desde un *defensivo* punto de vista personal en lugar de uno ofensivo. Como un líder, *su recurso más preciado es su reserva limitada de tiempo, atención y energía*. Si desarrolla agresivamente las responsabilidades y el poder de sus equipos sin aprender a proteger su cordura personal en el proceso, la escala está condenada al fracaso. Y entonces vamos a hablar sobre cómo escalar efectivamente *usted misma* través de este proceso.

El ciclo del éxito

Cuando un equipo aborda un problema difícil, surge un patrón estándar, un ciclo particular. Se parece a esto:

Análisis

Primero, recibes el problema y comienzas a lidiar con él. Usted identifica las anteojeras, encuentra todas las ventajas y desventajas y construye un consenso sobre cómo manejarlas.

Difícil

Comienza a avanzar en el trabajo, independientemente de si su equipo cree que está listo o no. Te preparas para fallas, reintentos e iteraciones. En este punto, su trabajo se trata principalmente de pastorear gatos. Anime a sus líderes y expertos sobre el terreno a formar opiniones y luego escuche atentamente y diseñe una estrategia general, incluso si tiene que " fingir" al principio.³

³ Es fácil que el síndrome del impostor se active en este punto. Una técnica para combatir la sensación de que no sé lo que estás haciendo es simplemente fingir que *alguno* experto sabe exactamente qué hacer, y que simplemente están de vacaciones y usted los reemplaza temporalmente. Es una excelente manera de eliminar las apuestas personales y darse permiso para fallar y aprender.

Tracción

Eventualmente, su equipo comienza a resolver las cosas. Está tomando decisiones más inteligentes y se logra un progreso real. La moral mejora. Está iterando en las concesiones y la organización está comenzando a solucionar el problema. ¡Buen trabajo!

Premio

Sucede algo inesperado. Su gerente lo lleva a un lado y lo felicita por su éxito. Descubres que tu recompensa no es solo una palmada en la espalda, sino una *problema completamente nuevo* para hacer frente a. Así es: la recompensa por el éxito es más trabajo... ¡y más responsabilidad! A menudo, es un problema similar o adyacente al primero, pero igualmente difícil.

Así que ahora estás en un lío. Se le ha dado un nuevo problema, pero (generalmente) no más personas. De alguna manera tienes que resolver *ambas cosas* problemas ahora, lo que probablemente signifique que el problema original todavía necesita ser manejado *con mitad* tantas personas en *mitad* del tiempo. ¡Necesitas a la otra mitad de tu gente para abordar el nuevo trabajo! Nos referimos a este último paso como la *etapa de compresión*: estás tomando todo lo que has estado haciendo y comprimiéndolo a la mitad del tamaño.

Entonces, en realidad, el ciclo del éxito es más una espiral (ver Figura 6-2). A lo largo de meses y años, su organización se amplía al abordar nuevos problemas y luego descubrir cómo comprimirlos para poder enfrentar nuevas luchas paralelas. Si tiene suerte, puede contratar a más personas a medida que avanza. Sin embargo, la mayoría de las veces, su contratación no sigue el ritmo de la escala. Larry Page, uno de los fundadores de Google, probablemente se referiría a esta espiral como "incómodamente emocionante".

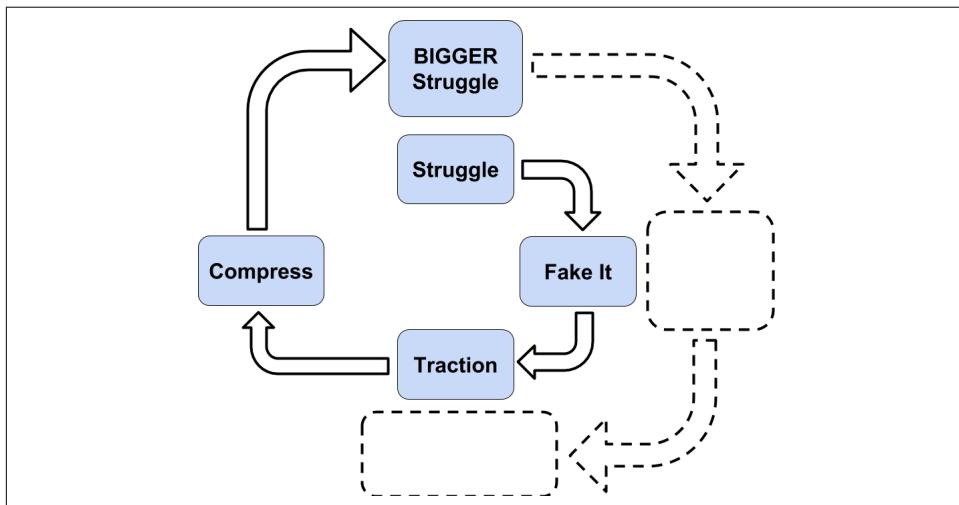


Figura 6-2. La espiral del éxito

La espiral del éxito es un enigma: es algo difícil de manejar y, sin embargo, es el paradigma principal para escalar un equipo de equipos. El acto de comprimir un problema no se trata solo de descubrir cómo maximizar la eficiencia de su equipo, sino también de aprender a escalar su *propio* tiempo y atención para estar a la altura de la nueva amplitud de responsabilidad.

Importante versus Urgente

Piense en una época en la que aún no era un líder, pero seguía siendo un colaborador individual despreocupado. Si solía ser programador, su vida probablemente era más tranquila y libre de pánico. Tenía una lista de trabajo por hacer, y cada día trabajaba metódicamente en su lista, escribiendo código y depurando problemas. Priorizar, planificar y ejecutar su trabajo fue sencillo.

Sin embargo, a medida que avanzaba hacia el liderazgo, es posible que haya notado que su principal modo de trabajo se volvió menos predecible y más relacionado con la extinción de incendios. Es decir, tu trabajo se volvió menos *proactivo* y más *reactivo*. Cuanto más alto esté en el liderazgo, más escalamientos recibirá. ¡Eres la cláusula "finalmente" en una larga lista de bloques de código! Todos sus medios de comunicación (correo electrónico, salas de chat, reuniones) comienzan a sentirse como un ataque de denegación de servicio contra su tiempo y atención. De hecho, si no eres consciente, terminas pasando el 100 % de tu tiempo en modo reactivo. La gente te tira pelotas y tú saltas frenéticamente de una pelota a la siguiente, tratando de que ninguna toque el suelo.

Muchos libros han tratado este problema. El autor de administración Stephen Covey es famoso por hablar sobre la idea de distinguir entre las cosas que son *importante* contra las cosas que son *urgente*. De hecho, fue el presidente estadounidense Dwight D. Eisenhower quien popularizó esta idea en una famosa cita de 1954:

Tengo dos clases de problemas, los urgentes y los importantes. Los urgentes no son importantes, y los importantes nunca son urgentes.

Esta tensión es uno de los mayores peligros para su eficacia como líder. Si te permites pasar al modo reactivo puro (lo que ocurre casi automáticamente), dedicas cada momento de tu vida a *urgentes* cosas, pero casi ninguna de esas cosas son *importante* en el cuadro grande. Recuerde que su trabajo como líder es hacer cosas que *solo tu puedes hacer*, como trazar un camino a través del bosque. Construir esa metaestrategia es increíblemente importante, pero casi nunca urgente. Siempre es más fácil responder al próximo correo electrónico urgente.

Entonces, ¿cómo puedes obligarte a trabajar principalmente en cosas importantes, en lugar de cosas urgentes? Aquí hay algunas técnicas clave:

Delegar

Muchas de las cosas urgentes que ve se pueden delegar a otros líderes de su organización. Puede sentirse culpable si se trata de una tarea trivial; o podrías preocuparte de que

traspasar un problema es ineficiente porque a esos otros líderes les puede llevar más tiempo solucionarlo. Pero es un buen entrenamiento para ellos y le libera tiempo para trabajar en cosas importantes que solo usted puede hacer.

Programar tiempo dedicado

Reserva regularmente dos horas o más para sentarte en silencio y trabajar. *solo en cosas importantes pero no urgentes, como la estrategia del equipo, las trayectorias profesionales de sus líderes o cómo planea colaborar con los equipos vecinos.*

Encuentre un sistema de rastreo que funcione

Hay docenas de sistemas para rastrear y priorizar el trabajo. Algunas se basan en software (p. ej., herramientas específicas de "tareas pendientes"), otras se basan en lápiz y papel (la "**Bullet Journal**"), y algunos sistemas son independientes de la implementación. En esta última categoría, el libro de David Allen, *Resolviendo las cosas*, es bastante popular entre los gerentes de ingeniería; es un algoritmo abstracto para trabajar en tareas y mantener un preciado "cero en la bandeja de entrada". El punto aquí es intentar estos diferentes sistemas y determinar lo que funciona para usted. Algunos de ellos harán clic con usted y otros no, pero definitivamente necesita encontrar algo más efectivo que las pequeñas notas Post-It que decoran la pantalla de su computadora.

Aprende a soltar bolas

Hay una técnica clave más para administrar su tiempo, y en la superficie suena radical. Para muchos, contradice años de instinto de ingeniería. Como ingeniero, prestas atención a los detalles; haces listas, marcas las cosas de las listas, eres preciso y terminas lo que empiezas. Es por eso que se siente tan bien cerrar errores en un rastreador de errores o reducir su correo electrónico a cero en la bandeja de entrada. Pero como líder de líderes, su tiempo y atención están bajo constante ataque. No importa cuánto intentes evitarlo, terminas tirando pelotas al suelo; simplemente te están arrojando demasiadas. Es abrumador y probablemente te sientas culpable por esto todo el tiempo.

Entonces, en este punto, demos un paso atrás y echemos un vistazo franco a la situación. Si dejar caer una cierta cantidad de bolas es inevitable, ¿no es mejor dejar caer ciertas bolas? *deliberadamente* en vez de *accidentalmente*? Al menos entonces tienes una apariencia de control.

Aquí hay una gran manera de hacer eso.

Marie Kondo es consultora organizacional y autora del libro extremadamente popular *La magia que cambia la vida de ordenar*. Su filosofía se trata de ordenar efectivamente toda la basura de su casa, pero también funciona para el desorden abstracto.

Piense en sus posesiones físicas como viviendo en tres montones. Alrededor del 20% de tus cosas son simplemente inútiles, cosas que literalmente nunca más tocas, y todas son muy fáciles de tirar. Alrededor del 60% de tus cosas son algo interesantes; varían en importancia para ti, ya veces los usas, a veces no. Y luego, alrededor del 20% de tus posesiones son extremadamente importantes: estas son las cosas que usas *todas* el tiempo,

que tienen un profundo significado emocional o, en palabras de la Sra. Kondo, despiertan una profunda "alegría" con solo sostenerlos. La tesis de su libro es que la mayoría de las personas ordenan sus vidas de manera incorrecta: pasan el tiempo tirando el 20% inferior a la basura, pero el 80% restante todavía se siente demasiado desordenado. Ella argumenta que la *verdadera* el trabajo de limpieza consiste en identificar el 20 % superior, no el 20 % inferior. Si puede identificar solo las cosas críticas, debe desechar el otro 80%. Suena extremo, pero es bastante efectivo. Es muy liberador ordenar tan radicalmente.

Resulta que también puedes aplicar esta filosofía a tu bandeja de entrada o lista de tareas: el aluvión de bolas que te lanzan. Divide tu montón de bolas en tres grupos: el 20% inferior probablemente no sea ni urgente ni importante y muy fácil de eliminar o ignorar. Hay un 60% medio, que puede contener algunos detalles de urgencia o importancia, pero es una bolsa mixta. En la parte superior, hay un 20% de cosas que son absolutamente importantes.

Y ahora, mientras trabaja en sus tareas, haga *no* intente abordar el 80 % superior: aún terminará abrumado y trabajando principalmente en tareas urgentes pero no importantes. En cambio, identifique conscientemente las bolas que caen estrictamente en el 20% superior: cosas críticas que *solo tu puedes hacer*—y centrarse estrictamente en ellos. Date permiso explícito para soltar el otro 80%.

Puede parecer terrible hacerlo al principio, pero a medida que dejas caer tantas bolas deliberadamente, descubrirás dos cosas asombrosas. En primer lugar, incluso si no delega ese 60% medio de las tareas, sus sublíderes a menudo las notan y las retoman automáticamente. En segundo lugar, si algo en ese cubo intermedio es realmente crítico, termina regresando a usted de todos modos, y eventualmente migra al 20% superior. Simplemente necesitas *confianza* que las cosas por debajo de su umbral del 20% superior se solucionarán o evolucionarán adecuadamente. Mientras tanto, debido a que se enfoca solo en las cosas de importancia crítica, puede escalar su tiempo y atención para cubrir las responsabilidades cada vez mayores de su grupo.

Protegiendo tu energía

Hemos hablado de proteger su tiempo y atención, pero su energía personal es la otra pieza de la ecuación. Toda esta escala es simplemente agotadora. En un entorno como este, ¿cómo te mantienes cargado y optimista?

Parte de la respuesta es que con el tiempo, a medida que envejeces, tu resistencia general se acumula. Al principio de su carrera, trabajar ocho horas al día en una oficina puede parecer un shock; llegas a casa cansado y aturrido. Pero al igual que entrenar para un maratón, tu cerebro y tu cuerpo acumulan mayores reservas de resistencia con el tiempo.

La otra parte clave de la respuesta es que los líderes gradualmente aprenden *agestionar* su energía más inteligentemente. Es algo a lo que aprenden a prestar atención constante. Típicamente, esto significa ser consciente de cuánta energía tienes en un momento dado,

y tomar decisiones deliberadas para "recargarse" en momentos específicos, de maneras específicas. Estos son algunos excelentes ejemplos de gestión consciente de la energía:

Llevar verdaderas vacaciones

Un fin de semana no son vacaciones. Se necesitan al menos tres días para "olvidarse" de su trabajo; se necesita al menos una semana para sentirse renovado. Pero si revisas el correo electrónico o los chats de tu trabajo,*ruina* la recarga Una avalancha de preocupación vuelve a tu mente y todos los beneficios del distanciamiento psicológico se disipan. Las vacaciones se recargan solo si eres verdaderamente disciplinado para desconectarte.⁴ Y, por supuesto, esto solo es posible si ha creado una organización autónoma.

Haz que desconectar sea trivial

Cuando desconecte, deje su computadora portátil de trabajo en la oficina. Si tiene comunicaciones de trabajo en su teléfono, elimínelas. Por ejemplo, si tu empresa usa G Suite (Gmail, Google Calendar, etc.), un gran truco es instalar estas aplicaciones en un "perfil de trabajo" en tu teléfono. Esto hace que aparezca un segundo conjunto de aplicaciones con insignia de trabajo en su teléfono. Por ejemplo, ahora tendrá dos aplicaciones de Gmail: una para el correo electrónico personal y otra para el correo electrónico del trabajo. En un teléfono Android, puede presionar un solo botón para desactivar todo el perfil de trabajo a la vez. Todas las aplicaciones de trabajo se atenúan, como si se hubieran desinstalado, y no puede revisar los mensajes de trabajo "accidentalmente" hasta que vuelva a habilitar el perfil de trabajo.

Llevar verdaderos fines de semana también

Un fin de semana no es tan efectivo como unas vacaciones, pero aun así tiene cierto poder rejuvenecedor. Nuevamente, esta recarga solo funciona si te desconectas de las comunicaciones laborales. Intente cerrar sesión realmente el viernes por la noche, pase el fin de semana haciendo las cosas que le gustan y luego vuelva a iniciar sesión el lunes por la mañana cuando esté de vuelta en la oficina.

Tomar descansos durante el día

Su cerebro opera en ciclos naturales de 90 minutos.⁵ Aproveche la oportunidad para levantarse y caminar por la oficina, o pase 10 minutos caminando afuera. Pequeños descansos como este son solo pequeñas recargas, pero pueden marcar una gran diferencia en sus niveles de estrés y en cómo se siente durante las próximas dos horas de trabajo.

Date permiso para tomarte un día de salud mental

A veces, sin razón, simplemente tienes un mal día. Es posible que haya dormido bien, comido bien, hecho ejercicio y, sin embargo, todavía está de un humor terrible de todos modos. Si eres un líder, esto es algo horrible. Tu mal humor marca la pauta para todos los que te rodean y puede conducir a decisiones terribles (correos electrónicos que no deberías haber enviado, juicios demasiado duros, etc.). Si te encuentras en esta situación, simplemente date la vuelta.

4 Debe planificar con anticipación y basarse en el supuesto de que su trabajo simplemente no se realizará durante las vacaciones. Trabajar duro (o inteligentemente) justo antes y después de sus vacaciones mitiga este problema.

5 Puede leer más sobre BRAC en https://en.wikipedia.org/wiki/Basic_rest-activity_cycle.

e ir a casa, declarando un día de enfermedad. Es mejor no hacer nada ese día que hacer daño activo.

Al final, administrar tu energía es tan importante como administrar tu tiempo. Si aprende a dominar estas cosas, estará listo para abordar el ciclo más amplio de escalar la responsabilidad y construir un equipo autosuficiente.

Conclusión

Los líderes exitosos naturalmente asumen más responsabilidades a medida que progresan (y eso es algo bueno y natural). A menos que encuentren técnicas efectivas para tomar decisiones rápidamente, delegar cuando sea necesario y gestionar su mayor responsabilidad, podrían terminar sintiéndose abrumados. Ser un líder eficaz no significa que deba tomar decisiones perfectas, hacer todo usted mismo o trabajar el doble de duro. En su lugar, esfuérzate por estar siempre decidiendo, siempre saliendo y siempre escalando.

TL; DR

- Sea siempre decisivo: los problemas ambiguos no tienen una respuesta mágica; se trata de encontrar el derecho *compensaciones* del momento, e iterando.
- Siempre se va: su trabajo, como líder, es construir una organización que resuelva automáticamente una clase de problemas ambiguos, sobre *hora*—sin necesidad de que estés presente.
- Sea siempre escalable: el éxito genera más responsabilidad con el tiempo, y debe administrar de manera proactiva la *escalada* de este trabajo para proteger sus escasos recursos de tiempo personal, atención y energía.

Medición de la productividad de la ingeniería

*Escrito por Ciera Jaspen
Editado por Riona Macnamara*

Google es una empresa basada en datos. Respaldamos la mayoría de nuestros productos y decisiones de diseño con datos duros. La cultura de la toma de decisiones basada en datos, utilizando métricas apropiadas, tiene algunos inconvenientes, pero en general, confiar en los datos tiende a hacer que la mayoría de las decisiones sean objetivas en lugar de subjetivas, lo que a menudo es algo bueno. Sin embargo, recopilar y analizar datos sobre el lado humano de las cosas tiene sus propios desafíos. Específicamente, dentro de la ingeniería de software, Google descubrió que tener un equipo de especialistas enfocado en la productividad de la ingeniería en sí misma es muy valioso e importante a medida que la empresa escala y puede aprovechar los conocimientos de dicho equipo.

¿Por qué debemos medir la productividad de la ingeniería?

Supongamos que tiene un negocio próspero (por ejemplo, ejecuta un motor de búsqueda en línea) y desea aumentar el alcance de su negocio (ingresar al mercado de aplicaciones empresariales, al mercado de la nube o al mercado móvil). Presumiblemente, para aumentar el alcance de su negocio, también deberá aumentar el tamaño de su organización de ingeniería. Sin embargo, a medida que las organizaciones crecen en tamaño linealmente, los costos de comunicación crecen cuadráticamente.¹ Será necesario agregar más personas para aumentar el alcance de su negocio, pero los costos generales de comunicación no aumentarán linealmente a medida que agregue personal adicional. Como resultado, no podrá escalar el alcance de su negocio linealmente al tamaño de su organización de ingeniería.

¹ Frederick P. Brooks, *El hombre-mes mítico: ensayos sobre ingeniería de software* (Nueva York: Addison-Wesley, 1995).

Sin embargo, hay otra forma de abordar nuestro problema de escalado:*podríamos hacer que cada individuo sea más productivo*. Si podemos aumentar la productividad de los ingenieros individuales en la organización, podemos aumentar el alcance de nuestro negocio sin el aumento proporcional de los gastos generales de comunicación.

Google ha tenido que crecer rápidamente en nuevos negocios, lo que ha significado aprender cómo hacer que nuestros ingenieros sean más productivos. Para hacer esto, necesitábamos comprender qué los hace productivos, identificar las ineficiencias en nuestros procesos de ingeniería y solucionar los problemas identificados. Luego, repetiríamos el ciclo según sea necesario en un ciclo de mejora continua. Al hacer esto, podríamos escalar nuestra organización de ingeniería con el aumento de la demanda.

Sin embargo, este ciclo de mejora *además* toma recursos humanos. No valdría la pena mejorar la productividad de su organización de ingeniería en el equivalente de 10 ingenieros por año si se necesitaran 50 ingenieros por año para comprender y solucionar los bloqueos de productividad. *Por lo tanto, nuestro objetivo no es solo mejorar la productividad de la ingeniería de software, sino hacerlo de manera eficiente*.

En Google, abordamos estas compensaciones mediante la creación de un equipo de investigadores dedicados a comprender la productividad de la ingeniería. Nuestro equipo de investigación incluye personas del campo de investigación de ingeniería de software e ingenieros de software generalistas, pero también incluye científicos sociales de una variedad de campos, incluida la psicología cognitiva y la economía del comportamiento. La incorporación de personas de las ciencias sociales nos permite no solo estudiar los artefactos de software que producen los ingenieros, sino también comprender el lado humano del desarrollo de software, incluidas las motivaciones personales, las estructuras de incentivos y las estrategias para administrar tareas complejas. El objetivo del equipo es adoptar un enfoque basado en datos para medir y mejorar la productividad de la ingeniería.

En este capítulo, explicamos cómo nuestro equipo de investigación logra este objetivo. Esto comienza con el proceso de clasificación: hay muchas partes del desarrollo de software que *pueden medir*, pero *que debería medir*? Después de seleccionar un proyecto, explicamos cómo el equipo de investigación identifica métricas significativas que identificarán las partes problemáticas del proceso. Finalmente, observamos cómo Google usa estas métricas para rastrear las mejoras en la productividad.

Para este capítulo, seguimos un ejemplo concreto planteado por los equipos de lenguaje C++ y Java de Google: la legibilidad. Durante la mayor parte de la existencia de Google, estos equipos han gestionado el proceso de legibilidad en Google. (Para obtener más información sobre la legibilidad, consulte [Capítulo 3](#).) El proceso de legibilidad se implementó en los primeros días de Google, antes de los formatos automáticos ([Capítulo 8](#)) y los linters que bloquean el envío eran comunes ([Capítulo 9](#)). El proceso en sí es costoso de ejecutar porque requiere que cientos de ingenieros realicen revisiones de legibilidad para otros ingenieros a fin de otorgarles legibilidad. Algunos ingenieros lo vieron como un proceso arcaico de novatadas que ya no tenía utilidad, y era un tema favorito para discutir alrededor de la mesa del almuerzo. El hormigón

La pregunta de los equipos de idiomas fue la siguiente: ¿Vale la pena el tiempo dedicado al proceso de legibilidad?

Triaje: ¿Vale la pena medirlo?

Antes de decidir cómo medir la productividad de los ingenieros, necesitamos saber cuándo vale la pena medir una métrica. La medición en sí es costosa: se necesita gente para medir el proceso, analizar los resultados y difundirlos al resto de la empresa. Además, el proceso de medición en sí puede ser oneroso y ralentizar el resto de la organización de ingeniería. Incluso si no es lento, el seguimiento del progreso puede cambiar el comportamiento de los ingenieros, posiblemente de manera que enmascare los problemas subyacentes. Necesitamos medir y estimar inteligentemente; aunque no queramos adivinar, no debemos perder tiempo y recursos midiendo innecesariamente.

En Google, hemos creado una serie de preguntas para ayudar a los equipos a determinar si vale la pena medir la productividad en primer lugar. Primero le pedimos a las personas que describan lo que quieren medir en forma de una pregunta concreta; encontramos que cuanto más concretas puedan hacer las personas esta pregunta, más probable es que se beneficien del proceso. Cuando el equipo de legibilidad se acercó a nosotros, su pregunta era simple: ¿los costos de un ingeniero que pasa por el proceso de legibilidad valen los beneficios que podrían derivar para la empresa?

Luego les pedimos que consideren los siguientes aspectos de su pregunta:

¿Qué resultado esperas y por qué?

Aunque nos gustaría pretender que somos investigadores neutrales, no lo somos.

Tenemos nociones preconcebidas sobre lo que debería suceder. Al reconocer esto desde el principio, podemos tratar de abordar estos sesgos y evitar explicaciones post hoc de los resultados.

Cuando se le planteó esta pregunta al equipo de legibilidad, señaló que no estaba seguro. La gente estaba segura de que los costos habían valido la pena en algún momento, pero con la llegada de los formateadores automáticos y las herramientas de análisis estático, nadie estaba completamente seguro. Había una creencia creciente de que el proceso ahora servía como un ritual de novatadas. Aunque aún podría brindar beneficios a los ingenieros (y tenían datos de encuestas que mostraban que las personas reclamaban estos beneficios), no estaba claro si valía la pena el compromiso de tiempo de los autores o los revisores del código.

Si los datos respaldan el resultado esperado, ¿qué acción se tomará?

Preguntamos esto porque si no se toman medidas, no tiene sentido medir. Tenga en cuenta que una acción podría ser, de hecho, "mantener el statu quo" si hay un cambio planificado que ocurrirá si no tuviéramos este resultado.

Cuando se les preguntó acerca de esto, la respuesta del equipo de legibilidad fue directa: si el beneficio era suficiente para justificar los costos del proceso, vincularían la investigación y los datos en las preguntas frecuentes sobre legibilidad y los anunciarían para establecer expectativas.

Si obtenemos un resultado negativo, ¿se tomarán las medidas adecuadas?

Hacemos esta pregunta porque en muchos casos encontramos que un resultado negativo no cambiará una decisión. Puede haber otras entradas en una decisión que anularían cualquier resultado negativo. Si ese es el caso, puede que no valga la pena medir en primer lugar. Esta es la pregunta que frena la mayoría de los proyectos que emprende nuestro equipo de investigación; aprendemos que los tomadores de decisiones estaban interesados en conocer los resultados, pero por otras razones, no optarán por cambiar de rumbo.

Sin embargo, en el caso de la legibilidad, tuvimos una fuerte declaración de acción del equipo. Se comprometió a que, si nuestro análisis mostraba que los costos superaban los beneficios o que los beneficios eran insignificantes, el equipo acabaría con el proceso.

Como los diferentes lenguajes de programación tienen diferentes niveles de madurez en formateadores y análisis estáticos, esta evaluación se realizaría por lenguaje.

¿Quién va a decidir tomar acción sobre el resultado y cuándo lo hará?

Pedimos esto para asegurar que la persona que solicita la medición es la que está facultada para actuar (o lo está haciendo directamente en su nombre). En última instancia, el objetivo de medir nuestro proceso de software es ayudar a las personas a tomar decisiones comerciales. Es importante comprender quién es esa persona, incluida la forma de datos que la convence. Aunque la mejor investigación incluye una variedad de enfoques (desde entrevistas estructuradas hasta análisis estadísticos de registros), puede haber un tiempo limitado para proporcionar a los tomadores de decisiones los datos que necesitan. En esos casos, podría ser mejor atender al tomador de decisiones. ¿Tienen a tomar decisiones empatizando a través de las historias que se pueden recuperar de las entrevistas?² ¿Confían en los resultados de las encuestas o en los datos de los registros? ¿Se sienten cómodos con análisis estadísticos complejos? Si el decisor no cree en la forma del resultado en principio, nuevamente no tiene sentido medir el proceso.

En el caso de la legibilidad, teníamos un tomador de decisiones claro para cada lenguaje de programación. Dos equipos de lenguaje, Java y C++, se comunicaron activamente con nosotros en busca de ayuda, y los demás estaban esperando a ver qué pasaba con esos lenguajes.

² Vale la pena señalar aquí que nuestra industria actualmente menosprecia las "anécdotas", y todos tienen el objetivo de ser "impulsado por datos". Sin embargo, las anécdotas continúan existiendo porque son poderosas. Una anécdota puede proporcionar un contexto y una narrativa que los números en bruto no pueden; puede proporcionar una explicación profunda que resuena con otros porque refleja la experiencia personal. Aunque nuestros investigadores no toman decisiones sobre anécdotas, sí usamos y fomentamos técnicas como entrevistas estructuradas y estudios de casos para comprender profundamente los fenómenos y brindar contexto a los datos cuantitativos.

primero.³ Los tomadores de decisiones confiaron en las experiencias autoinformadas de los ingenieros para comprender la felicidad y el aprendizaje, pero los tomadores de decisiones querían ver "cifras concretas" basadas en datos de registros para la velocidad y la calidad del código. Esto significaba que necesitábamos incluir análisis tanto cualitativos como cuantitativos para estas métricas. No hubo una fecha límite estricta para este trabajo, pero hubo una conferencia interna que sería un momento útil para anunciar si iba a haber un cambio. Ese plazo nos dio varios meses para completar el trabajo.

Al hacer estas preguntas, descubrimos que, en muchos casos, la medición simplemente no vale la pena... ¡y eso está bien! Hay muchas buenas razones para no medir el impacto de una herramienta o proceso en la productividad. Estos son algunos ejemplos que hemos visto:

No puede darse el lujo de cambiar el proceso/herramientas en este momento

Puede haber limitaciones de tiempo o financieras que lo impidan. Por ejemplo, puede determinar que si solo cambiara a una herramienta de compilación más rápida, ahorraría horas de tiempo cada semana. Sin embargo, el cambio significaría pausar el desarrollo mientras todos se convierten, y se acerca una fecha límite de financiación importante que no puede permitirse la interrupción. Las compensaciones de ingeniería no se evalúan en el vacío; en un caso como este, es importante darse cuenta de que el contexto más amplio justifica completamente retrasar la acción sobre un resultado.

Cualquier resultado pronto será invalidado por otros factores.

Los ejemplos aquí podrían incluir la medición del proceso de software de una organización justo antes de una reorganización planificada. O medir la cantidad de deuda técnica para un sistema obsoleto.

El tomador de decisiones tiene opiniones fuertes, y es poco probable que usted pueda proporcionar un cuerpo de evidencia lo suficientemente grande, del tipo correcto, para cambiar sus creencias.

Esto se reduce a conocer a tu audiencia. Incluso en Google, a veces encontramos personas que tienen creencias inquebrantables sobre un tema debido a sus experiencias pasadas. Hemos encontrado partes interesadas que nunca confían en los datos de las encuestas porque no creen en los autoinformes. También hemos encontrado partes interesadas que se dejan influir mejor por una narrativa convincente que fue informada por un pequeño número de entrevistas. Y, por supuesto, hay partes interesadas que solo se dejan influir por el análisis de registros. En todos los casos, intentamos triangular la verdad usando métodos mixtos, pero si un actor se limita a creer solo en métodos que no son apropiados para el problema, no tiene sentido hacer el trabajo.

³ Java y C++ tienen la mayor cantidad de soporte de herramientas. Ambos tienen formateadores maduros y análisis estático.

herramientas que detectan errores comunes. Ambos también están fuertemente financiados internamente. Aunque otros equipos de lenguaje, como Python, estaban interesados en los resultados, claramente no habría ningún beneficio para que Python eliminara la legibilidad si ni siquiera podíamos mostrar el mismo beneficio para Java o C++.

Los resultados se usarán solo como métricas de vanidad para respaldar algo que iba a hacer de todos modos

Esta es quizás la razón más común por la que le decimos a la gente de Google que no mida un proceso de software. Muchas veces, las personas han planeado una decisión por múltiples razones, y mejorar el proceso de desarrollo de software es solo un beneficio de varios. Por ejemplo, el equipo de herramientas de lanzamiento de Google solicitó una vez una medición de un cambio planificado en el sistema de flujo de trabajo de lanzamiento. Debido a la naturaleza del cambio, era obvio que el cambio no sería peor que el estado actual, pero no sabían si era una mejora menor o grande. Le preguntamos al equipo: si resulta ser solo una mejora menor, ¿gastaría los recursos para implementar la función de todos modos, incluso si no pareciera que vale la pena la inversión? ¡La respuesta fue sí! La función mejoró la productividad, pero esto fue un efecto secundario:

Las únicas métricas disponibles no son lo suficientemente precisas para medir el problema y pueden confundirse con otros factores

En algunos casos, las métricas necesarias (consulte la próxima sección sobre cómo identificar métricas) simplemente no están disponibles. En estos casos, puede ser tentador medir usando otras métricas menos precisas (líneas de código escritas, por ejemplo). Sin embargo, cualquier resultado de estas métricas será ininterpretable. Si la métrica confirma las creencias preexistentes de las partes interesadas, es posible que terminen procediendo con su plan sin tener en cuenta que la métrica no es una medida precisa. Si no confirma sus creencias, la imprecisión de la métrica en sí proporciona una explicación fácil y la parte interesada podría, nuevamente, continuar con su plan.

Cuando tiene éxito en la medición de su proceso de software, no se propone probar que una hipótesis es correcta o incorrecta; *el éxito significa dar a una parte interesada los datos que necesita para tomar una decisión*. Si esa parte interesada no usa los datos, el proyecto siempre es un fracaso. Solo debemos medir un proceso de software cuando se tomará una decisión concreta basada en el resultado. Para el equipo de legibilidad, había que tomar una decisión clara. Si las métricas mostraban que el proceso era beneficioso, publicitaban el resultado. De lo contrario, el proceso sería abolido. Lo más importante, el equipo de legibilidad tenía la autoridad para tomar esta decisión.

Selección de métricas significativas con objetivos y señales

Después de decidir medir un proceso de software, debemos determinar qué métricas utilizar.

Claramente, las líneas de código (LOC) no funcionarán,⁴ pero, ¿cómo medimos realmente la productividad de la ingeniería?

En Google, utilizamos el marco Goals/Signals/Metrics (GSM) para guiar la creación de métricas.

- *Aobjetivo* es un resultado final deseado. Está redactado en términos de lo que desea comprender en un nivel alto y no debe contener referencias a formas específicas de medirlo.
- Una señal es cómo puede saber que ha logrado el resultado final. Las señales son cosas que haríamos *comopara* medir, pero pueden no ser medibles en sí mismos.
- *Amétrico* es proxy de una señal. Es lo que realmente podemos medir. Puede que no sea la medida ideal, pero es algo que creemos que está lo suficientemente cerca.

El marco GSM fomenta varias propiedades deseables al crear métricas. Primero, al crear primero metas, luego señales y finalmente métricas, se evita la *efecto de farola*. El término proviene de la frase completa "buscar las llaves bajo la luz de la calle": si solo miras donde puedes ver, es posible que no estés buscando en el lugar correcto. Con las métricas, esto ocurre cuando usamos las métricas que tenemos fácilmente accesibles y que son fáciles de medir, independientemente de si esas métricas se adaptan a nuestras necesidades. En cambio, GSM nos obliga a pensar en qué métricas realmente nos ayudarán a lograr nuestros objetivos, en lugar de simplemente lo que tenemos disponible.

En segundo lugar, GSM ayuda a prevenir tanto el avance lento de las métricas como el sesgo de las métricas al alejarnos a crear el conjunto de métricas apropiado, utilizando un enfoque basado en principios, *por adelantado* de medir realmente el resultado. Considere el caso en el que seleccionamos métricas sin un enfoque de principios y luego los resultados no cumplen con las expectativas de nuestros grupos de interés. En ese punto, corremos el riesgo de que las partes interesadas propongan que usemos diferentes métricas que creen que producirán el resultado deseado. Y debido a que no seleccionamos en base a un enfoque de principios al principio, ¡no hay razón para decir que están equivocados! En cambio, GSM nos alienta a seleccionar métricas en función de su capacidad para medir los objetivos originales. Las partes interesadas pueden ver fácilmente que estas métricas se asignan a sus

4 "A partir de ahí, solo hay un pequeño paso para medir la 'productividad del programador' en términos de 'número de líneas de código producido por mes.' Esta es una unidad de medida muy costosa porque fomenta la escritura de código insípido, pero hoy en día estoy menos interesado en cuán tonta es una unidad incluso desde un punto de vista comercial puro. Mi punto de hoy es que, si deseamos contar líneas de código, no deberíamos considerarlas como 'líneas producidas' sino como 'líneas gastadas': la sabiduría convencional actual es tan tonta como para reservar que cuentan en el lado equivocado de la libro mayor." Edsger Dijkstra, sobre la crueldad de enseñar realmente ciencias de la computación, [EDW Manuscrito 1036](#).

metas originales y acuerde, de antemano, que este es el mejor conjunto de métricas para medir los resultados.

Finalmente, GSM puede mostrarnos dónde tenemos cobertura de medición y dónde no. Cuando ejecutamos el proceso GSM, enumeramos todos nuestros objetivos y creamos señales para cada uno. Como veremos en los ejemplos, no todas las señales van a ser medibles. ¡Y eso está bien! Con GSM, al menos hemos identificado lo que no es medible. Al identificar estas métricas faltantes, podemos evaluar si vale la pena crear nuevas métricas o incluso si vale la pena medirlas.

Lo importante es mantener *trazabilidad*. Para cada métrica, deberíamos poder rastrear la señal para la que pretende ser un proxy y el objetivo que está tratando de medir. Esto asegura que sabemos qué métricas estamos midiendo y por qué las estamos midiendo.

Metas

Un objetivo debe escribirse en términos de una propiedad deseada, sin referencia a ninguna métrica. Por sí mismos, estos objetivos no son medibles, pero un buen conjunto de objetivos es algo en lo que todos pueden estar de acuerdo antes de pasar a las señales y luego a las métricas.

Para que esto funcione, necesitamos haber identificado el conjunto correcto de objetivos para medir en primer lugar. Esto parecería sencillo: ¡seguramente el equipo conoce los objetivos de su trabajo! Sin embargo, nuestro equipo de investigación descubrió que, en muchos casos, las personas se olvidan de incluir todos los posibles *compensaciones dentro de la productividad*, lo que podría conducir a una medición incorrecta.

Tomando el ejemplo de la legibilidad, supongamos que el equipo estaba tan concentrado en hacer que el proceso de legibilidad fuera rápido y fácil que se había olvidado del objetivo de la calidad del código. El equipo configuró mediciones de seguimiento para determinar cuánto tiempo lleva completar el proceso de revisión y qué tan felices están los ingenieros con el proceso. Uno de nuestros compañeros propone lo siguiente:

Puedo hacer que su velocidad de revisión sea muy rápida: simplemente elimine las revisiones de código por completo.

Aunque este es obviamente un ejemplo extremo, los equipos se olvidan de las concesiones centrales todo el tiempo cuando miden: se enfocan tanto en mejorar la velocidad que se olvidan de medir la calidad (o viceversa). Para combatir esto, nuestro equipo de investigación divide la productividad en cinco componentes básicos. Estos cinco componentes se compensan entre sí, y alentamos a los equipos a considerar objetivos en cada uno de estos componentes para asegurarse de que no estén mejorando uno sin darse cuenta mientras empujan a otros hacia abajo. Para ayudar a las personas a recordar los cinco componentes, usamos el mnemotécnico "QUANTS":

***Q**ue **r**ealidad del código*

¿Cuál es la calidad del código producido? ¿Son los casos de prueba lo suficientemente buenos para evitar regresiones? ¿Qué tan buena es una arquitectura para mitigar riesgos y cambios?

***A**ttención de los ingenieros*

¿Con qué frecuencia los ingenieros alcanzan un estado de flujo? ¿Cuánto se distraen con las notificaciones? ¿Una herramienta anima a los ingenieros a cambiar de contexto?

***y**on **n**orte **c**omplejidad **i**ntelectual*

¿Cuánta carga cognitiva se requiere para completar una tarea? ¿Cuál es la complejidad inherente del problema que se está resolviendo? ¿Los ingenieros necesitan lidiar con una complejidad innecesaria?

***T**empo y **v**elocidad*

¿Qué tan rápido pueden los ingenieros realizar sus tareas? ¿Qué tan rápido pueden sacar sus lanzamientos? ¿Cuántas tareas completan en un período de tiempo determinado?

***S**satisfacción*

¿Qué tan felices están los ingenieros con sus herramientas? ¿Qué tan bien una herramienta satisface las necesidades de los ingenieros? ¿Qué tan satisfechos están con su trabajo y su producto final? ¿Los ingenieros se sienten agotados?

Volviendo al ejemplo de legibilidad, nuestro equipo de investigación trabajó con el equipo de legibilidad para identificar varios objetivos de productividad del proceso de legibilidad:

***C**alidad del código*

Los ingenieros escriben código de mayor calidad como resultado del proceso de legibilidad; escriben código más consistente como resultado del proceso de legibilidad; y contribuyen a una cultura de salud del código como resultado del proceso de legibilidad.

***A**tención de ingenieros.*

No teníamos ningún objetivo de atención para la legibilidad. ¡Esto está bien! No todas las preguntas sobre la productividad de la ingeniería implican compensaciones en las cinco áreas.

***C**omplejidad **i**ntelectual*

Los ingenieros aprenden sobre el código base de Google y las mejores prácticas de codificación como resultado del proceso de legibilidad y reciben tutoría durante el proceso de legibilidad.

***T**empo y **v**elocidad*

Los ingenieros completan las tareas de trabajo de manera más rápida y eficiente como resultado del proceso de legibilidad.

***S**satisfacción*

Los ingenieros ven el beneficio del proceso de legibilidad y tienen sentimientos positivos acerca de participar en él.

Señales

Una señal es la forma en la que sabremos que hemos conseguido nuestro objetivo. No todas las señales son medibles, pero eso es aceptable en esta etapa. No existe una relación 1:1 entre señales y goles. Cada objetivo debe tener al menos una señal, pero pueden tener más. Algunos objetivos también pueden compartir una señal. **Tabla 7-1** muestra algunas señales de ejemplo para los objetivos de la medición del proceso de legibilidad.

Tabla 7-1. Señales y goles

Metas	Señales
Los ingenieros escriben código de mayor calidad como resultado del proceso de legibilidad.	Los ingenieros a los que se les ha otorgado legibilidad juzgan que su código es de mayor calidad que los ingenieros a los que no se les ha otorgado legibilidad. El proceso de legibilidad tiene un impacto positivo en la calidad del código.
Los ingenieros aprenden sobre el código base de Google y las mejores prácticas de codificación como resultado del proceso de legibilidad.	Los ingenieros informan haber aprendido del proceso de legibilidad.
Los ingenieros reciben tutoría durante el proceso de legibilidad.	Los ingenieros informan interacciones positivas con ingenieros experimentados de Google que actúan como revisores durante el proceso de legibilidad.
Los ingenieros completan las tareas de trabajo de manera más rápida y eficiente como resultado del proceso de legibilidad.	Los ingenieros a los que se les ha otorgado legibilidad se consideran más productivos que los ingenieros a los que no se les ha otorgado legibilidad. Los cambios escritos por ingenieros a los que se les ha otorgado la legibilidad son más rápidos de revisar que los cambios escritos por ingenieros a los que no se les ha otorgado la legibilidad.
Los ingenieros ven el beneficio del proceso de legibilidad y tienen sentimientos positivos acerca de participar en él.	Los ingenieros consideran que el proceso de legibilidad merece la pena.

Métrica

Las métricas son donde finalmente determinamos cómo mediremos la señal. Las métricas no son la señal en sí; son el proxy medible de la señal. Debido a que son un proxy, es posible que no sean una medida perfecta. Por esta razón, algunas señales pueden tener múltiples métricas mientras tratamos de triangular la señal subyacente.

Por ejemplo, para medir si el código de los ingenieros se revisa más rápido después de la legibilidad, podríamos usar una combinación de datos de encuestas y datos de registros. Ninguna de estas métricas proporciona realmente la verdad subyacente. (Las percepciones humanas son falibles, y es posible que las métricas de registros no midan la imagen completa del tiempo que un ingeniero pasa revisando un fragmento de código o pueden confundirse con factores desconocidos en ese momento, como el tamaño o la dificultad de un cambio de código). Sin embargo, si estas métricas muestran resultados diferentes, indica que posiblemente una de ellas es incorrecta y necesitamos explorar más. Si son los mismos, tenemos más confianza en que hemos llegado a algún tipo de verdad.

Además, es posible que algunas señales no tengan ninguna métrica asociada porque la señal podría simplemente no ser medible en este momento. Considere, por ejemplo, medir la calidad del código. Aunque la literatura académica ha propuesto muchos indicadores de la calidad del código, ninguno de ellos la ha captado verdaderamente. Para facilitar la lectura, tuvimos la decisión de usar un proxy deficiente y posiblemente tomar una decisión basada en él, o simplemente reconocer que este es un punto que actualmente no se puede medir. En última instancia, decidimos no capturar esto como una medida cuantitativa, aunque les pedimos a los ingenieros que autoevaluaran la calidad de su código.

Seguir el marco GSM es una excelente manera de aclarar los objetivos de por qué está midiendo su proceso de software y cómo se medirá realmente. Sin embargo, aún es posible que las métricas seleccionadas no cuenten la historia completa porque no están capturando la señal deseada. En Google, usamos datos cualitativos para validar nuestras métricas y asegurarnos de que capturen la señal deseada.

Uso de datos para validar métricas

Como ejemplo, una vez creamos una métrica para medir la latencia de compilación mediana de cada ingeniero; el objetivo era capturar la "experiencia típica" de las latencias de construcción de los ingenieros. Luego ejecutamos un estudio de muestreo de experiencia. En este estilo de estudio, los ingenieros se interrumpen en el contexto de realizar una tarea de interés para responder algunas preguntas. Después de que un ingeniero comenzara una compilación, automáticamente le enviamos una pequeña encuesta sobre sus experiencias y expectativas de latencia de compilación. Sin embargo, en algunos casos, los ingenieros respondieron que ¡no habían comenzado una construcción! Resultó que las herramientas automatizadas estaban iniciando compilaciones, pero los ingenieros no estaban bloqueados en estos resultados y, por lo tanto, no "contaban" para su "experiencia típica". Luego ajustamos la métrica para excluir tales compilaciones.⁵

Las métricas cuantitativas son útiles porque le brindan poder y escala. Puede medir la experiencia de los ingenieros de toda la empresa durante un largo período de tiempo y tener confianza en los resultados. Sin embargo, no proporcionan ningún contexto o narrativa. Las métricas cuantitativas no explican por qué un ingeniero eligió usar una herramienta anticuada para realizar su tarea, o por qué tomó un flujo de trabajo inusual, o por qué estudió un proceso estándar. Solo los estudios cualitativos pueden proporcionar esta información, y solo los estudios cualitativos pueden proporcionar información sobre los próximos pasos para mejorar un proceso.

⁵ Ha sido rutinariamente nuestra experiencia en Google que cuando las métricas cuantitativas y cualitativas no están de acuerdo, fue porque las métricas cuantitativas no estaban capturando el resultado esperado.

Considere ahora las señales presentadas en [Tabla 7-2](#). ¿Qué métricas podría crear para medir cada uno de ellos? Algunas de estas señales pueden medirse mediante el análisis de registros de herramientas y códigos. Otros son medibles solo preguntando directamente a los ingenieros. Es posible que otros no sean perfectamente medibles; por ejemplo, ¿cómo medimos realmente la calidad del código?

En última instancia, al evaluar el impacto de la legibilidad en la productividad, terminamos con una combinación de métricas de tres fuentes. Primero, tuvimos una encuesta que trataba específicamente sobre el proceso de legibilidad. Esta encuesta se entregó a las personas después de que completaron el proceso; esto nos permitió obtener sus comentarios inmediatos sobre el proceso. Con suerte, esto evita el sesgo de recuerdo,⁶ pero introduce tanto el sesgo de actualidad⁷ y sesgo de muestreo.⁸ En segundo lugar, utilizamos una encuesta trimestral a gran escala para realizar un seguimiento de los elementos que no se referían específicamente a la legibilidad; en cambio, se trataban puramente de métricas que esperábamos que afectara la legibilidad. Por último, utilizamos métricas de registros detalladas de nuestras herramientas de desarrollador para determinar cuánto tiempo, según los registros, les tomó a los ingenieros completar tareas específicas.⁹ [Tabla 7-2](#) presenta la lista completa de métricas con sus correspondientes señales y objetivos.

Tabla 7-2. Objetivos, señales y métricas

CUANTOS	Objetivo	Señal	Métrico	
Q	ureabilidad del código	Los ingenieros escriben código de mayor calidad como resultado del proceso de legibilidad.	Los ingenieros a los que se les ha otorgado legibilidad juzgan que su código es de mayor calidad que los ingenieros a los que no se les ha otorgado legibilidad.	Encuesta Trimestral: Proporción de ingenieros que reportan estar satisfechos con la calidad de su propio código
		El proceso de legibilidad tiene un impacto positivo en la calidad del código.	Encuesta de legibilidad: proporción de ingenieros que informan que las revisiones de legibilidad no tienen impacto o tienen un impacto negativo en la calidad del código	

6 El sesgo de recuerdo es el sesgo de la memoria. Es más probable que las personas recuerden eventos que son particularmente interesantes o frustrante.

7 El sesgo de actualidad es otra forma de sesgo de memoria en el que las personas están sesgadas hacia sus experiencias más recientes. En este caso, como acaban de completar con éxito el proceso, es posible que se sientan particularmente bien al respecto.

8 Debido a que solo preguntamos a aquellas personas que completaron el proceso, no estamos capturando las opiniones de aquellos que no completó el proceso.

9 Existe la tentación de usar tales métricas para evaluar ingenieros individuales, o tal vez incluso para identificar altos y de bajo rendimiento. Sin embargo, hacerlo sería contraproducente. Si las métricas de productividad se utilizan para las revisiones de desempeño, los ingenieros se apresurarán a jugar con las métricas y ya no serán útiles para medir y mejorar la productividad en toda la organización. La única forma de hacer que estas mediciones funcionen es dejar de lado la idea de medir individuos y adoptar la medición del efecto agregado.

CUANTOS	Objetivo	Señal	Métrico
			Encuesta de legibilidad: proporción de ingenieros que informan que participar en el proceso de legibilidad ha mejorado la calidad del código para su equipo
	Los ingenieros escriben más código consistente como resultado del proceso de legibilidad.	Los ingenieros reciben retroalimentación y dirección consistentes en las revisiones de código por legibilidad revisores como parte del proceso de legibilidad.	Encuesta de legibilidad: Proporción de informes de ingenieros inconsistencia en los comentarios de los revisores de legibilidad y los criterios de legibilidad.
	Los ingenieros contribuyen a una cultura de salud del código como resultado del proceso de legibilidad.	Los ingenieros a los que se les ha otorgado legibilidad comentan regularmente sobre problemas de estilo y/o legibilidad en las revisiones de código.	Encuesta de legibilidad: proporción de ingenieros que informan que comentan regularmente sobre problemas de estilo o legibilidad en las revisiones de código
Atención de los ingenieros y/o intelectual	n / A	n / A	n / A
	Los ingenieros aprenden sobre el código base de Google y las mejores prácticas de codificación como resultado del proceso de legibilidad.	Los ingenieros informan haber aprendido del proceso de legibilidad.	Encuesta de legibilidad: Proporción de ingenieros que informaron que aprendieron sobre cuatro temas relevantes
	Los ingenieros reciben tutoría durante la legibilidad proceso.	Los ingenieros informan interacciones positivas con ingenieros experimentados de Google que actúan como revisores durante el proceso de legibilidad.	Encuesta de legibilidad: Proporción de ingenieros que informaron que trabajar con revisores de legibilidad fue una fortaleza del proceso de legibilidad
Tempo/velocidad	Los ingenieros son más productiva como resultado del proceso de legibilidad.	Los ingenieros a los que se les ha concedido legibilidad se consideran más productivos que los ingenieros a los que no se les ha concedido legibilidad.	Encuesta trimestral: Proporción de ingenieros que informan que son altamente productivos
		Los ingenieros informan que completar el proceso de legibilidad afecta positivamente su velocidad de ingeniería.	Encuesta de legibilidad: Proporción de ingenieros que informan que tener legibilidad reduce la velocidad de ingeniería del equipo
		Las listas de cambios (CL) escritas por ingenieros a los que se les ha otorgado la legibilidad son más rápidas de revisar que las CL escritas por ingenieros a los que no se les ha otorgado la legibilidad.	Datos de registros: tiempo medio de revisión de CL de autores con legibilidad y sin legibilidad

CUANTOS	Objetivo	Señal	Métrico
		Las CL escritas por ingenieros a los que se les ha otorgado la legibilidad son más fáciles de guiar a través de la revisión del código que las CL escritas por ingenieros a los que no se les ha otorgado la legibilidad.	Datos de registros: tiempo de pastoreo medio para CL de autores con legibilidad y sin legibilidad
		Las CL escritas por ingenieros a los que se les ha otorgado la legibilidad son más rápidas de revisar el código que las CL escritas por ingenieros a los que no se les ha otorgado la legibilidad.	Datos de registros: tiempo medio de envío de CL de autores con legibilidad y sin legibilidad
		El proceso de legibilidad no tiene un impacto negativo en la velocidad de ingeniería.	Encuesta de legibilidad: proporción de ingenieros que informan que el proceso de legibilidad afecta negativamente su velocidad
			Encuesta de legibilidad: Proporción de ingenieros que informan que los revisores de legibilidad respondió con prontitud
			Encuesta de legibilidad: proporción de ingenieros que informaron que la puntualidad de las revisiones fue una fortaleza del proceso de legibilidad
Satisfacción	Los ingenieros ven el beneficio del proceso de legibilidad y tienen sentimientos positivos acerca de participar en él.	Los ingenieros ven el proceso de legibilidad como una experiencia positiva en general.	Encuesta de legibilidad: proporción de ingenieros que informaron que su experiencia con el proceso de legibilidad fue positiva en general
		Los ingenieros consideran que el proceso de legibilidad merece la pena	Encuesta de legibilidad: Proporción de ingenieros que informan que el proceso de legibilidad es vale la pena
			Encuesta de legibilidad: Proporción de ingenieros que informan que la calidad de las revisiones de legibilidad es una fortaleza del proceso
			Encuesta de legibilidad: Proporción de ingenieros que informan que la minuciosidad es una fortaleza del proceso
		Los ingenieros no ven el proceso de legibilidad como frustrante.	Encuesta de legibilidad: proporción de ingenieros que informan que el proceso de legibilidad es incierto, poco claro, lento o frustrante

CUANTOS	Objetivo	Señal	Métrico
			Encuesta trimestral: Proporción de ingenieros que informan que están satisfechos con su propia velocidad de ingeniería

Tomar medidas y hacer un seguimiento de los resultados

Recuerde nuestro objetivo original en este capítulo: queremos tomar medidas y mejorar la productividad. Después de realizar una investigación sobre un tema, el equipo de Google siempre prepara una lista de recomendaciones sobre cómo podemos seguir mejorando. Podríamos sugerir nuevas funciones para una herramienta, mejorar la latencia de una herramienta, mejorar la documentación, eliminar procesos obsoletos o incluso cambiar las estructuras de incentivos para los ingenieros. Idealmente, estas recomendaciones son "impulsadas por herramientas": no sirve de nada decirles a los ingenieros que cambien su proceso o forma de pensar si las herramientas no los ayudan a hacerlo. En cambio, siempre asumimos que los ingenieros harán las compensaciones apropiadas si tienen los datos adecuados disponibles y las herramientas adecuadas a su disposición.

En cuanto a la legibilidad, nuestro estudio mostró que, en general, valió la pena: los ingenieros que lograron la legibilidad estaban satisfechos con el proceso y sintieron que aprendieron de él. Nuestros registros mostraron que también revisaron su código más rápido y lo enviaron más rápido, incluso teniendo en cuenta que ya no necesitaban tantos revisores. Nuestro estudio también mostró áreas de mejora en el proceso: los ingenieros identificaron puntos débiles que habrían hecho que el proceso fuera más rápido o más placentero. Los equipos de idiomas tomaron estas recomendaciones y mejoraron las herramientas y el proceso para hacerlo más rápido y más transparente para que los ingenieros tuvieran una experiencia más placentera.

Conclusión

En Google, descubrimos que dotar de personal a un equipo de especialistas en productividad de ingeniería tiene beneficios generalizados para la ingeniería de software; en lugar de confiar en que cada equipo trace su propio curso para aumentar la productividad, un equipo centralizado puede centrarse en soluciones de base amplia para problemas complejos. Dichos factores "basados en humanos" son notoriamente difíciles de medir, y es importante que los expertos comprendan los datos que se analizan dado que muchas de las compensaciones involucradas en los procesos de ingeniería cambiantes son difíciles de medir con precisión y, a menudo, tienen consecuencias no deseadas. . Tal equipo debe permanecer impulsado por los datos y apuntar a eliminar el sesgo subjetivo.

TL; DR

- Antes de medir la productividad, pregúntese si el resultado es procesable, independientemente de si el resultado es positivo o negativo. Si no puede hacer nada con el resultado, es probable que no valga la pena medirlo.

- Seleccionar métricas significativas utilizando el marco GSM. Una buena métrica es un indicador razonable de la señal que está tratando de medir y se puede rastrear hasta sus objetivos originales.
- Seleccione métricas que cubran todas las partes de la productividad (QUANTS). Al hacer esto, se asegura de no mejorar un aspecto de la productividad (como la velocidad del desarrollador) a costa de otro (como la calidad del código).
- ¡Las métricas cualitativas también son métricas! Considere tener un mecanismo de encuesta para rastrear métricas longitudinales sobre las creencias de los ingenieros. Las métricas cualitativas también deben alinearse con las métricas cuantitativas; si no lo hacen, es probable que las métricas cuantitativas sean incorrectas.
- Trate de crear recomendaciones integradas en el flujo de trabajo del desarrollador y las estructuras de incentivos. Aunque a veces es necesario recomendar capacitación o documentación adicional, es más probable que ocurra un cambio si se integra en los hábitos diarios del desarrollador.

PARTE III

Procesos

Guías de estilo y reglas

*Escrito por Shaindel Schwartz
Editado por Tom Mansreck*

La mayoría de las organizaciones de ingeniería tienen reglas que rigen sus bases de código: reglas sobre dónde se almacenan los archivos fuente, reglas sobre el formato del código, reglas sobre nombres y patrones, excepciones y subprocessos. La mayoría de los ingenieros de software trabajan dentro de los límites de un conjunto de políticas que controlan cómo funcionan. En Google, para administrar nuestra base de código, mantenemos un conjunto de guías de estilo que definen nuestras reglas.

Las reglas son leyes. No son solo sugerencias o recomendaciones, sino leyes estrictas y de obligado cumplimiento. Como tales, son universalmente exigibles: las reglas no pueden ignorarse excepto cuando se aprueben según la necesidad de uso. A diferencia de las reglas, la guía brinda recomendaciones y mejores prácticas. Estas partes son buenas para seguir, incluso muy recomendables, pero a diferencia de las reglas, por lo general tienen cierto margen de variación.

Recopilamos las reglas que definimos, lo que se debe y no se debe hacer al escribir código, en nuestras guías de estilo de programación, que se tratan como canon. "Estilo" puede ser un nombre un poco inapropiado aquí, lo que implica una colección limitada a las prácticas de formato. Nuestras guías de estilo son más que eso; son el conjunto completo de convenciones que rigen nuestro código. Eso no quiere decir que nuestras guías de estilo sean estrictamente prescriptivas; las reglas de la guía de estilo pueden requerir juicio, como la regla de usar nombres que son "**lo más descriptivo posible, dentro de lo razonable**." Más bien, nuestras guías de estilo sirven como la fuente definitiva de las reglas a las que se responsabiliza a nuestros ingenieros.

Mantenemos guías de estilo separadas para cada uno de los lenguajes de programación utilizados en Google.
¹En un alto nivel, todas las guías tienen objetivos similares, con el objetivo de dirigir el código

1 Muchas de nuestras guías de estilo tienen versiones externas, que puede encontrar en https://google.github.io/guia_de_estilo. Nosotros cite numerosos ejemplos de estas guías dentro de este capítulo.

desarrollo con miras a la sostenibilidad. Al mismo tiempo, hay mucha variación entre ellos en alcance, extensión y contenido. Los lenguajes de programación tienen diferentes fortalezas, diferentes características, diferentes prioridades y diferentes caminos históricos para la adopción dentro de los repositorios de código en constante evolución de Google. Por lo tanto, es mucho más práctico adaptar de forma independiente las pautas de cada idioma. Algunas de nuestras guías de estilo son concisas y se centran en algunos principios generales, como nombres y formato, como se demuestra en nuestras guías Dart, R y Shell. Otras guías de estilo incluyen muchos más detalles, profundizan en características específicas del lenguaje y se extienden a documentos mucho más extensos, en particular, nuestras guías de C++, Python y Java. Algunas guías de estilo destacan el uso típico del lenguaje que no es de Google: nuestra guía de estilo de Go es muy corta, **convenciones reconocidas externamente**. Otros incluyen reglas que difieren fundamentalmente de las normas externas; nuestras reglas de C++ no permiten el uso de excepciones, una función de lenguaje muy utilizada fuera del código de Google.

La amplia variación incluso entre nuestras propias guías de estilo hace que sea difícil precisar la descripción precisa de lo que debe cubrir una guía de estilo. Las decisiones que guían el desarrollo de las guías de estilo de Google surgen de la necesidad de mantener sostenible nuestra base de código. Las bases de código de otras organizaciones tendrán inherentemente diferentes requisitos de sostenibilidad que requieren un conjunto diferente de reglas adaptadas. Este capítulo analiza los principios y procesos que guían el desarrollo de nuestras reglas y orientación, y extrae ejemplos principalmente de las guías de estilo C++, Python y Java de Google.

¿Por qué tener reglas?

Entonces, ¿por qué tenemos reglas? El objetivo de tener reglas establecidas es fomentar el “buen” comportamiento y desalentar el “malo” comportamiento. La interpretación de “bueno” y “malo” varía según la organización, dependiendo de lo que le importa a la organización. Tales designaciones no son preferencias universales; bueno versus malo es subjetivo y se adapta a las necesidades. Para algunas organizaciones, “bueno” podría promover patrones de uso que admitan una pequeña huella de memoria o priorizar posibles optimizaciones de tiempo de ejecución. En otras organizaciones, “bueno” podría promover opciones que ejerzan nuevas características del lenguaje. A veces, una organización se preocupa más profundamente por la consistencia, por lo que cualquier cosa que no sea consistente con los patrones existentes es “mala”. Primero debemos reconocer lo que una determinada organización valora; usamos reglas y orientación para alentar y desalentar el comportamiento en consecuencia.

A medida que crece una organización, las reglas y pautas establecidas dan forma al vocabulario común de codificación. Un vocabulario común permite a los ingenieros concentrarse en lo que su código necesita decir en lugar de cómo lo dicen. Al dar forma a este vocabulario, los ingenieros tenderán a hacer las cosas “buenas” por defecto, incluso de manera subconsciente. Por lo tanto, las reglas nos brindan una amplia influencia para empujar los patrones de desarrollo comunes en las direcciones deseadas.

Crear las reglas

Al definir un conjunto de reglas, la pregunta clave no es "¿Qué reglas deberíamos tener?" La pregunta que debe hacerse es: "¿Qué objetivo estamos tratando de avanzar?" Cuando nos enfocamos en el objetivo al que servirán las reglas, identificar qué reglas respaldan este objetivo hace que sea más fácil destilar el conjunto de reglas útiles. En Google, donde la guía de estilo sirve como ley para las prácticas de codificación, no preguntamos: "¿Qué incluye la guía de estilo?" sino más bien, "¿Por qué algo va en la guía de estilo?" ¿Qué gana nuestra organización al tener un conjunto de reglas para regular la escritura de código?

Principios rectores

Pongamos las cosas en contexto: la organización de ingeniería de Google está compuesta por más de 30.000 ingenieros. Esa población de ingenieros exhibe una variación salvaje en habilidades y antecedentes. Cada día se realizan alrededor de 60 000 envíos a una base de código de más de dos mil millones de líneas de código que probablemente existirá durante décadas. Estamos optimizando para un conjunto diferente de valores que la mayoría de las otras organizaciones necesitan, pero hasta cierto punto, estas preocupaciones son omnipresentes: necesitamos mantener un entorno de ingeniería que sea resistente tanto a la escala como al tiempo.

En este contexto, el objetivo de nuestras reglas es administrar la complejidad de nuestro entorno de desarrollo, manteniendo la base de código manejable y permitiendo que los ingenieros trabajen de manera productiva. Estamos haciendo una compensación aquí: el gran cuerpo de reglas que nos ayuda a alcanzar este objetivo significa que estamos restringiendo las opciones. Perdemos algo de flexibilidad e incluso podemos ofender a algunas personas, pero las ganancias de consistencia y la reducción del conflicto proporcionadas por un estándar autoritativo ganan.

Teniendo en cuenta este punto de vista, reconocemos una serie de principios generales que guían el desarrollo de nuestras reglas, que deben:

- Jalar su peso
- Optimizar para el lector
- Se consistente
- Evite construcciones sorprendentes y propensas a errores
- Ceder a los aspectos prácticos cuando sea necesario

Las reglas deben tirar de su peso

No todo debe ir en una guía de estilo. Hay un costo distinto de cero al pedir a todos los ingenieros de una organización que aprendan y se adapten a cualquier regla nueva que se establezca. Con demasiadas reglas,² no solo será más difícil para los ingenieros recordar todas las reglas relevantes mientras escriben su código, sino que también será más difícil para los nuevos ingenieros aprender su camino. Más reglas también hacen que sea más desafiante y más costoso mantener el conjunto de reglas.

Con este fin, elegimos deliberadamente no incluir reglas que se espera que sean evidentes. La guía de estilo de Google no está destinada a ser interpretada de manera legal; el hecho de que algo no esté explícitamente prohibido no implica que sea legal. Por ejemplo, la guía de estilo de C++ no tiene ninguna regla contra el uso de `deir a`. Los programadores de C++ ya tienden a evitarlo, por lo que incluir una regla explícita que lo prohíba generaría una sobrecarga innecesaria. Si solo uno o dos ingenieros se equivocan en algo, agregar a la carga mental de todos mediante la creación de nuevas reglas no escala.

Optimizar para el lector

Otro principio de nuestras reglas es optimizar para el lector del código en lugar del autor. Dado el paso del tiempo, nuestro código se leerá con mucha más frecuencia de lo que se escribe. Preferimos que el código sea tedioso de escribir que difícil de leer. En nuestra guía de estilo de Python, cuando hablamos de expresiones condicionales, reconocemos que son más cortas que si declaraciones y por lo tanto más conveniente para los autores de código. Sin embargo, debido a que tienden a ser más difíciles de entender para los lectores que los más detallados si declaraciones, **restringimos su uso**. Valoramos "simple de leer" sobre "simple de escribir". Estamos haciendo una compensación aquí: puede costar más por adelantado cuando los ingenieros deben escribir repetidamente nombres descriptivos potencialmente más largos para variables y tipos. Elegimos pagar este costo por la legibilidad que brinda a todos los futuros lectores.

Como parte de esta priorización, también exigimos que los ingenieros dejen evidencia explícita del comportamiento previsto en su código. Queremos que los lectores entiendan claramente lo que hace el código mientras lo leen. Por ejemplo, nuestras guías de estilo de Java, JavaScript y C++ exigen el uso de la anotación o palabra clave `override` cada vez que un método reemplaza un método de superclase. Sin la evidencia explícita del diseño en el lugar, es probable que los lectores puedan descifrar esta intención, aunque se necesitaría un poco más de investigación por parte de cada lector para trabajar en el código.

² Las herramientas importan aquí. La medida de "demasiadas" no es el número bruto de reglas en juego, sino cuántas ingenieros necesitan recordar. Por ejemplo, en el formato pre-clang de los viejos tiempos, necesitábamos recordar un montón de reglas de formato. Esas reglas no han desaparecido, pero con nuestras herramientas actuales, el costo de cumplimiento se ha reducido drásticamente. Hemos llegado a un punto en el que alguien podría agregar un número arbitrario de reglas de formato y a nadie le importaría, porque la herramienta lo hace por usted.

La evidencia del comportamiento previsto se vuelve aún más importante cuando podría ser sorprendente. En C++, a veces es difícil rastrear la propiedad de un puntero simplemente leyendo un fragmento de código. Si se pasa un puntero a una función, sin estar familiarizados con el comportamiento de la función, no podemos estar seguros de qué esperar. ¿La persona que llama todavía posee el puntero? ¿La función tomó posesión? ¿Puedo continuar usando el puntero después de que la función regrese o podría haberse eliminado? Para evitar este problema, nuestro [La guía de estilo de C++ prefiere el uso deestándar::único_ptr](#)cuando se pretenda transferir la propiedad.único_ptr es una construcción que administra la propiedad del puntero, lo que garantiza que solo exista una copia del puntero. Cuando una función toma unúnico_ptrcomo argumento y tiene la intención de tomar posesión del puntero, las personas que llaman deben invocar explícitamente la semántica de movimiento:

```
// Función que toma un Foo* y puede o no asumir la propiedad // del puntero
```

```
pasado.
```

```
vacio
```

```
tomarfoo(Foo*argumento);
```

```
// Las llamadas a la función no le dicen nada al lector sobre qué esperar // con respecto a la propiedad después de que la función regrese.
```

```
Foo*mi_foo(NuevoFoo());
```

```
tomarfoo(mi_foo);
```

Compare esto con lo siguiente:

```
// Función que toma un std::unique_ptr<Foo>. vacío
```

```
tomarfoo(estándar::único_ptr<Foo>argumento);
```

```
// Cualquier llamada a la función muestra explícitamente que se // otorga la propiedad y que el unique_ptr no se puede usar después de que la función // regrese.
```

```
estándar::único_ptr<Foo>mi_foo(FooFactory());
```

```
tomarfoo(estándar::moverse(mi_foo));
```

Dada la regla de la guía de estilo, garantizamos que todos los sitios de llamadas incluirán evidencia clara de transferencia de propiedad siempre que corresponda. Con esta señal en su lugar, los lectores del código no necesitan comprender el comportamiento de cada llamada de función. Proporcionamos suficiente información en la API para razonar sobre sus interacciones. Esta documentación clara del comportamiento en los sitios de llamadas garantiza que los fragmentos de código sigan siendo legibles y comprensibles. Nuestro objetivo es el razonamiento local, donde el objetivo es una comprensión clara de lo que sucede en el sitio de la llamada sin necesidad de buscar y hacer referencia a otro código, incluida la implementación de la función.

La mayoría de las reglas de la guía de estilo que cubren los comentarios también están diseñadas para respaldar este objetivo de evidencia en el lugar para los lectores. Los comentarios de documentación (los comentarios de bloque antepuestos a un archivo, clase o función determinados) describen el diseño o la intención del código que sigue. Los comentarios de implementación (los comentarios intercalados a lo largo del código mismo) justifican o resaltan opciones no obvias, explican partes engañosas y subrayan partes importantes del código. Tenemos reglas de guía de estilo que cubren ambos tipos de

comentarios, lo que requiere que los ingenieros proporcionen las explicaciones que otro ingeniero podría estar buscando al leer el código.

Se consistente

Nuestra opinión sobre la coherencia dentro de nuestro código base es similar a la filosofía que aplicamos en nuestras oficinas de Google. Con una gran población de ingeniería distribuida, los equipos se dividen con frecuencia entre las oficinas y los Googlers a menudo se encuentran viajando a otros sitios. Aunque cada oficina mantiene su personalidad única, adoptando el sabor y el estilo local, para cualquier cosa necesaria para hacer el trabajo, las cosas se mantienen deliberadamente igual. La insignia de Googler visitante funcionará con todos los lectores de insignias locales; cualquier dispositivo de Google siempre obtendrá WiFi; la configuración de videoconferencia en cualquier sala de conferencias tendrá la misma interfaz. Un Googler no necesita perder tiempo aprendiendo cómo configurar todo esto; saben que será lo mismo sin importar dónde estén. Es fácil moverse entre oficinas y seguir trabajando.

Eso es lo que buscamos con nuestro código fuente. La consistencia es lo que permite a cualquier ingeniero saltar a una parte desconocida de la base de código y ponerse a trabajar con bastante rapidez. Un proyecto local puede tener su personalidad única, pero sus herramientas son las mismas, sus técnicas son las mismas, sus bibliotecas son las mismas y todo simplemente funciona.

Ventajas de la consistencia

Aunque puede parecer restrictivo que una oficina no pueda personalizar un lector de credenciales o una interfaz de videoconferencia, los beneficios de la coherencia superan con creces la libertad creativa que perdemos. Ocurre lo mismo con el código: ser consistente puede parecer restrictivo a veces, pero significa que más ingenieros realizan más trabajo con menos esfuerzo:³

- Cuando una base de código es internamente consistente en su estilo y normas, los ingenieros que escriben el código y otros que lo leen pueden enfocarse en lo que se está haciendo en lugar de cómo se presenta. En gran medida, esta consistencia permite una fragmentación experta.⁴ Cuando resolvemos nuestros problemas con las mismas interfaces y formateamos el código de manera consistente, es más fácil para los expertos echar un vistazo a algún código, concentrarse en lo que es importante y comprender lo que está haciendo. También facilita modularizar el código y detectar la duplicación. Por estas razones, enfocamos mucha atención en convenciones de nomenclatura consistentes, uso consistente de patrones comunes y formato y estructura consistentes. También hay muchas reglas que proponen una decisión sobre un tema aparentemente pequeño únicamente para garantizar que las cosas se hagan de una sola manera. Para

³ Crédito a H. Wright por la comparación del mundo real, realizada en el momento de haber visitado alrededor de 15 diferentes oficinas de Google.

⁴ La "fragmentación" es un proceso cognitivo que agrupa piezas de información en "fragmentos" significativos en lugar de que tomar nota de ellos individualmente. Los ajedrecistas expertos, por ejemplo, piensan en configuraciones de piezas más que en las posiciones de los individuos.

por ejemplo, elija la cantidad de espacios que se usarán para la sangría o el límite establecido en la longitud de la línea.⁵ Es la consistencia de tener una respuesta en lugar de la respuesta en sí misma lo que es la parte valiosa aquí.

- La consistencia permite escalar. Las herramientas son clave para que una organización escale, y el código coherente facilita la creación de herramientas que puedan comprender, editar y generar código. Los beneficios completos de las herramientas que dependen de la uniformidad no se pueden aplicar si todos tienen pequeños bolsillos de código que difieren: si una herramienta puede mantener los archivos fuente actualizados agregando importaciones faltantes o eliminando las inclusiones no utilizadas, si diferentes proyectos eligen diferentes estrategias de clasificación. para sus listas de importación, es posible que la herramienta no funcione en todas partes. Cuando todos usan los mismos componentes y cuando el código de todos sigue las mismas reglas de estructura y organización, podemos invertir en herramientas que funcionen en todas partes, incorporando automatización para muchas de nuestras tareas de mantenimiento. Si cada equipo necesitara invertir por separado en una versión personalizada de la misma herramienta,
- La coherencia también ayuda a escalar la parte humana de una organización. A medida que crece una organización, aumenta la cantidad de ingenieros que trabajan en el código base. Mantener el código en el que todos están trabajando lo más consistente posible permite una mejor movilidad entre proyectos, minimizando el tiempo de aceleración para que un ingeniero cambie de equipo y construyendo la capacidad de la organización para flexibilizarse y adaptarse a medida que fluctúan las necesidades de personal. Una organización en crecimiento también significa que las personas en otros roles interactúan con el código: SRE, ingenieros de bibliotecas y conserjes de código, por ejemplo. En Google, estos roles a menudo abarcan varios proyectos, lo que significa que los ingenieros que no están familiarizados con el proyecto de un equipo determinado pueden comenzar a trabajar en el código de ese proyecto. Una experiencia consistente en toda la base de código hace que esto sea eficiente.
- La consistencia también asegura la resiliencia en el tiempo. A medida que pasa el tiempo, los ingenieros abandonan los proyectos, se incorporan nuevas personas, cambia la propiedad y los proyectos se fusionan o dividen. Esforzarse por una base de código coherente garantiza que estas transiciones sean de bajo costo y nos permite una fluidez casi ilimitada tanto para el código como para los ingenieros que trabajan en él, lo que simplifica los procesos necesarios para el mantenimiento a largo plazo.

⁵ Ver 4.2 Sangría de bloque: +2 espacios, Espacios frente a pestañas, 4.4 Límite de columna: 100yLongitud de la línea.

A escala

Hace unos años, nuestra guía de estilo de C++ prometía casi nunca cambiar las reglas de la guía de estilo que harían que el código antiguo fuera inconsistente: "En algunos casos, puede haber buenos argumentos para cambiar ciertas reglas de estilo, pero de todos modos mantenemos las cosas tal como están en orden. para preservar la consistencia."

Cuando el código base era más pequeño y había menos rincones viejos y polvorrientos, eso tenía sentido.

Cuando el código base se hizo más grande y más viejo, eso dejó de ser una prioridad. Esto fue (al menos para los árbitros detrás de nuestra guía de estilo de C++) un cambio consciente: al tocar este bit, estábamos declarando explícitamente que el código base de C++ nunca volvería a ser completamente consistente, ni siquiera apuntábamos a eso.

Simplemente sería una carga demasiado pesada no solo actualizar las reglas a las mejores prácticas actuales, sino también exigir que apliquemos esas reglas a todo lo que se haya escrito. Nuestras herramientas y procesos de cambio a gran escala nos permiten actualizar casi todo nuestro código para seguir casi todos los patrones o sintaxis nuevos, de modo que la mayoría del código antiguo muestre el estilo aprobado más reciente ([ver capítulo 22](#)). Sin embargo, tales mecanismos no son perfectos; cuando el código base se vuelve tan grande, no podemos estar seguros de que todo el código antiguo se ajuste a las nuevas mejores prácticas. Requerir una consistencia perfecta ha llegado al punto en que hay demasiado costo por el valor.

Estableciendo el estándar. Cuando abogamos por la coherencia, tendemos a centrarnos en la coherencia interna. A veces, las convenciones locales surgen antes de que se adopten las globales, y no es razonable ajustar todo para que coincida. En ese caso, defendemos una jerarquía de consistencia: "Ser consistente" comienza localmente, donde las normas dentro de un archivo dado preceden a las de un equipo dado, que preceden a las del proyecto más grande, que preceden a las del código base general. De hecho, las guías de estilo contienen una serie de reglas que difieren explícitamente de las convenciones locales,⁶ valorando esta coherencia local por encima de una elección científico-técnica.

Sin embargo, no siempre es suficiente para una organización crear y adherirse a un conjunto de convenciones internas. A veces, se deben tener en cuenta los estándares adoptados por la comunidad externa.

⁶uso de constante, por ejemplo.

Contar espacios

La guía de estilo de Python en Google inicialmente exigía sangrías de dos espacios para todo nuestro código de Python. La guía de estilo estándar de Python, utilizada por la comunidad externa de Python, utiliza sangrías de cuatro espacios. La mayor parte de nuestro desarrollo inicial de Python fue en apoyo directo de nuestros proyectos de C++, no para aplicaciones de Python reales. Por lo tanto, elegimos usar una sangría de dos espacios para ser coherentes con nuestro código C++, que ya estaba formateado de esa manera. Con el paso del tiempo, vimos que este razonamiento realmente no se sosténía. Los ingenieros que escriben código Python leen y escriben otro código Python con mucha más frecuencia que leen y escriben código C++. Les estábamos costando a nuestros ingenieros un esfuerzo adicional cada vez que necesitaban buscar algo o hacer referencia a fragmentos de código externos. También sufrímos mucho cada vez que tratábamos de exportar partes de nuestro código a código abierto,

Cuando llegó el momento de [dealondra](#)(un lenguaje basado en Python diseñado en Google para servir como lenguaje de descripción de compilación) para tener su propia guía de estilo, decidimos cambiar y usar sangrías de cuatro espacios para ser coherentes con el mundo exterior.⁷

Si ya existen convenciones, generalmente es una buena idea que una organización sea consistente con el mundo exterior. Para esfuerzos pequeños, autónomos y de corta duración, probablemente no hará una diferencia; la consistencia interna es más importante que cualquier cosa que suceda fuera del alcance limitado del proyecto. Una vez que el paso del tiempo y el escalamiento potencial se convierten en factores, aumenta la probabilidad de que su código interactúe con proyectos externos o incluso termine en el mundo exterior. Mirando a largo plazo, adherirse al estándar ampliamente aceptado probablemente valdrá la pena.

Evite construcciones sorprendentes y propensas a errores

Nuestras guías de estilo restringen el uso de algunas de las construcciones más sorprendentes, inusuales o complicadas en los lenguajes que usamos. Las características complejas a menudo tienen trampas sutiles que no son obvias a primera vista. El uso de estas funciones sin comprender a fondo sus complejidades facilita el mal uso de ellas y la introducción de errores. Incluso si los ingenieros de un proyecto entienden bien una construcción, no se garantiza que los futuros miembros y mantenedores del proyecto tengan la misma comprensión.

Este razonamiento está detrás de nuestra decisión de la guía de estilo de Python para evitar el uso [características de poder](#) como la reflexión. Las funciones reflexivas de Python `hasattr()` y `getattr()` permitir que un usuario acceda a los atributos de los objetos usando cadenas:

⁷ El formato de estilo para los archivos BUILD implementados con Starlark se aplica mediante el constructor. Ver https://github.com/bazelbuild/construir_herramientas.

```
sihasattr(mi_objeto,'foo'): alguna_var= obtener(mi_objeto,'foo')
```

Ahora, con ese ejemplo, todo podría parecer bien. Pero considera esto:

algún_archivo.py.

```
UNA CONSTANTE=[  
    'foo',  
    'bar',  
    'baz',  
]
```

otro_archivo.py.

```
valores=[]  
porcampoenalgún_archivo.UNA CONSTANTE:  
    valores.adjuntar(obtener(mi_objeto,campo))
```

Al buscar a través del código, ¿cómo sabe que los campos `comida`, `bar`, `ybaz` se accede aquí? No queda evidencia clara para el lector. No ve fácilmente y, por lo tanto, no puede validar fácilmente qué cadenas se utilizan para acceder a los atributos de su objeto. ¿Qué pasa si, en lugar de leer esos valores de `UNA CONSTANTE`, los leemos de un mensaje de solicitud de llamada a procedimiento remoto (RPC) o de un almacén de datos? Tal código ofuscado podría causar una gran falla de seguridad, que sería muy difícil de notar, simplemente por validar el mensaje incorrectamente. También es difícil probar y verificar dicho código.

La naturaleza dinámica de Python permite tal comportamiento, y en circunstancias muy limitadas, usando `hasattr()`/`getattr()` es válida. En la mayoría de los casos, sin embargo, solo causan confusión e introducen errores.

Si bien estas características avanzadas del lenguaje podrían resolver perfectamente un problema para un experto que sabe cómo aprovecharlas, las características avanzadas suelen ser más difíciles de entender y no se utilizan mucho. Necesitamos que todos nuestros ingenieros puedan operar en el código base, no solo los expertos. No es solo soporte para el ingeniero de software novato, sino que también es un mejor entorno para los SRE: si un SRE está depurando una interrupción de la producción, accederá a cualquier fragmento de código sospechoso, incluso código escrito en un lenguaje en el que se encuentran. no es fluido. Damos mayor valor al código simplificado y directo que es más fácil de entender y mantener.

Conceder a los aspectos prácticos

En palabras de Ralph Waldo Emerson: “**Una consistencia tonta es el duende de las mentes pequeñas..**” En nuestra búsqueda de un código base consistente y simplificado, no queremos ignorar ciegamente todo lo demás. Sabemos que algunas de las reglas de nuestras guías de estilo encontrarán casos que justifiquen excepciones, y eso está bien. Cuando es necesario, permitimos concesiones a optimizaciones y aspectos prácticos que, de lo contrario, podrían entrar en conflicto con nuestras reglas.

El rendimiento importa. A veces, incluso si eso significa sacrificar la consistencia o la legibilidad, tiene sentido adaptarse a las optimizaciones de rendimiento. Por ejemplo, aunque nuestra guía de estilo de C++ prohíbe el uso de excepciones, incluye una regla que permite el uso de **no excepto**, un especificador de lenguaje relacionado con excepciones que puede desencadenar optimizaciones del compilador.

La interoperabilidad también importa. El código que está diseñado para funcionar con piezas específicas que no son de Google podría funcionar mejor si se adapta a su objetivo. Por ejemplo, nuestra guía de estilo de C++ incluye una excepción a la pauta de nomenclatura general de CamelCase que permite el uso del estilo `snake_case` de la biblioteca estándar para entidades que imitan las características de la biblioteca estándar.⁸ La guía de estilo de C++ también permite **excepciones para la programación de Windows**, donde la compatibilidad con las funciones de la plataforma requiere herencia múltiple, algo explícitamente prohibido para todos los demás códigos de C++. Nuestras guías de estilo de Java y JavaScript establecen explícitamente que el código generado, que con frecuencia interactúa con componentes que no pertenecen a un proyecto o depende de ellos, está fuera del alcance de las reglas de la guía.⁹

La consistencia es vital; la adaptación es clave.

La guía de estilo

Entonces, ¿qué incluye una guía de estilo de idioma? Hay aproximadamente tres categorías en las que caen todas las reglas de la guía de estilo:

- Reglas para evitar peligros
- Reglas para hacer cumplir las mejores prácticas
- Reglas para asegurar la consistencia

Evitando el peligro

En primer lugar, nuestras guías de estilo incluyen reglas sobre las características del idioma que deben o no deben realizarse por razones técnicas. Tenemos reglas sobre cómo usar variables y miembros estáticos; reglas sobre el uso de expresiones lambda; reglas sobre el manejo de excepciones; reglas sobre la creación de subprocesos, control de acceso y herencia de clases. Cubrimos qué características del lenguaje usar y qué construcciones evitar. Mencionamos los tipos de vocabulario estándar que se pueden usar y con qué fines. Específicamente, incluimos reglas sobre lo difícil de usar y lo difícil de usar correctamente: algunas características del idioma tienen patrones de uso matizados que pueden no ser intuitivos o fáciles de aplicar.

⁸ Ver [Excepciones a las reglas de nomenclatura](#). Como ejemplo, nuestras bibliotecas de Abseil de código abierto utilizan la nomenclatura `snake_case` para tipos destinados a ser reemplazos de los tipos estándar. Ver los tipos definidos en <https://github.com/abseil/abseilcpp/blob/master/absl/utility/utility.h>. Estos son la implementación de C++ 11 de los tipos estándar de C++ 14 y, por lo tanto, utilizan el estilo `snake_case` favorito del estándar en lugar del formato CamelCase preferido de Google.

⁹ Ver [Código generado: en su mayoría exento](#).

correctamente, lo que provoca la aparición de errores sutiles. Para cada decisión de la guía, nuestro objetivo es incluir los pros y los contras que se sopesaron con una explicación de la decisión a la que se llegó. La mayoría de estas decisiones se basan en la necesidad de resiliencia en el tiempo, apoyando y fomentando el uso sostenible del lenguaje.

Hacer cumplir las mejores prácticas

Nuestras guías de estilo también incluyen reglas que imponen algunas de las mejores prácticas para escribir código fuente. Estas reglas ayudan a mantener la base de código saludable y mantenible. Por ejemplo, especificamos dónde y cómo los autores del código deben incluir comentarios.¹⁰ Nuestras reglas para comentarios cubren las convenciones generales para comentar y se extienden para incluir casos específicos que deben incluir documentación en el código, casos en los que la intención no siempre es obvia, como fallas en declaraciones de cambio, bloques de captura de excepción vacíos, y metaprogramación de plantillas. También tenemos reglas que detallan la estructuración de los archivos fuente, delineando la organización del contenido esperado. Tenemos reglas sobre la denominación: denominación de paquetes, de clases, de funciones, de variables. Todas estas reglas están destinadas a guiar a los ingenieros hacia prácticas que respalden un código más saludable y sostenible.

Algunas de las mejores prácticas aplicadas por nuestras guías de estilo están diseñadas para hacer que el código fuente sea más legible. Muchas reglas de formato entran en esta categoría. Nuestras guías de estilo especifican cuándo y cómo usar espacios en blanco verticales y horizontales para mejorar la legibilidad. También cubren los límites de longitud de línea y la alineación de tornapuntas. Para algunos idiomas, cubrimos los requisitos de formato defiriendo las herramientas de formato automático `gofmt` para Go, `dartfmt` para Dart. Al detallar una lista detallada de requisitos de formato o nombrar una herramienta que se debe aplicar, el objetivo es el mismo: tenemos un conjunto consistente de reglas de formato diseñadas para mejorar la legibilidad que aplicamos a todo nuestro código.

Nuestras guías de estilo también incluyen limitaciones en características de lenguaje nuevas y aún no bien entendidas. El objetivo es instalar de forma preventiva vallas de seguridad alrededor de los peligros potenciales de una función mientras todos pasamos por el proceso de aprendizaje. Al mismo tiempo, antes de que todos empiecen a correr, limitar el uso nos da la oportunidad de observar los patrones de uso que se desarrollan y extraer las mejores prácticas de los ejemplos que observamos. Para estas nuevas características, al principio, a veces no estamos seguros de la orientación adecuada para dar. A medida que se extiende la adopción, los ingenieros que desean usar las nuevas funciones de diferentes maneras discuten sus ejemplos con los propietarios de la guía de estilo y solicitan concesiones para permitir casos de uso adicionales más allá de los cubiertos por las restricciones iniciales. Al ver las solicitudes de exención que llegan, tenemos una idea de cómo se está utilizando la función y, finalmente, recopilamos suficientes ejemplos para generalizar las buenas prácticas de las malas. Después de que nosotros

10 Ver <https://google.github.io/styleguide/cppguide.html#Comentarios>, <http://google.github.io/styleguide/pyguide#38-comentarios-y-docstrings>, y <https://google.github.io/styleguide/javaguide.html#s7-javadoc>, donde varios idiomas definen reglas generales de comentarios.

tenemos esa información, podemos regresar a la regla restrictiva y enmendarla para permitir un uso más amplio.

Estudio de caso: Presentación de std::unique_ptr

Cuando se introdujo C++11 est^ándar::único_ptr, un tipo de puntero inteligente que expresa la propiedad exclusiva de un objeto asignado dinámicamente y elimina el objeto cuando el único_ptr sale del alcance, nuestra guía de estilo inicialmente no permitió su uso. El comportamiento de los único_ptr no era familiar para la mayoría de los ingenieros, y la semántica de movimiento relacionada que introdujo el lenguaje era muy nueva y, para la mayoría de los ingenieros, muy confusa. Evitar la introducción de est<á>ndar::único_ptr en el código base parecía la opción más segura. Actualizamos nuestras herramientas para capturar referencias al tipo no permitido y mantuvimos nuestra guía existente recomendando otros tipos de punteros inteligentes existentes.

Pasó el tiempo. Los ingenieros tuvieron la oportunidad de adaptarse a las implicaciones de la semántica de movimientos y nos convencimos cada vez más de que usare^ándar::único_ptr estaba directamente en línea con los objetivos de nuestra guía de estilo. La información sobre la propiedad del objeto que un est<á>ndar::único_ptr facilita en un sitio de llamada de función hace que sea mucho más fácil para un lector entender ese código. La complejidad añadida de introducir este nuevo tipo y la novedosa semántica de movimiento que viene con él seguía siendo una gran preocupación, pero la mejora significativa en el estado general a largo plazo de la base de código hizo que la adopción de est<á>ndar::único_ptr una compensación que vale la pena.

Construyendo en consistencia

Nuestras guías de estilo también contienen reglas que cubren muchas de las cosas más pequeñas. Para estas reglas, tomamos y documentamos una decisión principalmente para tomar y documentar una decisión. Muchas reglas de esta categoría no tienen un impacto técnico significativo. Cosas como convenciones de nomenclatura, espaciado de sangría, orden de importación: por lo general, no hay un beneficio técnico claro y mensurable para una forma sobre otra, lo que podría ser la razón por la cual la comunidad técnica tiende a seguir debatiéndolas.¹¹ Al elegir uno, nos salimos del interminable ciclo de debate y podemos seguir adelante. Nuestros ingenieros ya no pierden tiempo discutiendo dos espacios versus cuatro. Lo importante para esta categoría de reglas no es qué hemos elegido para una regla dada tanto como el hecho de que *tenerelegido*.

Y por todo lo demás...

Con todo eso, hay muchas cosas que no están en nuestras guías de estilo. Intentamos centrarnos en las cosas que tienen el mayor impacto en el estado de nuestro código base. Estos documentos no especifican absolutamente las mejores prácticas, incluidas muchas piezas fundamentales de buenos consejos de ingeniería: no sea inteligente, no bifurque la base de código, no reinvente el

¹¹ Tales discusiones son realmente solo [cobertizo para bicicletas](#), una ilustración de Ley de la trivialidad de Parkinson.

rueda, y así sucesivamente. Documentos como nuestras guías de estilo no pueden servir para llevar a un novato completo hasta una comprensión de nivel maestro de la ingeniería de software; hay algunas cosas que asumimos, y esto es intencional.

Cambiar las reglas

Nuestras guías de estilo no son estáticas. Como ocurre con la mayoría de las cosas, dado el paso del tiempo, es probable que cambie el panorama dentro del cual se tomó una decisión sobre la guía de estilo y los factores que guiaron una decisión dada. A veces, las condiciones cambian lo suficiente como para justificar una reevaluación. Si se lanza una nueva versión de idioma, es posible que deseemos actualizar nuestras reglas para permitir o excluir nuevas funciones y modismos. Si una regla hace que los ingenieros inviertan esfuerzos para eludirla, es posible que debamos volver a examinar los beneficios que se suponía que proporcionaría la regla. Si las herramientas que usamos para hacer cumplir una regla se vuelven demasiado complejas y onerosas de mantener, es posible que la regla en sí se haya deteriorado y deba revisarse. Darse cuenta de cuándo una regla está lista para otro vistazo es una parte importante del proceso que mantiene nuestro conjunto de reglas relevante y actualizado.

Las decisiones detrás de las reglas capturadas en nuestras guías de estilo están respaldadas por evidencia. Al agregar una regla, dedicamos tiempo a discutir y analizar las ventajas y desventajas relevantes, así como las posibles consecuencias, tratando de verificar que un cambio determinado sea apropiado para la escala en la que opera Google. La mayoría de las entradas en las guías de estilo de Google incluyen estas consideraciones, presentando los pros y los contras que se sopesaron durante el proceso y brindando el razonamiento para la decisión final. Idealmente, priorizamos este razonamiento detallado y lo incluimos con cada regla.

Documentar el razonamiento detrás de una decisión determinada nos brinda la ventaja de poder reconocer cuándo es necesario cambiar las cosas. Dado el paso del tiempo y las condiciones cambiantes, una buena decisión tomada anteriormente puede no ser la mejor actual. Con los factores influyentes claramente señalados, podemos identificar cuándo los cambios relacionados con uno o más de estos factores justifican la reevaluación de la regla.

Estudio de caso: CamelCase Naming

En Google, cuando definimos nuestra guía de estilo inicial para el código de Python, elegimos usar el estilo de nomenclatura CamelCase en lugar del estilo de nomenclatura `snake_case` para los nombres de métodos. Aunque la guía de estilo pública de Python ([PEP 8](#)) y la mayor parte de la comunidad de Python usaba nombres de serpientes, la mayor parte del uso de Python de Google en ese momento era para desarrolladores de C++ que usaban Python como una capa de secuencias de comandos sobre una base de código C++. Muchos de los tipos de Python definidos eran envoltorios para los tipos de C++ correspondientes, y debido a que las convenciones de nomenclatura de C++ de Google siguen el estilo CamelCase, la coherencia entre idiomas se consideró clave.

Más tarde, llegamos a un punto en el que estábamos creando y admitiendo aplicaciones de Python independientes. Los ingenieros que usaron Python con mayor frecuencia fueron los ingenieros de Python.

Neers desarrollando proyectos de Python, no ingenieros de C++ que elaboran un script rápido. Estábamos causando un grado de incomodidad y problemas de legibilidad para nuestros ingenieros de Python, requiriendo que mantuvieran un estándar para nuestro código interno pero que se ajustaran constantemente a otro estándar cada vez que hacían referencia a código externo. También estábamos dificultando que los nuevos empleados que llegaron con experiencia en Python se adaptaran a nuestras normas de base de código.

A medida que crecían nuestros proyectos de Python, nuestro código interactuaba con mayor frecuencia con proyectos de Python externos. Estábamos incorporando bibliotecas de Python de terceros para algunos de nuestros proyectos, lo que llevó a una combinación dentro de nuestra base de código de nuestro propio formato CamelCase con el estilo snake_case preferido externamente. A medida que comenzamos a abrir el código fuente de algunos de nuestros proyectos de Python, mantenerlos en un mundo externo donde nuestras convenciones eran inconformistas agregó complejidad de nuestra parte y cautela de una comunidad que encontró nuestro estilo sorprendente y algo extraño.

Presentado con estos argumentos, después de discutir tanto los costos (perdiendo consistencia con otro código de Google, reeducación para Googlers acostumbrados a nuestro estilo Python) como los beneficios (ganando consistencia con la mayoría del otro código de Python, permitiendo que lo que ya se estaba filtrando con terceros). bibliotecas), los árbitros de estilo de la guía de estilo de Python decidieron cambiar la regla. Con la restricción de que se aplique como una opción para todo el archivo, una exención para el código existente y la libertad de los proyectos para decidir qué es lo mejor para ellos, la guía de estilo de Google Python se actualizó para permitir la nomenclatura de snake_case.

El proceso

Reconociendo que las cosas tendrán que cambiar, dada la larga vida útil y la capacidad de escalar que buscamos, creamos un proceso para actualizar nuestras reglas. El proceso para cambiar nuestra guía de estilo se basa en soluciones. Las propuestas de actualización de la guía de estilo se enmarcan en esta perspectiva, identificando un problema existente y presentando el cambio propuesto como una forma de solucionarlo. Los “problemas”, en este proceso, no son ejemplos hipotéticos de cosas que podrían salir mal; los problemas se prueban con patrones que se encuentran en el código existente de Google. Dado un problema demostrado, debido a que tenemos el razonamiento detallado detrás de la decisión de la guía de estilo existente, podemos reevaluar, verificando si una conclusión diferente ahora tiene más sentido.

La comunidad de ingenieros que escribe código regido por la guía de estilo suele estar mejor posicionada para darse cuenta de cuándo es necesario cambiar una regla. De hecho, aquí en Google, la mayoría de los cambios en nuestras guías de estilo comienzan con la discusión de la comunidad. Cualquier ingeniero puede hacer preguntas o proponer un cambio, generalmente comenzando con las listas de correo específicas del idioma dedicadas a las discusiones de la guía de estilo.

Las propuestas de cambios en la guía de estilo pueden venir completamente formadas, con sugerencias de redacción específica y actualizada, o pueden comenzar como preguntas vagas sobre la aplicabilidad de una regla determinada. Las ideas entrantes son discutidas por la comunidad, recibiendo comentarios de otros usuarios del idioma. Algunas propuestas son rechazadas por consenso comunitario, calibradas

ser innecesario, demasiado ambiguo o no beneficioso. Otros reciben comentarios positivos, evaluados para tener mérito, ya sea tal cual o con algún refinamiento sugerido. Estas propuestas, las que pasan por la revisión de la comunidad, están sujetas a la aprobación final de la toma de decisiones.

Los árbitros del estilo

En Google, para la guía de estilo de cada idioma, las decisiones finales y las aprobaciones las toman los propietarios de la guía de estilo, nuestros árbitros de estilo. Para cada lenguaje de programación, un grupo de expertos en idiomas desde hace mucho tiempo son los propietarios de la guía de estilo y los responsables de la toma de decisiones designados. Los árbitros de estilo para un idioma determinado suelen ser miembros senior del equipo de la biblioteca del idioma y otros Googlers de mucho tiempo con experiencia en idiomas relevantes.

La toma de decisiones real para cualquier cambio en la guía de estilo es una discusión de las compensaciones de ingeniería para la modificación propuesta. Los árbitros toman decisiones dentro del contexto de los objetivos acordados para los cuales se optimiza la guía de estilo. No se realizan cambios según preferencias personales; son juicios de compensación. De hecho, el grupo de árbitros de estilo C++ actualmente consta de cuatro miembros. Esto puede parecer extraño: tener un número impar de miembros del comité evitaría votos empatados en caso de una decisión dividida. Sin embargo, debido a la naturaleza del enfoque de toma de decisiones, donde nada es “porque creo que debería ser así” y todo es una evaluación de compensación, las decisiones se toman por consenso y no por votación. El grupo de cuatro miembros es felizmente funcional tal como está.

Excepciones

Sí, nuestras reglas son ley, pero sí, algunas reglas garantizan excepciones. Nuestras reglas están típicamente diseñadas para el caso general mayor. A veces, situaciones específicas se beneficiarían de una exención a una regla en particular. Cuando surge tal escenario, se consulta a los árbitros de estilo para determinar si existe un caso válido para otorgar una exención a una regla en particular.

Las exenciones no se otorgan a la ligera. En el código C++, si se introduce una API macro, la guía de estilo exige que se nombre con un prefijo específico del proyecto. Debido a la forma en que C++ maneja las macros, tratándolas como miembros del espacio de nombres global, todas las macros que se exportan desde archivos de encabezado deben tener nombres únicos globales para evitar colisiones. La regla de la guía de estilo con respecto a la denominación de macros permite exenciones otorgadas por árbitros para algunas macros de utilidades que son genuinamente globales. Sin embargo, cuando el motivo detrás de una solicitud de exención que pide excluir un prefijo específico del proyecto se reduce a preferencias debido a la longitud del nombre de la macro o la coherencia del proyecto, se rechaza la exención. La integridad del código base supera la consistencia del proyecto aquí.

Se permiten excepciones para los casos en los que se juzgue más beneficioso permitir el incumplimiento de la regla que evitarlo. La guía de estilo de C++ no permite la con-

versiones, incluidos los constructores de un solo argumento. Sin embargo, para los tipos que están diseñados para envolver de forma transparente a otros tipos, donde los datos subyacentes aún se representan de manera precisa y precisa, es perfectamente razonable permitir la conversión implícita. En tales casos, se otorgan exenciones a la regla de no conversión implícita. Tener un caso tan claro para las exenciones válidas podría indicar que la regla en cuestión debe aclararse o modificarse. Sin embargo, para esta regla específica, se reciben suficientes solicitudes de exención que parecen encajar en el caso válido de exención pero que en realidad no lo hacen, ya sea porque el tipo específico en cuestión no es en realidad un tipo contenedor transparente o porque el tipo es un contenedor pero no lo es. no es realmente necesario, que mantener la regla tal como está aún vale la pena.

Guía

Además de las reglas, seleccionamos la guía de programación en varias formas, que van desde una discusión larga y profunda de temas complejos hasta consejos breves y precisos sobre las mejores prácticas que respaldamos.

La guía representa la sabiduría recopilada de nuestra experiencia en ingeniería, documentando las mejores prácticas que hemos extraído de las lecciones aprendidas a lo largo del camino. La orientación tiende a centrarse en las cosas que hemos observado que las personas se equivocan con frecuencia o en cosas nuevas que no son familiares y, por lo tanto, están sujetas a confusión. Si las reglas son los "debe", nuestra guía es el "debería".

Un ejemplo de un grupo de orientación que cultivamos es un conjunto de manuales para algunos de los idiomas predominantes que usamos. Si bien nuestras guías de estilo son prescriptivas y dictaminan qué características del idioma están permitidas y cuáles no, las cartillas son descriptivas y explican las características que respaldan las guías. Son bastante amplios en su cobertura y abordan casi todos los temas que un ingeniero nuevo en el uso del lenguaje en Google necesitaría consultar. No profundizan en cada detalle de un tema determinado, pero brindan explicaciones y recomendaciones de uso. Cuando un ingeniero necesita averiguar cómo aplicar una función que quiere usar, los manuales tienen como objetivo servir como guía de referencia.

Hace algunos años, comenzamos a publicar una serie de consejos de C++ que ofrecían una combinación de consejos generales sobre idiomas y consejos específicos de Google. Cubrimos cosas difíciles: duración del objeto, semántica de copiar y mover, búsqueda dependiente de argumentos; cosas nuevas: características de C++ 11 tal como se adoptaron en la base de código, tipos de C++17 preadoptados como cadena_vista, opcional,yvariante;y cosas que necesitaban un pequeño empujón de corrección—recordatorios para no usarusandodirectivas, advertencias para recordar tener cuidado con implícito boolconversiones Los consejos surgen de los problemas reales encontrados y abordan cuestiones de programación reales que no están cubiertas por las guías de estilo. Sus consejos, a diferencia de las reglas de la guía de estilo, no son un verdadero canon; todavía están en la categoría de consejo más que de regla. Sin embargo, dada la forma en que crecen a partir de patrones observados en lugar de ideales abstractos, su aplicabilidad amplia y directa los distingue.

de la mayoría de los otros consejos como una especie de "canon de lo común". Los consejos tienen un enfoque limitado y son relativamente breves, cada uno de ellos de no más de unos pocos minutos de lectura. Esta serie de "Consejos de la semana" ha tenido mucho éxito internamente, con citas frecuentes durante las revisiones de código y las discusiones técnicas.¹²

Los ingenieros de software llegan a un nuevo proyecto o base de código con conocimiento del lenguaje de programación que van a usar, pero sin el conocimiento de cómo se usa el lenguaje de programación dentro de Google. Para cerrar esta brecha, mantenemos una serie de cursos "<Idioma>@Google 101" para cada uno de los principales lenguajes de programación en uso. Estos cursos de un día completo se enfocan en lo que hace que el desarrollo con ese lenguaje sea diferente en nuestra base de código. Cubren las bibliotecas y expresiones idiomáticas más utilizadas, las preferencias internas y el uso de herramientas personalizadas. Para un ingeniero de C++ que acaba de convertirse en ingeniero de Google C++, el curso completa las piezas faltantes que lo convierten no solo en un buen ingeniero, sino también en un buen ingeniero de código base de Google.

Además de impartir cursos que tienen como objetivo que alguien que no esté completamente familiarizado con nuestra configuración se ponga en marcha rápidamente, también cultivamos referencias listas para ingenieros en lo profundo de la base de código para encontrar la información que podría ayudarlos sobre la marcha. Estas referencias varían en forma y abarcan los idiomas que usamos. Algunas de las referencias útiles que mantenemos internamente incluyen las siguientes:

- Asesoramiento específico del idioma para las áreas que generalmente son más difíciles de corregir (como concurrencia y hashing).
- Desgloses detallados de las nuevas funciones que se introducen con una actualización del idioma y consejos sobre cómo usarlas dentro del código base.
- Listados de abstracciones clave y estructuras de datos proporcionados por nuestras bibliotecas. Esto evita que reinventemos estructuras que ya existen y da respuesta a "Necesito una cosa, pero no sé cómo se llama en nuestras bibliotecas".

Aplicar las reglas

Las reglas, por su naturaleza, otorgan mayor valor cuando son exigibles. Las reglas se pueden hacer cumplir socialmente, a través de la enseñanza y la capacitación, o técnicamente, con herramientas. Tenemos varios cursos de capacitación formales en Google que cubren muchas de las mejores prácticas que requieren nuestras reglas. También invertimos recursos en mantener nuestra documentación actualizada para garantizar que el material de referencia siga siendo preciso y actual. Una parte clave de nuestro enfoque de capacitación general en lo que respecta al conocimiento y la comprensión de nuestras reglas es el papel que desempeñan las revisiones de código. El proceso de legibilidad que ejecutamos aquí en Google —donde los ingenieros nuevos en el entorno de desarrollo de Google para un lenguaje determinado son asesorados a través de revisiones de código— se trata, en gran medida, de cultivar los hábitos

¹²<https://abseil.io/tips>tiene una selección de algunos de nuestros consejos más populares.

y patrones requeridos por nuestras guías de estilo (ver detalles sobre el proceso de legibilidad en [Capítulo 3](#)). El proceso es una parte importante de cómo nos aseguramos de que estas prácticas se aprendan y apliquen a través de los límites del proyecto.

Aunque siempre es necesario cierto nivel de capacitación (después de todo, los ingenieros deben aprender las reglas para poder escribir un código que las siga), cuando se trata de verificar el cumplimiento, en lugar de depender exclusivamente de la verificación basada en ingenieros, preferimos encarecidamente automatizar la aplicación con herramientas.

La aplicación automatizada de reglas garantiza que las reglas no se eliminen ni se olviden a medida que pasa el tiempo o se amplía la organización. Se unen nuevas personas; es posible que aún no conozcan todas las reglas. Las reglas cambian con el tiempo; incluso con una buena comunicación, no todos recordarán el estado actual de todo. Los proyectos crecen y agregan nuevas características; reglas que previamente no habían sido relevantes de repente son aplicables. Un ingeniero que verifica el cumplimiento de las reglas depende de la memoria o la documentación, las cuales pueden fallar. Siempre que nuestras herramientas se mantengan actualizadas, sincronizadas con los cambios de nuestras reglas, sabemos que todos nuestros ingenieros aplican nuestras reglas para todos nuestros proyectos.

Otra ventaja de la aplicación automatizada es la minimización de la variación en cómo se interpreta y aplica una regla. Cuando escribimos un script o usamos una herramienta para verificar el cumplimiento, validamos todas las entradas contra una definición única e invariable de la regla. No estamos dejando la interpretación en manos de cada ingeniero individual. Los ingenieros humanos ven todo con una perspectiva coloreada por sus prejuicios. Inconscientes o no, potencialmente sutiles e incluso posiblemente inofensivos, los sesgos aún cambian la forma en que las personas ven las cosas. Es probable que dejar la aplicación en manos de los ingenieros genere una interpretación y aplicación inconsistente de las reglas, potencialmente con expectativas inconsistentes de responsabilidad. Cuanto más deleguemos en las herramientas, menos puntos de entrada dejaremos para que entren los sesgos humanos.

Las herramientas también hacen que la aplicación sea escalable. A medida que una organización crece, un solo equipo de expertos puede escribir herramientas que el resto de la empresa puede usar. Si la empresa duplica su tamaño, el esfuerzo por hacer cumplir todas las reglas en toda la organización no se duplica, cuesta casi lo mismo que antes.

Incluso con las ventajas que obtenemos al incorporar herramientas, es posible que no sea posible automatizar la aplicación de todas las reglas. Algunas reglas técnicas exigen explícitamente el juicio humano. En la guía de estilo de C++, por ejemplo: "Evite la metaprogramación de plantilla complicada". "Utilizará para evitar nombres de tipo que sean ruidosos, obvios o sin importancia, casos en los que el tipo no ayuda a la claridad para el lector". "La composición suele ser más apropiada que la herencia". En la guía de estilo de Java: "No existe una única receta correcta sobre cómo [ordenar los miembros e inicializadores de su clase]; diferentes clases pueden ordenar sus contenidos de diferentes maneras". "Muy rara vez es correcto no hacer nada en respuesta a una excepción detectada". "Es extremadamente raro anular Objeto.finalizar." Para todas estas reglas, se requiere juicio y las herramientas no pueden (*¡todavía!*) ocupar ese lugar.

Otras reglas son sociales más que técnicas y, a menudo, no es prudente resolver los problemas sociales con una solución técnica. Para muchas de las reglas que se incluyen en esta categoría, los detalles tienden a estar un poco menos definidos y las herramientas se volverían complejas y costosas. A menudo es mejor dejar la aplicación de esas reglas a los humanos. Por ejemplo, cuando se trata del tamaño de un cambio de código dado (es decir, el número de archivos afectados y líneas modificadas), recomendamos que los ingenieros prefieran cambios más pequeños. Los cambios pequeños son más fáciles de revisar para los ingenieros, por lo que las revisiones tienden a ser más rápidas y exhaustivas. También es menos probable que introduzcan errores porque es más fácil razonar sobre el impacto potencial y los efectos de un cambio más pequeño. La definición de pequeño, sin embargo, es algo nebulosa. Un cambio que propague la actualización idéntica de una línea en cientos de archivos podría ser fácil de revisar. Por el contrario, un cambio más pequeño de 20 líneas podría introducir una lógica compleja con efectos secundarios que son difíciles de evaluar. Reconocemos que hay muchas medidas diferentes de tamaño, algunas de las cuales pueden ser subjetivas, especialmente cuando se tiene en cuenta la complejidad de un cambio. Es por eso que no tenemos ninguna herramienta para rechazar automáticamente un cambio propuesto que exceda un límite de línea arbitrario. Los revisores pueden (y lo hacen) retroceder si consideran que un cambio es demasiado grande. Para esta y otras reglas similares, la aplicación queda a discreción de los ingenieros que crean y revisan el código. Sin embargo, cuando se trata de reglas técnicas, siempre que sea factible, favorecemos la aplicación técnica. Un cambio más pequeño de 20 líneas podría introducir una lógica compleja con efectos secundarios que son difíciles de evaluar. Reconocemos que hay muchas medidas diferentes de tamaño, algunas de las cuales pueden ser subjetivas, especialmente cuando se tiene en cuenta la complejidad de un cambio. Es por eso que no tenemos ninguna herramienta para rechazar automáticamente un cambio propuesto que exceda un límite de línea arbitrario. Los revisores pueden (y lo hacen) retroceder si consideran que un cambio es demasiado grande. Para esta y otras reglas similares, la aplicación queda a discreción de los ingenieros que crean y revisan el código. Sin embargo, cuando se trata de reglas técnicas, siempre que sea factible, favorecemos la aplicación técnica. Un cambio más pequeño de 20 líneas podría introducir una lógica compleja con efectos secundarios que son difíciles de evaluar. Reconocemos que hay muchas medidas diferentes de tamaño, algunas de las cuales pueden ser subjetivas, especialmente cuando se tiene en cuenta la complejidad de un cambio. Es por eso que no tenemos ninguna herramienta para rechazar automáticamente un cambio propuesto que exceda un límite de línea arbitrario. Los revisores pueden (y lo hacen) retroceder si consideran que un cambio es demasiado grande. Para esta y otras reglas similares, la aplicación queda a discreción de los ingenieros que crean y revisan el código. Sin embargo, cuando se trata de reglas técnicas, siempre que sea factible, favorecemos la aplicación técnica. Algunos de los cuales pueden ser subjetivos, particularmente cuando se tiene en cuenta la complejidad de un cambio. Es por eso que no tenemos ninguna herramienta para rechazar automáticamente un cambio propuesto que excede un límite de línea arbitrario. Los revisores

Comprobadores de errores

Muchas reglas que cubren el uso del idioma se pueden aplicar con herramientas de análisis estático. De hecho, una encuesta informal de la guía de estilo de C++ realizada por algunos de nuestros bibliotecarios de C++ a mediados de 2018 estimó que aproximadamente el 90 % de sus reglas podrían verificarse automáticamente. Las herramientas de verificación de errores toman un conjunto de reglas o patrones y verifican que una muestra de código dada cumple completamente. La verificación automatizada elimina la carga de recordar todas las reglas aplicables del autor del código. Si un ingeniero solo necesita buscar advertencias de infracción, muchas de las cuales vienen con soluciones sugeridas, que aparecen durante la revisión del código por un analizador que se ha integrado estrechamente en el flujo de trabajo de desarrollo, minimizamos el esfuerzo que se necesita para cumplir con las reglas. Cuando comenzamos a usar herramientas para marcar funciones obsoletas basadas en el etiquetado de origen, apareciendo tanto la advertencia como la solución sugerida en el lugar, el problema de tener nuevos usos de API obsoletas desapareció casi de la noche a la mañana. Mantener bajo el costo del cumplimiento hace que sea más probable que los ingenieros lo sigan felizmente.

Utilizamos herramientas como [limpio](#) y [ordenado](#) (para C++) y [Propenso a errores](#) (para Java) para automatizar el proceso de hacer cumplir las reglas. Ver [capítulo 20](#) para una discusión en profundidad de nuestro enfoque.

Las herramientas que utilizamos están diseñadas y adaptadas para respaldar las reglas que definimos. La mayoría de las herramientas que respaldan las reglas son absolutas; todos deben cumplir con las reglas, por lo que todos usan las herramientas que las verifican. A veces, cuando las herramientas respaldan las mejores prácticas

ces donde hay un poco más de flexibilidad para cumplir con las convenciones, existen mecanismos de exclusión voluntaria para permitir que los proyectos se ajusten a sus necesidades.

Formateadores de código

En Google, generalmente utilizamos correctores de estilo y formateadores automáticos para aplicar un formato consistente dentro de nuestro código. La cuestión de la longitud de las líneas ha dejado de ser interesante.¹³ Los ingenieros solo ejecutan las verificadoras de estilo y siguen avanzando. Cuando el formateo se realiza de la misma manera cada vez, deja de ser un problema durante la revisión del código, lo que elimina los ciclos de revisión que, de lo contrario, se dedican a buscar, marcar y corregir problemas de estilo menores.

Al administrar la base de código más grande de la historia, hemos tenido la oportunidad de observar los resultados del formateo realizado por humanos frente al formateo realizado por herramientas automatizadas. Los robots son mejores en promedio que los humanos por una cantidad significativa. Hay algunos lugares donde la experiencia en el dominio es importante: formatear una matriz, por ejemplo, es algo que un ser humano generalmente puede hacer mejor que un formateador de propósito general. De lo contrario, formatear el código con un verificador de estilo automatizado rara vez sale mal.

Reforzamos el uso de estos formateadores con comprobaciones previas al envío: antes de que se pueda enviar el código, un servicio comprueba si la ejecución del formateador en el código produce diferencias. Si es así, el envío se rechaza con instrucciones sobre cómo ejecutar el formateador para corregir el código. La mayor parte del código en Google está sujeto a dicha verificación previa al envío. Para nuestro código, usamos `fmt` para C++; una envoltura interna alrededor de `Yapf` para Pitón; `gofmt` para Ir; `dartfmt` para dardo; y `constructor` para nuestro CONSTRUIR Archivos

Estudio de caso: gofmt

Samir Ajmani

Google lanzó el lenguaje de programación Go como código abierto el 10 de noviembre de 2009. Desde entonces, Go ha crecido como lenguaje para desarrollar servicios, herramientas, infraestructura en la nube y software de código abierto.¹⁴

Sabíamos que necesitábamos un formato estándar para el código Go desde el primer día. También sabíamos que sería casi imposible actualizar un formato estándar después del lanzamiento de código abierto. Entonces, el lanzamiento inicial de Go incluía `gofmt`, la herramienta de formato estándar para Go.

¹³ Cuando considera que se necesitan al menos dos ingenieros para tener la discusión y multiplique eso por el número de las veces que es probable que esta conversación suceda dentro de una colección de más de 30,000 ingenieros, resulta que "cuántos caracteres" puede convertirse en una pregunta muy costosa.

¹⁴ En diciembre de 2018, Go fue el idioma número 4 en GitHub según lo medido por solicitudes de incorporación de cambios..

Motivaciones

Las revisiones de código son una de las mejores prácticas de ingeniería de software, sin embargo, se dedicó demasiado tiempo a la revisión discutiendo sobre el formato. Aunque un formato estándar no sería el favorito de todos, sería lo suficientemente bueno para eliminar esta pérdida de tiempo.¹⁵

Al estandarizar el formato, sentamos las bases para las herramientas que podrían actualizar automáticamente el código Go sin crear diferencias falsas: el código editado por máquina sería indistinguible del código editado por humanos.^{diecisésis}

Por ejemplo, en los meses previos a Go 1.0 en 2012, el equipo de Go usó una herramienta llamada gofix para actualizar automáticamente el código Go anterior a 1.0 a la versión estable del lenguaje y las bibliotecas. Gracias a gofmt, las diferencias que produjo gofix incluyeron solo las partes importantes: cambios en los usos del lenguaje y las API. Esto permitió a los programadores revisar más fácilmente los cambios y aprender de los cambios realizados por la herramienta.

Impacto

Vaya programadores esperan que *todas*El código Go está formateado con gofmt. gofmt no tiene perillas de configuración y su comportamiento raramente cambia. Todos los principales editores e IDE usan gofmt o emulan su comportamiento, por lo que casi todo el código Go existente tiene el mismo formato. Al principio, los usuarios de Go se quejaron del estándar impuesto; ahora, los usuarios a menudo mencionan gofmt como una de las muchas razones por las que les gusta Go. Incluso al leer un código Go desconocido, el formato es familiar.

Miles de paquetes de código abierto leen y escriben código Go.¹⁷ Debido a que todos los editores e IDE están de acuerdo con el formato Go, las herramientas Go son portátiles y se integran fácilmente en nuevos entornos y flujos de trabajo de desarrolladores a través de la línea de comandos.

reequipamiento

En 2012, decidimos formatear automáticamente todosCONSTRUIRArchivos en Google usando un nuevo formateador estándar:constructor CONSTRUIRLos archivos contienen las reglas para compilar el software de Google con Blaze, el sistema de compilación de Google. Un estandarCONSTRUIREl formato nos permitiría crear herramientas que editen automáticamenteCONSTRUIRArchivos sin alterar su formato, tal como lo hacen las herramientas de Go con los archivos de Go.

Un ingeniero tardó seis semanas en volver a formatear las 200.000 páginas de GoogleCONSTRUIR archivos aceptados por los distintos propietarios de códigos, durante los cuales más de mil nuevos

¹⁵Charla de Robert Griesemer de 2015, "La evolución cultural de gofmt", proporciona detalles sobre la motivación, el diseño, e impacto de gofmt en Go y otros idiomas.

^{diecisésis}Russ Cox explicó en 2009que gofmt se trataba de cambios automatizados: "Así que tenemos todas las partes difíciles de un pro-herramienta de manipulación de gramo esperando ser utilizada. Aceptar aceptar el 'estilo gofmt' es la parte que lo hace factible en una cantidad finita de código".

¹⁷ El Ir ASTpaquetes de formatocada uno tiene miles de importadores.

CONSTRUIRSe agregaron archivos cada semana. La incipiente infraestructura de Google para realizar cambios a gran escala aceleró enormemente este esfuerzo. (Ver [capítulo 22](#).)

Conclusión

Para cualquier organización, pero especialmente para una organización tan grande como la fuerza de ingeniería de Google, las reglas nos ayudan a administrar la complejidad y construir una base de código mantenable. Un conjunto compartido de reglas enmarca los procesos de ingeniería para que puedan escalar y seguir creciendo, manteniendo tanto el código base como la organización sostenibles a largo plazo.

TL; DR

- Las reglas y la orientación deben apuntar a apoyar la resiliencia al tiempo y la escala.
- Conocer los datos para poder ajustar las reglas.
- **No todo debe ser una regla.**
- La consistencia es clave.
- Automatizar la aplicación cuando sea posible.

Revisión de código

Escrita por Tom Mansreck y Caitlin Sadowski
Editado por Lisa Carey

La revisión de código es un proceso en el que el código es revisado por alguien que no es el autor, a menudo antes de la introducción de ese código en una base de código. Aunque esa es una definición simple, las implementaciones del proceso de revisión de código varían ampliamente en la industria del software. Algunas organizaciones tienen un grupo selecto de "guardianes" en la base de código que revisan los cambios. Otros delegan los procesos de revisión de código a equipos más pequeños, lo que permite que diferentes equipos requieran diferentes niveles de revisión de código. En Google, prácticamente todos los cambios se revisan antes de confirmarse, y cada ingeniero es responsable de iniciar las revisiones y revisar los cambios.

Las revisiones de código generalmente requieren una combinación de un proceso y una herramienta que respalde ese proceso. En Google, utilizamos una herramienta de revisión de código personalizado, Critique, para respaldar nuestro proceso.¹ La crítica es una herramienta lo suficientemente importante en Google como para merecer su propio capítulo en este libro. Este capítulo se centra en el proceso de revisión de código tal como se practica en Google en lugar de la herramienta específica, porque estos fundamentos son más antiguos que la herramienta y porque la mayoría de estos conocimientos se pueden adaptar a cualquier herramienta que pueda usar para la revisión de código. .



Para obtener más información sobre la crítica, consulte [capítulo 19](#).

¹ También usamos Gerrit¹ para revisar el código Git, principalmente para nuestros proyectos de código abierto. Sin embargo, la Crítica es el principal herramienta de un ingeniero de software típico en Google.

Algunos de los beneficios de la revisión del código, como la detección de errores en el código antes de que ingresen a una base de código, están bien establecidos.²y algo obvio (si se mide de manera imprecisa). Otros beneficios, sin embargo, son más sutiles. Debido a que el proceso de revisión de código en Google es tan ubicuo y extenso, hemos notado muchos de estos efectos más sutiles, incluidos los psicológicos, que brindan muchos beneficios a una organización a lo largo del tiempo y la escala.

Flujo de revisión de código

Las revisiones de código pueden ocurrir en muchas etapas del desarrollo de software. En Google, las revisiones de código se llevan a cabo antes de que se pueda confirmar un cambio en la base de código; Esta etapa también se conoce como *revisión previa al compromiso*. El objetivo final principal de una revisión de código es lograr que otro ingeniero dé su consentimiento para el cambio, lo que denotamos etiquetando el cambio como "me parece bien" (LGTM). Usamos este LGTM como un "bit" de permisos necesarios (combinado con otros bits que se indican a continuación) para permitir que se confirme el cambio.

Una revisión de código típica en Google sigue los siguientes pasos:

1. Un usuario escribe un cambio en el código base en su espacio de trabajo. Esto *autor* luego crea una instantánea del cambio: un parche y la descripción correspondiente que se cargan en la herramienta de revisión de código. Este cambio produce una *diferencia* contra el código base, que se utiliza para evaluar qué código ha cambiado.
2. El autor puede usar este parche inicial para aplicar comentarios de revisión automatizados o realizar una auto revisión. Cuando el autor está satisfecho con la diferencia del cambio, envía el cambio por correo a uno o más revisores. Este proceso notifica a esos revisores y les pide que vean y comenten la instantánea.
3. *revisores* abra el cambio en la herramienta de revisión de código y publique comentarios en la diferencia. Algunos comentarios solicitan una resolución explícita. Algunos son meramente informativos.
4. El autor modifica el cambio y carga nuevas instantáneas basadas en los comentarios y luego responde a los revisores. Los pasos 3 y 4 se pueden repetir varias veces.
5. Una vez que los revisores están satisfechos con el estado más reciente del cambio, aceptan el cambio y lo aceptan marcándolo como "me parece bien" (LGTM). Solo se requiere una LGTM por defecto, aunque la convención podría solicitar que todos los revisores estén de acuerdo con el cambio.
6. Después de marcar un cambio como LGTM, el autor puede confirmar el cambio en el código base, siempre que *resolver todos los comentarios* que el cambio es *aprobado*. Cubriremos la aprobación en la siguiente sección.

²Steve McConnell, *Código completo*(Redmond: Microsoft Press, 2004).

Veremos este proceso con más detalle más adelante en este capítulo.

El código es una responsabilidad

Es importante recordar (y aceptar) que el código en sí mismo es una responsabilidad. Puede ser una responsabilidad necesaria, pero por sí mismo, el código es simplemente una tarea de mantenimiento para alguien en algún momento. Al igual que el combustible que lleva un avión, tiene peso, aunque es, por supuesto, necesario para que ese avión vuela.

Por supuesto, a menudo se necesitan nuevas características, pero se debe tener cuidado antes de desarrollar el código en primer lugar para garantizar que cualquier característica nueva esté garantizada. El código duplicado no solo es un esfuerzo desperdiciado, sino que en realidad puede costar más tiempo que no tener el código en absoluto; los cambios que podrían realizarse fácilmente bajo un patrón de código a menudo requieren más esfuerzo cuando hay duplicación en la base de código. Escribir un código completamente nuevo está tan mal visto que algunos de nosotros tenemos un dicho: "¡Si lo estás escribiendo desde cero, lo estás haciendo mal!"

Esto es especialmente cierto en el caso de la biblioteca o el código de utilidad. Lo más probable es que, si está escribiendo una utilidad, alguien más en algún lugar en una base de código del tamaño de Google probablemente haya hecho algo similar. Herramientas como las discutidas en [capítulo 17](#) por lo tanto, son fundamentales tanto para encontrar dicho código de utilidad como para evitar la introducción de código duplicado. Idealmente, esta investigación se realiza de antemano y se ha comunicado un diseño para cualquier cosa nueva a los grupos adecuados antes de escribir cualquier código nuevo.

Por supuesto, surgen nuevos proyectos, se introducen nuevas técnicas, se necesitan nuevos componentes, etc. Dicho todo esto, una revisión de código no es una ocasión para repetir o debatir decisiones de diseño anteriores. Las decisiones de diseño a menudo toman tiempo, lo que requiere la circulación de propuestas de diseño, el debate sobre el diseño en revisiones de API o reuniones similares, y tal vez el desarrollo de prototipos. Si bien una revisión de código completamente nueva no debe surgir de la nada, el proceso de revisión de código en sí mismo tampoco debe verse como una oportunidad para revisar decisiones anteriores.

Cómo funciona la revisión de código en Google

Hemos señalado aproximadamente cómo funciona el proceso típico de revisión de código, pero el problema está en los detalles. Esta sección describe en detalle cómo funciona la revisión de código en Google y cómo estas prácticas le permiten escalar adecuadamente con el tiempo.

Hay tres aspectos de la revisión que requieren "aprobación" para cualquier cambio en Google:

- Una verificación de corrección y comprensión por parte de otro ingeniero de que el código es apropiado y hace lo que el autor afirma que hace. A menudo se trata de un miembro del equipo, aunque no es necesario que lo sea. Esto se refleja en los permisos LGTM

"bit", que se establecerá después de que un revisor esté de acuerdo en que el código "se ve bien" para ellos.

- Aprobación de uno de los propietarios del código de que el código es apropiado para esta parte particular de la base de código (y se puede registrar en un directorio particular). Esta aprobación puede ser implícita si el autor es uno de esos propietarios. El código base de Google es una estructura de árbol con propietarios jerárquicos de directorios particulares. (Ver [capítulo 16](#)). Los propietarios actúan como guardianes de sus directorios particulares. Un cambio puede ser propuesto por cualquier ingeniero y aprobado por cualquier otro ingeniero, pero el propietario del directorio en cuestión también debe aprobar esta adición a su parte de la base de código. Dicho propietario podría ser un líder técnico u otro ingeniero considerado experto en esa área particular de la base de código. Por lo general, depende de cada equipo decidir qué tan amplio o limitado asignar los privilegios de propiedad.
- Aprobación de alguien con "legibilidad" del lenguaje³ que el código se ajuste al estilo y las mejores prácticas del lenguaje, verificando si el código está escrito de la manera que esperamos. Esta aprobación, nuevamente, podría estar implícita si el autor tiene tal legibilidad. Estos ingenieros provienen de un grupo de ingenieros de toda la empresa a quienes se les ha otorgado legibilidad en ese lenguaje de programación.

Aunque este nivel de control suena oneroso y, hay que reconocerlo, a veces lo es, la mayoría de las revisiones tienen una persona que asume los tres roles, lo que acelera bastante el proceso. Es importante destacar que el autor también puede asumir los dos últimos roles, ya que solo necesita un LGTM de otro ingeniero para verificar el código en su propia base de código, siempre que ya tenga legibilidad en ese idioma (lo que los propietarios suelen hacer).

Estos requisitos permiten que el proceso de revisión del código sea bastante flexible. Un líder técnico que es propietario de un proyecto y tiene la legibilidad del idioma de ese código puede enviar un cambio de código con solo un LGTM de otro ingeniero. Un pasante sin dicha autoridad puede enviar el mismo cambio a la misma base de código, siempre que obtenga la aprobación de un propietario con legibilidad del idioma. Los tres "bits" de permiso antes mencionados se pueden combinar en cualquier combinación. Un autor puede incluso solicitar más de una LGTM de personas separadas al etiquetar explícitamente el cambio como que desea una LGTM de todos los revisores.

En la práctica, la mayoría de las revisiones de código que requieren más de una aprobación generalmente pasan por un proceso de dos pasos: obtener una LGTM de un ingeniero colega y luego buscar la aprobación del propietario del código/revisor(es) de legibilidad apropiados. Esto permite que los dos roles se centren en diferentes aspectos de la revisión del código y ahorra tiempo de revisión. El revisor principal puede centrarse en la corrección del código y la validez general del cambio de código; el propietario del código puede centrarse en si este cambio es apropiado para su parte

3 En Google, "legibilidad" no se refiere simplemente a la comprensión, sino al conjunto de estilos y mejores prácticas que permitir que el código sea manejable para otros ingenieros. Ver [Capítulo 3](#).

del código base sin tener que centrarse en los detalles de cada línea de código. En otras palabras, un aprobador a menudo busca algo diferente a un revisor. Después de todo, alguien está tratando de registrar el código en su proyecto/directorio. Les preocupan más preguntas como: "¿Este código será fácil o difícil de mantener?" "¿Se suma a mi deuda técnica?" "¿Tenemos la experiencia para mantenerlo dentro de nuestro equipo?"

Si los tres tipos de revisiones pueden ser manejados por un solo revisor, ¿por qué no hacer que esos tipos de revisores manejen todas las revisiones de código? La respuesta corta es la escala. Separar los tres roles agrega flexibilidad al proceso de revisión de código. Si está trabajando con un compañero en una nueva función dentro de una biblioteca de utilidades, puede pedirle a alguien de su equipo que revise el código para verificar su corrección y comprensión. Después de varias rondas (quizás durante varios días), su código satisface a su revisor y obtiene un LGTM. Ahora, solo necesita obtener un *propietario* de la biblioteca (y los propietarios suelen tener la legibilidad adecuada) para aprobar el cambio.

Propiedad

hyrum-wright

Cuando se trabaja en un equipo pequeño en un repositorio dedicado, es común otorgar acceso a todo el equipo a todo el repositorio. Después de todo, usted conoce a los otros ingenieros, el dominio es lo suficientemente estrecho como para que cada uno de ustedes pueda ser experto, y los números pequeños restringen el efecto de los errores potenciales.

A medida que el equipo crece, este enfoque puede fallar a la hora de escalar. El resultado es una división desordenada del repositorio o un enfoque diferente para registrar quién tiene qué conocimiento y responsabilidades en diferentes partes del repositorio. En Google, llamamos a este conjunto de conocimientos y responsabilidades *propiedad* el pueblo para ejercerlos *dueños*. Este concepto es diferente a la posesión de una colección de código fuente, sino que implica un sentido de administración para actuar en el mejor interés de la empresa con una sección de la base de código. (De hecho, "mayordomos" sería casi seguro un mejor término si tuviéramos que hacerlo de nuevo).

Los archivos OWNERS con nombres especiales enumeran los nombres de usuario de las personas que tienen responsabilidades de propiedad sobre un directorio y sus elementos secundarios. Estos archivos también pueden contener referencias a otros archivos de PROPIETARIOS o listas de control de acceso externo, pero finalmente se resuelven en una lista de personas. Cada subdirectorio también puede contener un archivo OWNERS separado, y la relación es jerárquicamente aditiva: un archivo dado generalmente es propiedad de la unión de los miembros de todos los archivos OWNERS que se encuentran arriba en el árbol de directorios. Los archivos de PROPIETARIOS pueden tener tantas entradas como deseen los equipos, pero recomendamos una lista relativamente pequeña y enfocada para garantizar que la responsabilidad sea clara.

La propiedad del código de Google otorga derechos de aprobación para el código dentro del alcance de uno, pero estos derechos también llevan un conjunto de responsabilidades, como comprender el código que se posee o saber cómo encontrar a alguien que lo haga. Los diferentes equipos tienen

diferentes criterios para otorgar la propiedad a los nuevos miembros, pero generalmente los alentamos a no usar la propiedad como un rito de iniciación y alentamos a los miembros salientes a ceder la propiedad tan pronto como sea práctico.

Esta estructura de propiedad distribuida permite muchas de las otras prácticas que hemos descrito en este libro. Por ejemplo, el conjunto de personas en el archivo raíz OWNERS puede actuar como aprobadores globales para cambios a gran escala (ver [capítulo 22](#)) sin tener que molestar a los equipos locales. Del mismo modo, los archivos OWNERS actúan como una especie de documentación, lo que facilita que las personas y las herramientas encuentren a los responsables de un fragmento de código determinado simplemente subiendo por el árbol de directorios. Cuando se crean nuevos proyectos, no existe una autoridad central que deba registrar nuevos privilegios de propiedad: un nuevo archivo OWNERS es suficiente.

Este mecanismo de propiedad es simple, pero poderoso, y se ha escalado bien en las últimas dos décadas. Es una de las formas en que Google garantiza que decenas de miles de ingenieros puedan operar de manera eficiente en miles de millones de líneas de código en un solo repositorio.

Beneficios de la revisión de código

En toda la industria, la revisión del código en sí no es controvertida, aunque está lejos de ser una práctica universal. Muchas (quizás incluso la mayoría) de otras empresas y proyectos de código abierto tienen algún tipo de revisión de código, y la mayoría considera que el proceso es tan importante como una verificación de cordura en la introducción de código nuevo en una base de código. Los ingenieros de software comprenden algunos de los beneficios más obvios de la revisión de código, incluso si no creen personalmente que se aplica en todos los casos. Pero en Google, este proceso es generalmente más completo y generalizado que en la mayoría de las otras empresas.

La cultura de Google, como la de muchas empresas de software, se basa en dar a los ingenieros una amplia libertad en la forma en que realizan su trabajo. Se reconoce que los procesos estrictos tienden a no funcionar bien para una empresa dinámica que necesita responder rápidamente a las nuevas tecnologías, y que las reglas burocráticas tienden a no funcionar bien con los profesionales creativos. Sin embargo, la revisión del código es un mandato, uno de los pocos procesos generales en los que deben participar todos los ingenieros de software de Google. Google exige la revisión del código durante casi⁴cada cambio de código en el código base, por pequeño que sea. Este mandato tiene un costo y un efecto en la velocidad de la ingeniería dado que ralentiza la introducción de código nuevo en una base de código y puede afectar el tiempo de producción para cualquier cambio de código dado. (Ambas son quejas comunes de los ingenieros de software sobre los estrictos procesos de revisión de código). ¿Por qué, entonces, requerimos este proceso? ¿Por qué creemos que este es un beneficio a largo plazo?

⁴ Es posible que algunos cambios en la documentación y las configuraciones no requieran una revisión del código, pero a menudo aún es preferible era posible obtener tal revisión.

Un proceso de revisión de código bien diseñado y una cultura de tomarse en serio la revisión de código proporciona los siguientes beneficios:

- Verifica la corrección del código
- Garantiza que el cambio de código sea comprensible para otros ingenieros
- Hace cumplir la coherencia en la base de código
- Psicológicamente promueve la propiedad del equipo
- Permite compartir conocimientos
- Proporciona un registro histórico de la revisión del código en sí

Muchos de estos beneficios son críticos para una organización de software a lo largo del tiempo, y muchos de ellos son beneficiosos no solo para el autor sino también para los revisores. Las siguientes secciones entran en más detalles para cada uno de estos elementos.

Corrección del código

Un beneficio obvio de la revisión del código es que permite que un revisor verifique la "corrección" del cambio de código. Tener otro par de ojos mirando un cambio ayuda a garantizar que el cambio haga lo que se pretendía. Los revisores generalmente buscan si un cambio tiene las pruebas adecuadas, está diseñado correctamente y funciona de manera correcta y eficiente. En muchos casos, verificar la corrección del código es verificar si el cambio en particular puede introducir errores en la base de código.

Muchos informes apuntan a la eficacia de la revisión de código en la prevención de futuros errores en el software. Un estudio de IBM encontró que descubrir defectos antes en un proceso, como era de esperar, llevó a que se requiriera menos tiempo para solucionarlos más adelante. La inversión en el tiempo para la revisión del código ahorró el tiempo que de otro modo se dedicaría a probar, depurar y realizar regresiones, siempre que el proceso de revisión del código en sí se simplifique para mantenerlo liviano. Este último punto es importante; Los procesos de revisión de código que son pesados o que no se escalan correctamente se vuelven insostenibles. Veremos algunas de las mejores prácticas para mantener el proceso liviano más adelante en este capítulo.

Para evitar que la evaluación de la corrección se vuelva más subjetiva que objetiva, a los autores generalmente se les da preferencia a su enfoque particular, ya sea en el diseño o en la función del cambio introducido. Un revisor no debe proponer

5 "Avances en la Inspección de Software," *Transacciones IEEE sobre ingeniería de software*, SE-12(7): 744-751, julio de 1986.

Por supuesto, este estudio se llevó a cabo antes de que las herramientas robustas y las pruebas automatizadas se volvieran tan importantes en el proceso de desarrollo de software, pero los resultados aún parecen relevantes en la era moderna del software.

6 Rigby, Peter C. y Christian Bird. 2013. "Prácticas convergentes de revisión por pares de software". CESE/FSE 2013: *Pro-actas de la 9º Reunión Conjunta sobre Fundamentos de Ingeniería de Software 2013*, agosto de 2013: 202-212. <https://dl.acm.org/doi/10.1145/2491411.2491444>.

alternativas debido a la opinión personal. Los revisores pueden proponer alternativas, pero solo si mejoran la comprensión (por ser menos complejos, por ejemplo) o la funcionalidad (por ser más eficientes, por ejemplo). En general, se alienta a los ingenieros a aprobar cambios que mejoren la base de código en lugar de esperar el consenso sobre una solución más "perfecta". Este enfoque tiende a acelerar las revisiones de código.

A medida que las herramientas se fortalecen, muchas verificaciones de corrección se realizan automáticamente a través de técnicas como el análisis estático y las pruebas automatizadas (aunque es posible que las herramientas nunca eliminen por completo el valor de la inspección del código basada en humanos; consulte [capítulo 20](#) para más información). Aunque esta herramienta tiene sus límites, definitivamente ha enseñado la necesidad de confiar en las revisiones de código basadas en humanos para verificar la corrección del código.

Dicho esto, la verificación de defectos durante el proceso inicial de revisión del código sigue siendo una parte integral de una estrategia general de "cambio a la izquierda", con el objetivo de descubrir y resolver problemas lo antes posible para que no requieran costos y recursos escalados. más abajo en el ciclo de desarrollo. Una revisión de código no es una panacea ni la única verificación de dicha corrección, pero es un elemento de una defensa en profundidad contra tales problemas en el software. Como resultado, la revisión del código no necesita ser "perfecta" para lograr resultados.

Sorprendentemente, verificar la corrección del código no es el beneficio principal que obtiene Google del proceso de revisión del código. La verificación de la corrección del código generalmente garantiza que un cambio funcione, pero se le da más importancia a garantizar que un cambio de código sea comprensible y tenga sentido con el tiempo y a medida que la base de código se escala. Para evaluar esos aspectos, debemos observar otros factores además de si el código es lógicamente "correcto" o entendido.

Comprensión de código

Una revisión de código suele ser la primera oportunidad para que alguien que no sea el autor inspeccione un cambio. Esta perspectiva le permite a un revisor hacer algo que ni siquiera el mejor ingeniero puede hacer: proporcionar comentarios imparciales desde la perspectiva del autor. *Una revisión de código suele ser la primera prueba de si un cambio dado es comprensible para una audiencia más amplia.* Esta perspectiva es de vital importancia porque el código se leerá muchas más veces de las que se escribirá, y la comprensión y la comprensión son de vital importancia.

Suele ser útil encontrar un revisor que tenga una perspectiva diferente a la del autor, especialmente un revisor que podría necesitar, como parte de su trabajo, mantener o usar el código propuesto dentro del cambio. A diferencia de la deferencia que los revisores deben dar a los autores con respecto a las decisiones de diseño, a menudo es útil tratar las preguntas sobre la comprensión del código utilizando la máxima "el cliente siempre tiene la razón". En cierto sentido, cualquier pregunta que reciba ahora se multiplicará muchas veces con el tiempo, así que considere válida cada pregunta sobre comprensión de código. Esto no significa que deba cambiar su

enfoque o su lógica en respuesta a la crítica, pero sí significa que es posible que deba explicarlo más claramente.

Juntas, las verificaciones de corrección y comprensión del código son los criterios principales para un LGTM de otro ingeniero, que es uno de los bits de aprobación necesarios para una revisión de código aprobada. Cuando un ingeniero marca una revisión de código como LGTM, está diciendo que el código hace lo que dice y que es comprensible. Google, sin embargo, también requiere que el código se mantenga de manera sostenible, por lo que tenemos aprobaciones adicionales necesarias para el código en ciertos casos.

Coherencia de código

A escala, el código que escriba dependerá de otra persona y, finalmente, lo mantendrá. Muchos otros necesitarán leer su código y entender lo que hizo. Es posible que otros (incluidas las herramientas automatizadas) necesiten refactorizar su código mucho después de que haya pasado a otro proyecto. El código, por lo tanto, necesita ajustarse a algunos estándares de consistencia para que pueda ser entendido y mantenido. El código también debe evitar ser demasiado complejo; un código más simple es más fácil de entender y mantener para otros también. Los revisores pueden evaluar qué tan bien cumple este código con los estándares del propio código base durante la revisión del código. Una revisión del código, por lo tanto, debe actuar para garantizar *código de salud*.

Es por mantenibilidad que el estado LGTM de una revisión de código (que indica la corrección y comprensión del código) se separa del estado de aprobación de la legibilidad. Las aprobaciones de legibilidad solo pueden ser otorgadas por personas que hayan superado con éxito el proceso de capacitación en legibilidad del código en un lenguaje de programación en particular. Por ejemplo, el código Java requiere la aprobación de un ingeniero que tenga "legibilidad de Java".

Un aprobador de legibilidad tiene la tarea de revisar el código para asegurarse de que sigue las mejores prácticas acordadas para ese lenguaje de programación en particular, es consistente con el código base para ese lenguaje dentro del repositorio de código de Google y evita ser demasiado complejo. El código que es consistente y simple es más fácil de entender y más fácil de actualizar para las herramientas cuando llega el momento de la refactorización, lo que lo hace más resistente. Si un patrón en particular siempre se hace de una manera en el código base, es más fácil escribir una herramienta para refactorizarlo.

Además, el código puede escribirse solo una vez, pero se leerá docenas, cientos o incluso miles de veces. Tener un código que sea coherente en todo el código base mejora la comprensión de toda la ingeniería, y esta coherencia incluso afecta al proceso de revisión del código en sí. La consistencia a veces choca con la funcionalidad; un revisor de legibilidad puede preferir un cambio menos complejo que puede no ser funcionalmente "mejor" pero es más fácil de entender.

Con una base de código más consistente, es más fácil para los ingenieros intervenir y revisar el código en los proyectos de otra persona. En ocasiones, los ingenieros pueden necesitar mirar fuera del

equipo para obtener ayuda en una revisión de código. Ser capaz de comunicarse y pedirles a los expertos que revisen el código, sabiendo que pueden esperar que el código en sí sea consistente, les permite a esos ingenieros enfocarse más adecuadamente en la corrección y comprensión del código.

Beneficios Psicológicos y Culturales

La revisión del código también tiene importantes beneficios culturales: refuerza a los ingenieros de software que el código no es "suyo", sino que, de hecho, forma parte de una empresa colectiva. Dichos beneficios psicológicos pueden ser sutiles, pero siguen siendo importantes. Sin la revisión del código, la mayoría de los ingenieros se inclinarían naturalmente hacia un estilo personal y su propio enfoque para el diseño de software. El proceso de revisión del código obliga a un autor no solo a dejar que otros hagan aportes, sino también a comprometerse por el bien común.

Es parte de la naturaleza humana estar orgulloso del oficio de uno y ser reacio a abrir el código de uno a la crítica de los demás. También es natural ser algo reticente a recibir comentarios críticos sobre el código que uno escribe. El proceso de revisión del código proporciona un mecanismo para mitigar lo que de otro modo podría ser una interacción cargada de emociones. La revisión del código, cuando funciona mejor, proporciona no solo un desafío a las suposiciones de un ingeniero, sino que también lo hace de manera prescrita y neutral, actuando para atenuar cualquier crítica que de otro modo podría dirigirse al autor si se proporciona de manera no solicitada. Después de todo, el proceso requiere revisión crítica (de hecho, llamamos a nuestra herramienta de revisión de código "Crítica"), por lo que no puede culpar a un revisor por hacer su trabajo y ser crítico. El proceso de revisión de código en sí mismo, por lo tanto, puede actuar como el "policía malo", mientras que el revisor aún puede verse como el "policía bueno".

Por supuesto, no todos, ni siquiera la mayoría de los ingenieros necesitan tales dispositivos psicológicos. Pero amortiguar tales críticas a través del proceso de revisión del código a menudo proporciona una introducción mucho más suave para la mayoría de los ingenieros a las expectativas del equipo. Muchos ingenieros que se unen a Google, o a un nuevo equipo, se sienten intimidados por la revisión del código. Es fácil pensar que cualquier forma de revisión crítica se refleja negativamente en el desempeño laboral de una persona. Pero con el tiempo, casi todos los ingenieros llegan a esperar desafíos al enviar una revisión de código y llegan a valorar los consejos y las preguntas que se ofrecen a través de este proceso (aunque, es cierto, esto a veces lleva un tiempo).

Otro beneficio psicológico de la revisión de código es la validación. Incluso los ingenieros más capaces pueden sufrir el síndrome del impostor y ser demasiado autocríticos. Un proceso como la revisión del código actúa como validación y reconocimiento del trabajo de uno. A menudo, el proceso implica un intercambio de ideas y conocimientos (que se tratan en la siguiente sección), lo que beneficia tanto al revisor como al revisor. A medida que un ingeniero crece en el conocimiento de su dominio, a veces es difícil para ellos obtener comentarios positivos sobre cómo mejoran. El proceso de revisión de código puede proporcionar ese mecanismo.

El proceso de iniciar una revisión de código también obliga a todos los autores a tener un poco más de cuidado con sus cambios. Muchos ingenieros de software no son perfeccionistas; la mayoría admitirá que el código que "hace el trabajo" es mejor que el código que es perfecto pero que requiere demasiado

mucho tiempo en desarrollarse. Sin la revisión del código, es natural que muchos de nosotros tomemos atajos, incluso con la plena intención de corregir dichos defectos más adelante. "Claro, no tengo todas las pruebas unitarias hechas, pero puedo hacerlo más tarde". Una revisión de código obliga a un ingeniero a resolver esos problemas antes de enviar el cambio. La recopilación de los componentes de un cambio para la revisión del código obliga psicológicamente a un ingeniero a asegurarse de que todos sus patos estén en fila. El pequeño momento de reflexión que viene antes de enviar su cambio es el momento perfecto para leer su cambio y asegurarse de que no se está perdiendo nada.

El intercambio de conocimientos

Uno de los beneficios más importantes, pero subestimado, de la revisión de código es el intercambio de conocimientos. La mayoría de los autores eligen revisores que son expertos, o al menos conocedores, en el área bajo revisión. El proceso de revisión permite a los revisores impartir conocimiento del dominio al autor, lo que permite que los revisores ofrezcan sugerencias, nuevas técnicas o información de asesoramiento al autor. (Los revisores pueden incluso marcar algunos comentarios como "FYI", que no requieren ninguna acción; simplemente se agregan como una ayuda para el autor). Poder actuar como revisores para otros ingenieros.

Parte del proceso de revisión de código de retroalimentación y confirmación implica hacer preguntas sobre por qué el cambio se realiza de una manera particular. Este intercambio de información facilita el intercambio de conocimientos. De hecho, muchas revisiones de código implican un intercambio de información en ambos sentidos: tanto los autores como los revisores pueden aprender nuevas técnicas y patrones a partir de la revisión de código. En Google, los revisores pueden incluso compartir directamente las ediciones sugeridas con un autor dentro de la propia herramienta de revisión de código.

Es posible que un ingeniero no lea todos los correos electrónicos que se le envían, pero tiende a responder a cada revisión de código enviada. Este intercambio de conocimientos también puede ocurrir entre zonas horarias y proyectos, utilizando la escala de Google para difundir información rápidamente a los ingenieros en todos los rincones de la base de código. La revisión del código es el momento perfecto para la transferencia de conocimientos: es oportuna y procesable. (¡Muchos ingenieros en Google "conocen" a otros ingenieros primero a través de sus revisiones de código!)

Dada la cantidad de tiempo que los ingenieros de Google dedican a la revisión del código, el conocimiento acumulado es bastante significativo. La tarea principal de un ingeniero de Google sigue siendo la programación, por supuesto, pero una gran parte de su tiempo aún se dedica a la revisión del código. El proceso de revisión de código proporciona una de las principales formas en que los ingenieros de software interactúan entre sí e intercambian información sobre técnicas de codificación. A menudo, los nuevos patrones se anuncian en el contexto de la revisión del código, a veces a través de refactorizaciones, como cambios a gran escala.

Además, debido a que cada cambio se convierte en parte del código base, la revisión del código actúa como un registro histórico. Cualquier ingeniero puede inspeccionar el código base de Google y determinar cuándo se introdujo algún patrón en particular y mostrar la revisión del código real en

pregunta. A menudo, esa arqueología proporciona información a muchos más ingenieros que el autor original y los revisores.

Mejores prácticas de revisión de código

Es cierto que la revisión del código puede introducir fricciones y demoras en una organización. La mayoría de estos problemas no son problemas con la revisión del código per se, sino con la implementación elegida de la revisión del código. Mantener el proceso de revisión de código funcionando sin problemas en Google no es diferente, y requiere una serie de mejores prácticas para garantizar que la revisión de código valga la pena el esfuerzo realizado en el proceso. La mayoría de esas prácticas enfatizan mantener el proceso ágil y rápido para que la revisión del código pueda escalar adecuadamente.

Sea cortés y profesional

Como se señaló en la sección Cultura de este libro, Google fomenta en gran medida una cultura de confianza y respeto. Esto se filtra a nuestra perspectiva sobre la revisión del código. Un ingeniero de software necesita un LGTM de solo otro ingeniero para satisfacer nuestro requisito de comprensión de código, por ejemplo. Muchos ingenieros hacen comentarios y LGTM un cambio con el entendimiento de que el cambio se puede enviar después de realizar esos cambios, sin rondas adicionales de revisión. Dicho esto, las revisiones de código pueden generar ansiedad y estrés incluso a los ingenieros más capaces. Es sumamente importante mantener todos los comentarios y críticas firmemente en el ámbito profesional.

En general, los revisores deben deferir a los autores sobre enfoques particulares y solo señalar alternativas si el enfoque del autor es deficiente. Si un autor puede demostrar que varios enfoques son igualmente válidos, el revisor debe aceptar la preferencia del autor. Incluso en esos casos, si se encuentran defectos en un enfoque, considere la revisión como una oportunidad de aprendizaje (¡para ambas partes!). Todos los comentarios deben ser estrictamente profesionales. Los revisores deben tener cuidado de sacar conclusiones precipitadas basadas en el enfoque particular del autor del código. Es mejor hacer preguntas sobre por qué algo se hizo de la forma en que se hizo antes de asumir que ese enfoque es incorrecto.

Los revisores deben ser rápidos con sus comentarios. En Google, esperamos comentarios de una revisión de código dentro de las 24 horas (laborables). Si un revisor no puede completar una revisión en ese tiempo, es una buena práctica (y se espera) responder que al menos ha visto el cambio y llegará a la revisión lo antes posible. Los revisores deben evitar responder a la revisión del código de forma fragmentaria. Pocas cosas molestan más a un autor que recibir comentarios de una revisión, abordarla y luego continuar recibiendo más comentarios no relacionados en el proceso de revisión.

Así como esperamos profesionalismo por parte del revisor, también esperamos profesionalismo por parte del autor. Recuerda que tú no eres tu código, y que este cambio que propones no es “tuyo” sino del equipo. Despues de verificar ese fragmento de código en la base de código, ya no es suyo en ningún caso. Ser receptivo a

preguntas sobre su enfoque y prepárese para explicar por qué hizo las cosas de cierta manera. Recuerde que parte de la responsabilidad de un autor es asegurarse de que este código sea comprensible y mantenible para el futuro.

Es importante tratar cada comentario de un revisor dentro de una revisión de código como un elemento TODO; es posible que no sea necesario aceptar un comentario en particular sin cuestionarlo, pero al menos debe abordarse. Si no está de acuerdo con el comentario de un revisor, hágaselo saber y hágale saber por qué y no marque un comentario como resuelto hasta que cada parte haya tenido la oportunidad de ofrecer alternativas. Una forma común de mantener este tipo de debates civilizados si un autor no está de acuerdo con un revisor es ofrecer una alternativa y pedirle al revisor PTAL (por favor, eche otro vistazo). Recuerde que la revisión de código es una oportunidad de aprendizaje tanto para el revisor como para el autor. Esa idea a menudo ayuda a mitigar cualquier posibilidad de desacuerdo.

Del mismo modo, si es propietario de un código y responde a una revisión de código dentro de su base de código, sea receptivo a los cambios de un autor externo.

Siempre que el cambio sea una mejora en el código base, aún debe dar deferencia al autor de que el cambio indica algo que podría y debería mejorarse.

Escribir pequeños cambios

Probablemente la práctica más importante para mantener ágil el proceso de revisión de código es mantener los cambios pequeños. Idealmente, una revisión de código debería ser fácil de digerir y centrarse en un solo tema, tanto para el revisor como para el autor. El proceso de revisión de código de Google desalienta los cambios masivos que consisten en proyectos completamente formados, y los revisores pueden rechazar dichos cambios con razón por ser demasiado grandes para una sola revisión. Los cambios más pequeños también evitan que los ingenieros pierdan tiempo esperando revisiones de cambios más grandes, lo que reduce el tiempo de inactividad. Estos pequeños cambios también tienen beneficios más adelante en el proceso de desarrollo de software. Es mucho más fácil determinar la fuente de un error dentro de un cambio si ese cambio en particular es lo suficientemente pequeño como para reducirlo.

Dicho esto, es importante reconocer que un proceso de revisión de código que se basa en pequeños cambios a veces es difícil de reconciliar con la introducción de nuevas funciones importantes. Un conjunto de pequeños cambios de código incrementales puede ser más fácil de digerir individualmente, pero más difícil de comprender dentro de un esquema más grande. Es cierto que algunos ingenieros de Google no son fanáticos de la preferencia dada a los pequeños cambios. Existen técnicas para administrar dichos cambios de código (desarrollo en ramas de integración, administración de cambios utilizando una base diferente a HEAD), pero esas técnicas inevitablemente implican más gastos generales. Considere la optimización para cambios pequeños solo eso: una optimización, y permita que su proceso se adapte a cambios más grandes ocasionales.

Los cambios "pequeños" generalmente deben limitarse a unas 200 líneas de código. Un pequeño cambio debe ser fácil para un revisor y, casi tan importante, no ser tan engorroso que los cambios adicionales se retrasen esperando una revisión extensa. La mayoría de los cambios en

Se espera que Google sea revisado dentro de aproximadamente un día.⁷(Esto no significa necesariamente que la revisión finalice en un día, sino que los comentarios iniciales se brindan en un día). Alrededor del 35% de los cambios en Google se realizan en un solo archivo.⁸Ser fácil para un revisor permite cambios más rápidos en el código base y también beneficia al autor. El autor quiere una revisión rápida; esperar una revisión extensa durante una semana probablemente afectaría los cambios posteriores. Una pequeña revisión inicial también puede evitar un esfuerzo desperdiciado mucho más costoso en un enfoque incorrecto más adelante.

Debido a que las revisiones de código suelen ser pequeñas, es común que casi todas las revisiones de código en Google sean revisadas por una sola persona. Si ese no fuera el caso, si se esperara que un equipo sopesara todos los cambios en una base de código común, no hay forma de que el proceso en sí se escale. Al mantener pequeñas las revisiones de código, habilitamos esta optimización. No es raro que varias personas comenten sobre cualquier cambio dado, la mayoría de las revisiones de código se envían a un miembro del equipo, pero también se envían CC a los equipos apropiados, pero el revisor principal sigue siendo aquel cuyo LGTM se desea, y solo se necesita un LGTM para cualquier cambio. Cualquier otro comentario, aunque importante, sigue siendo opcional.

Mantener los cambios pequeños también permite que los revisores de "aprobación" aprueben más rápidamente cualquier cambio dado. Pueden inspeccionar rápidamente si el revisor del código principal hizo la diligencia debida y centrarse únicamente en si este cambio aumenta la base de código mientras mantiene la salud del código a lo largo del tiempo.

Escribir buenas descripciones de cambios

Una descripción de cambio debe indicar su tipo de cambio en la primera línea, como un resumen. La primera línea es un espacio principal y se utiliza para proporcionar resúmenes dentro de la propia herramienta de revisión de código, actuar como línea de asunto en cualquier correo electrónico asociado y convertirse en la línea visible que ven los ingenieros de Google en un resumen de historial dentro de Code Search (ver[capítulo 17](#)), por lo que la primera línea es importante.

Aunque la primera línea debe ser un resumen de todo el cambio, la descripción aún debe incluir detalles sobre lo que se está cambiando,*y por qué*. Una descripción de "Corrección de errores" no es útil para un revisor o un futuro arqueólogo del código. Si se realizaron varias modificaciones relacionadas en el cambio, enumérelas dentro de una lista (manteniéndola en el mensaje y pequeña). La descripción es el registro histórico de este cambio, y herramientas como la búsqueda de código le permiten encontrar quién escribió qué línea en cualquier cambio en particular en la base de código. Profundizar en el cambio original suele ser útil cuando se intenta corregir un error.

7 Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko y Alberto Bacchelli, "[Revisión de código moderno: un caso de estudio en Google](#)."

8 Ibíd.

Las descripciones no son la única oportunidad para agregar documentación a un cambio. Al escribir una API pública, generalmente no desea filtrar los detalles de la implementación, pero hágalo dentro de la implementación real, donde debe comentar generosamente. Si un revisor no entiende por qué hiciste algo, incluso si es correcto, es un buen indicador de que dicho código necesita una mejor estructura o mejores comentarios (o ambos). Si, durante el proceso de revisión del código, se llega a una nueva decisión, actualice la descripción del cambio o agregue los comentarios apropiados dentro de la implementación. Una revisión de código no es solo algo que haces en el momento presente; es algo que haces para registrar lo que hiciste para la posteridad.

Mantenga a los revisores al mínimo

La mayoría de las revisiones de código en Google son revisadas por un solo revisor.⁹ Debido a que el proceso de revisión de código permite que un solo individuo maneje los bits sobre la corrección del código, la aceptación del propietario y la legibilidad del lenguaje, el proceso de revisión de código se adapta bastante bien a una organización del tamaño de Google.

Hay una tendencia dentro de la industria, y dentro de los individuos, de tratar de obtener aportes adicionales (y consentimiento unánime) de una muestra representativa de ingenieros. Después de todo, cada revisor adicional puede agregar su propia perspectiva particular a la revisión del código en cuestión. Pero hemos encontrado que esto conduce a rendimientos decrecientes; el LGTM más importante es el primero, y los posteriores no aportan tanto como se podría pensar a la ecuación. El costo de revisores adicionales supera rápidamente su valor.

El proceso de revisión de código se optimiza en torno a la confianza que depositamos en nuestros ingenieros para que hagan lo correcto. En ciertos casos, puede ser útil hacer que varias personas revisen un cambio en particular, pero incluso en esos casos, esos revisores deben centrarse en diferentes aspectos del mismo cambio.

Automatice donde sea posible

La revisión del código es un proceso humano, y esa entrada humana es importante, pero si hay componentes del proceso del código que se pueden automatizar, intente hacerlo. Se deben explorar oportunidades para automatizar tareas humanas mecánicas; las inversiones en herramientas adecuadas cosechan dividendos. En Google, nuestras herramientas de revisión de código permiten a los autores enviar y sincronizar automáticamente los cambios en el sistema de control de código fuente una vez aprobados (generalmente se usan para cambios bastante simples).

Una de las mejoras tecnológicas más importantes con respecto a la automatización en los últimos años es el análisis estático automático de un cambio de código dado (ver [capítulo 20](#)). En lugar de exigir a los autores que ejecuten pruebas, linters o formateadores, las herramientas actuales de revisión de código de Google proporcionan la mayor parte de esa utilidad automáticamente a través de lo que es

⁹ Ibíd.

conocido como *presupone*. Un proceso de envío previo se ejecuta cuando un cambio se envía inicialmente a un revisor. Antes de que se envíe ese cambio, el proceso de envío previo puede detectar una variedad de problemas con el cambio existente, rechazar el cambio actual (y evitar enviar un correo electrónico incómodo a un revisor) y pedirle al autor original que corrija el cambio primero. Dicha automatización no solo ayuda con el proceso de revisión del código en sí, sino que también permite a los revisores concentrarse en preocupaciones más importantes que el formato.

Tipos de revisiones de código

¡Todas las revisiones de código no son iguales! Los diferentes tipos de revisión de código requieren diferentes niveles de enfoque en los diversos aspectos del proceso de revisión. Los cambios de código en Google generalmente caen en uno de los siguientes grupos (aunque a veces se superponen):

- Revisiones Greenfield y desarrollo de nuevas características
- Cambios de comportamiento, mejoras y optimizaciones
- Corrección de errores y reverisiones
- Refactorizaciones y cambios a gran escala

Reseñas de Código Greenfield

El tipo menos común de revisión de código es el de código completamente nuevo, el llamado *revisión de campo verde*. Una revisión de campo nuevo es el momento más importante para evaluar si el código resistirá la prueba del tiempo: que será más fácil de mantener a medida que el tiempo y la escala cambien las suposiciones subyacentes del código. Por supuesto, la introducción de un código completamente nuevo no debería ser una sorpresa. Como se mencionó anteriormente en este capítulo, el código es una desventaja, por lo que la introducción de un código completamente nuevo generalmente debería resolver un problema real en lugar de simplemente proporcionar otra alternativa. En Google, generalmente requerimos que el código y/o los proyectos nuevos se sometan a una revisión de diseño exhaustiva, además de una revisión de código. Una revisión de código no es el momento de debatir decisiones de diseño que ya se tomaron en el pasado (y de la misma manera, una revisión de código no es el momento de presentar el diseño de una API propuesta).

Para garantizar que el código sea sostenible, una revisión de campo nuevo debe garantizar que una API coincida con un diseño acordado (que puede requerir la revisión de un documento de diseño) y se pruebe *completamente*, con todos los extremos de la API teniendo algún tipo de prueba unitaria, y esas pruebas fallan cuando cambian las suposiciones del código. (Ver [Capítulo 11](#)). El código también debe tener propietarios adecuados (una de las primeras revisiones en un nuevo proyecto es a menudo de un solo archivo OWNERS para el nuevo directorio), estar suficientemente comentado y proporcionar documentación complementaria, si es necesario. Una revisión totalmente nueva también podría requerir la introducción de un proyecto en el sistema de integración continua. (Ver [Capítulo 23](#)).

Cambios de comportamiento, mejoras y optimizaciones

La mayoría de los cambios en Google generalmente caen en la amplia categoría de modificaciones al código existente dentro de la base de código. Estas adiciones pueden incluir modificaciones a los puntos finales de la API, mejoras a las implementaciones existentes u optimizaciones para otros factores, como el rendimiento. Estos cambios son el pan de cada día de la mayoría de los ingenieros de software.

En cada uno de estos casos, también se aplican las pautas que se aplican a una revisión totalmente nueva: ¿es necesario este cambio y este cambio mejora la base de código? ¡Algunas de las mejores modificaciones a un código base son en realidad eliminaciones! Deshacerse del código muerto u obsoleto es una de las mejores formas de mejorar el estado general del código de una base de código.

Cualquier modificación de comportamiento debe incluir necesariamente revisiones a las pruebas apropiadas para cualquier nuevo comportamiento de API. Los aumentos a la implementación deben probarse en un sistema de integración continua (CI) para garantizar que esas modificaciones no rompan ninguna suposición subyacente de las pruebas existentes. Además, las optimizaciones deben, por supuesto, garantizar que no afecten esas pruebas y es posible que deban incluir puntos de referencia de rendimiento para que los revisores los consulten. Algunas optimizaciones también pueden requerir pruebas comparativas.

Corrección de errores y reversiones

Inevitablemente, deberá enviar un cambio para corregir un error en su base de código. *Al hacerlo, evite la tentación de abordar otros temas.* Esto no solo aumenta el riesgo de aumentar el tamaño de la revisión del código, sino que también hace que sea más difícil realizar pruebas de regresión o que otros reviertan su cambio. Una corrección de errores debe centrarse únicamente en corregir el error indicado y (por lo general) actualizar las pruebas asociadas para detectar el error que ocurrió en primer lugar.

A menudo es necesario abordar el error con una prueba revisada. El error surgió porque las pruebas existentes eran inadecuadas o el código tenía ciertas suposiciones que no se cumplieron. Como revisor de una corrección de errores, es importante solicitar actualizaciones de las pruebas unitarias, si corresponde.

A veces, un cambio de código en una base de código tan grande como la de Google hace que falle alguna dependencia que no fue detectada correctamente por las pruebas o que descubre una parte no probada de la base de código. En esos casos, Google permite que tales cambios sean "retrocedidos", por lo general por parte de los clientes intermedios afectados. Una reversión consiste en un cambio que básicamente deshace el cambio anterior. Tales reversiones se pueden crear en segundos porque simplemente revierten el cambio anterior a un estado conocido, pero aún requieren una revisión del código.

También se vuelve de vital importancia que cualquier cambio que pueda causar una reversión potencial (¡y eso incluye todos los cambios!) sea lo más pequeño y atómico posible para que una reversión, si es necesaria, no cause más interrupciones en otras dependencias que puede ser

difícil de desenredar. En Google, hemos visto que los desarrolladores comienzan a depender del nuevo código muy rápidamente después de enviarlo y, como resultado, las revisiones a veces perjudican a estos desarrolladores. Los pequeños cambios ayudan a mitigar estas preocupaciones, tanto por su atomicidad como porque las revisiones de pequeños cambios tienden a realizarse rápidamente.

Refactorizaciones y cambios a gran escala

Muchos cambios en Google se generan automáticamente: el autor del cambio no es una persona, sino una máquina. Discutimos más sobre el proceso de cambio a gran escala (LSC) en [capítulo 22](#), pero incluso los cambios generados por máquinas requieren revisión. En los casos en que el cambio se considera de bajo riesgo, lo revisan revisores designados que tienen privilegios de aprobación para todo nuestro código base. Pero para los casos en los que el cambio puede ser arriesgado o requiere experiencia en el dominio local, se les puede pedir a los ingenieros individuales que revisen los cambios generados automáticamente como parte de su flujo de trabajo normal.

A primera vista, una revisión de un cambio generado automáticamente debe manejarse de la misma manera que cualquier otra revisión de código: el revisor debe verificar la corrección y la aplicabilidad del cambio. Sin embargo, alentamos a los revisores a limitar los comentarios en el cambio asociado y solo marcar las inquietudes que son específicas de su código, no la herramienta subyacente o el LSC que genera los cambios. Si bien el cambio específico puede ser generado por una máquina, el proceso general que genera estos cambios ya se revisó y los equipos individuales no pueden vetar el proceso, o no sería posible escalar dichos cambios en toda la organización. Si existe una inquietud sobre la herramienta o el proceso subyacente, los revisores pueden escalar fuera de banda a un grupo de supervisión de LSC para obtener más información.

También alentamos a los revisores de cambios automáticos a evitar ampliar su alcance. Al revisar una nueva característica o un cambio escrito por un compañero de equipo, a menudo es razonable pedirle al autor que aborde las inquietudes relacionadas dentro del mismo cambio, siempre que la solicitud siga el consejo anterior de mantener el cambio pequeño. Esto no se aplica a los cambios generados automáticamente porque el humano que ejecuta la herramienta puede tener cientos de cambios en curso, e incluso un pequeño porcentaje de cambios con comentarios de revisión o preguntas no relacionadas limita la escala a la que el humano puede operar la herramienta de manera efectiva.

Conclusión

La revisión de código es uno de los procesos más importantes y críticos en Google. La revisión de código actúa como el pegamento que conecta a los ingenieros entre sí, y el proceso de revisión de código es el principal flujo de trabajo del desarrollador del que deben depender casi todos los demás procesos, desde las pruebas hasta el análisis estático y la CI. Un proceso de revisión de código debe escalar adecuadamente y, por ese motivo, las mejores prácticas, incluidos pequeños cambios y comentarios e iteraciones rápidos, son importantes para mantener la satisfacción del desarrollador y la velocidad de producción adecuada.

TL; DR

- La revisión del código tiene muchos beneficios, incluida la garantía de la corrección, la comprensión y la coherencia del código en una base de código.
- Siempre verifique sus suposiciones a través de otra persona; optimizar para el lector.
- Proporcione la oportunidad de recibir comentarios críticos sin dejar de ser profesional.
- La revisión del código es importante para compartir conocimientos en toda la organización.
- La automatización es fundamental para escalar el proceso.
- La revisión del código en sí proporciona un registro histórico.

Documentación

*Escrito por Tom Mansreck
Editado por Riona MacNamara*

De las quejas que tienen la mayoría de los ingenieros sobre la escritura, el uso y el mantenimiento del código, una frustración común singular es la falta de documentación de calidad. “¿Cuáles son los efectos secundarios de este método?” “Recibí un error después del paso 3”. “¿Qué significa este acrónimo?” “¿Este documento está actualizado?” Todos los ingenieros de software han expresado quejas sobre la calidad, la cantidad o la falta de documentación a lo largo de su carrera, y los ingenieros de software de Google no son diferentes.

Los escritores técnicos y los gerentes de proyectos pueden ayudar, pero los ingenieros de software siempre necesitarán escribir la mayoría de la documentación ellos mismos. Los ingenieros, por lo tanto, necesitan las herramientas y los incentivos adecuados para hacerlo de manera efectiva. La clave para facilitarles la escritura de documentación de calidad es introducir procesos y herramientas que se adapten a la organización y que se vinculen con su flujo de trabajo existente.

En general, el estado de la documentación de ingeniería a fines de la década de 2010 es similar al estado de las pruebas de software a fines de la década de 1980. Todo el mundo reconoce que se necesita hacer más esfuerzo para mejorarlo, pero aún no hay un reconocimiento organizacional de sus beneficios críticos. Eso está cambiando, aunque lentamente. En Google, nuestros esfuerzos más exitosos han sido cuando la documentación *estratégo como código* e incorporado al flujo de trabajo de ingeniería tradicional, lo que facilita a los ingenieros escribir y mantener documentos simples.

¿Qué califica como documentación?

Cuando nos referimos a "documentación", nos referimos a todos los textos complementarios que un ingeniero necesita escribir para hacer su trabajo: no solo documentos independientes, sino también comentarios de código. (De hecho, la mayor parte de la documentación que escribe un ingeniero de Google

viene en forma de comentarios de código.) Discutiremos los diversos tipos de documentos de ingeniería más adelante en este capítulo.

¿Por qué se necesita la documentación?

La documentación de calidad tiene enormes beneficios para una organización de ingeniería. El código y las API se vuelven más comprensibles, lo que reduce los errores. Los equipos de proyecto están más enfocados cuando sus metas de diseño y los objetivos del equipo están claramente establecidos. Los procesos manuales son más fáciles de seguir cuando los pasos están claramente definidos. La incorporación de nuevos miembros a un equipo o base de código requiere mucho menos esfuerzo si el proceso está claramente documentado.

Pero debido a que todos los beneficios de la documentación son necesariamente posteriores, generalmente no obtienen beneficios inmediatos para el autor. A diferencia de las pruebas, que (como veremos) brindan beneficios rápidamente a un programador, la documentación generalmente requiere más esfuerzo por adelantado y no brinda beneficios claros a un autor hasta más tarde. Pero, al igual que las inversiones en pruebas, la inversión realizada en documentación se amortizará con el tiempo. Después de todo, es posible que escriba un documento solo una vez,¹ pero será leído cientos, quizás miles de veces después; su costo inicial se amortiza entre todos los futuros lectores. La documentación no solo se amplía con el tiempo, sino que también es fundamental para el resto de la organización. Ayuda a responder preguntas como estas:

- ¿Por qué se tomaron estas decisiones de diseño?
- ¿Por qué implementamos este código de esta manera?
- Por qué² implementar este código de esta manera, si está viendo su propio código dos años después?

Si la documentación transmite todos estos beneficios, ¿por qué los ingenieros generalmente la consideran "deficiente"? Una razón, como hemos mencionado, es que los beneficios no son *inmediato*, especialmente al escritor. Pero hay varias otras razones:

- Los ingenieros a menudo ven la escritura como una habilidad separada de la programación. (Intentaremos ilustrar que este no es exactamente el caso, e incluso donde lo es, no es necesariamente una habilidad separada de la ingeniería de software).
- Algunos ingenieros no sienten que sean escritores capaces. Pero no necesitas un dominio sólido del inglés.² para producir documentación viable. Solo necesitas salir un poco de ti mismo y ver las cosas desde la perspectiva de la audiencia.

1 Bien, deberá mantenerlo y revisarlo de vez en cuando.

2 El inglés sigue siendo el idioma principal para la mayoría de los programadores, y la mayoría de la documentación técnica para programadores se basa en la comprensión del inglés.

- Escribir documentación suele ser más difícil debido a la limitada compatibilidad con las herramientas o la integración en el flujo de trabajo del desarrollador.
- La documentación se ve como una carga adicional, algo más que mantener en lugar de algo que facilitará el mantenimiento de su código existente.

No todos los equipos de ingeniería necesitan un escritor técnico (e incluso si ese fuera el caso, no hay suficientes). Esto significa que los ingenieros, en general, escribirán la mayor parte de la documentación ellos mismos. Entonces, en lugar de obligar a los ingenieros a convertirse en redactores técnicos, deberíamos pensar en cómo hacer que la escritura de la documentación sea más fácil para los ingenieros. Decidir cuánto esfuerzo dedicar a la documentación es una decisión que su organización deberá tomar en algún momento.

La documentación beneficia a varios grupos diferentes. Incluso para el escritor, la documentación proporciona los siguientes beneficios:

- Ayuda a formular una API. Escribir documentación es una de las formas más seguras de averiguar si su API tiene sentido. A menudo, la propia redacción de la documentación lleva a los ingenieros a reevaluar decisiones de diseño que de otro modo no se cuestionarían. Si no puede explicarlo y no puede definirlo, probablemente no lo haya diseñado lo suficientemente bien.
- Proporciona una hoja de ruta para el mantenimiento y un registro histórico. En cualquier caso, se deben evitar los trucos en el código, pero los buenos comentarios ayudan mucho cuando miras el código que escribiste hace dos años, tratando de descubrir qué está mal.
- Hace que su código se vea más profesional y genere tráfico. Los desarrolladores asumirán naturalmente que una API bien documentada es una API mejor diseñada. Ese no es siempre el caso, pero a menudo están altamente correlacionados. Aunque este beneficio suena cosmético, no lo es tanto: el hecho de que un producto tenga una buena documentación suele ser un indicador bastante bueno de qué tan bien se mantendrá un producto.
- Generará menos preguntas de otros usuarios. Este es probablemente el mayor beneficio con el tiempo para alguien que escribe la documentación. Si tiene que explicarle algo a alguien más de una vez, por lo general tiene sentido documentar ese proceso.

Tan grandes como estos beneficios son para el escritor de la documentación, la parte del león de los beneficios de la documentación recaerá naturalmente en el lector. La Guía de estilo de C++ de Google señala la máxima "[optimizar para el lector](#)." Esta máxima se aplica no solo al código, sino también a los comentarios sobre el código o al conjunto de documentación adjunto a una API. Al igual que las pruebas, el esfuerzo que pone en escribir buenos documentos obtendrá beneficios muchas veces durante su vida útil. La documentación es crítica a lo largo del tiempo y obtiene enormes beneficios para el código especialmente crítico a medida que una organización escala.

La documentación es como el código

Los ingenieros de software que escriben en un único lenguaje de programación principal aún suelen utilizar diferentes lenguajes para resolver problemas específicos. Un ingeniero puede escribir scripts de shell o Python para ejecutar tareas de línea de comandos, o puede escribir la mayor parte de su código de back-end en C++ pero escribir código de middleware en Java, y así sucesivamente. Cada idioma es una herramienta en la caja de herramientas.

La documentación no debería ser diferente: es una herramienta, escrita en un idioma diferente (generalmente inglés) para realizar una tarea en particular. Escribir documentación no es muy diferente a escribir código. Como un lenguaje de programación, tiene reglas, una sintaxis particular y decisiones de estilo, a menudo para lograr un propósito similar al del código: hacer cumplir la coherencia, mejorar la claridad y evitar errores (de comprensión). Dentro de la documentación técnica, la gramática es importante no porque se necesiten reglas, sino para estandarizar la voz y evitar confundir o distraer al lector. Google requiere un cierto estilo de comentario para muchos de sus idiomas por este motivo.

Al igual que el código, los documentos también deben tener propietarios. Los documentos sin propietarios se vuelven obsoletos y difíciles de mantener. La propiedad clara también facilita el manejo de la documentación a través de los flujos de trabajo de los desarrolladores existentes: sistemas de seguimiento de errores, herramientas de revisión de código, etc. Por supuesto, los documentos con diferentes propietarios aún pueden entrar en conflicto entre sí. En esos casos, es importante designar *canónica*/codocumentación: determine la fuente primaria y consolide otros documentos asociados en esa fuente primaria (o elimine los duplicados).

El uso frecuente de "ir/enlaces" en Google (ver [Capítulo 3](#)) facilita este proceso. Los documentos con enlaces go/ sencillos a menudo se convierten en la fuente canónica de verdad. Otra forma de promover los documentos canónicos es asociarlos directamente con el código que documentan colocándolos directamente bajo control de código fuente y junto con el propio código fuente.

La documentación suele estar tan estrechamente unida al código que, en la medida de lo posible, debería tratarse *como código*. Es decir, su documentación debe:

- Tener políticas internas o reglas a seguir
- Estar bajo control de fuente
- Tener una propiedad clara responsable de mantener los documentos
- Someterse a revisiones de cambios (y cambiar *con el* código que documenta)
 - Haga un seguimiento de los problemas, ya que los errores se rastrean en el código
 - Ser evaluado periódicamente (probado, en algún aspecto)
- Si es posible, mida aspectos como precisión, frescura, etc. (las herramientas aún no se han puesto al día aquí)

Cuento más traten los ingenieros la documentación como "una de" las tareas necesarias del desarrollo de software, menos se resentirán por los costos iniciales de escritura y más beneficios obtendrán a largo plazo. Además, facilitar la tarea de documentación reduce esos costos iniciales.

Estudio de caso: Google Wiki

Cuando Google era mucho más pequeño y delgado, tenía pocos escritores técnicos. La forma más sencilla de compartir información era a través de nuestro propio wiki interno (GooWiki). Al principio, esto parecía un enfoque razonable; todos los ingenieros compartían un solo conjunto de documentación y podían actualizarlo según fuera necesario.

Pero a medida que Google escaló, los problemas con un enfoque de estilo wiki se hicieron evidentes. Debido a que no había verdaderos propietarios de los documentos, muchos quedaron obsoletos.³ Debido a que no se implementó ningún proceso para agregar nuevos documentos, comenzaron a aparecer documentos duplicados y conjuntos de documentos. GooWiki tenía un espacio de nombres plano y las personas no eran buenas para aplicar ninguna jerarquía a los conjuntos de documentación. En un momento, había de 7 a 10 documentos (dependiendo de cómo los contara) sobre la configuración de Borg, nuestro entorno informático de producción, solo unos pocos parecían mantenerse y la mayoría eran específicos de ciertos equipos con ciertos permisos. y suposiciones.

Otro problema con GooWiki se hizo evidente con el tiempo: las personas que podían arreglar los documentos no eran las personas que los usaban. Los nuevos usuarios que descubrían documentos incorrectos no podían confirmar que los documentos eran incorrectos o no tenían una manera fácil de informar errores. Sabían que algo andaba mal (porque el documento no funcionaba), pero no podían "arreglarlo". Por el contrario, las personas más capaces de arreglar los documentos a menudo no necesitaban consultarlos una vez que estaban escritos. La documentación se volvió tan deficiente a medida que Google creció que la calidad de la documentación se convirtió en la principal queja de los desarrolladores de Google en nuestras encuestas anuales para desarrolladores.

La forma de mejorar la situación era mover la documentación importante bajo el mismo tipo de control de código fuente que se usaba para rastrear los cambios de código. Los documentos comenzaron a tener sus propios propietarios, ubicaciones canónicas dentro del árbol de fuentes y procesos para identificar errores y corregirlos; la documentación comenzó a mejorar dramáticamente. Además, la forma en que se escribía y mantenía la documentación comenzó a parecerse a la forma en que se escribía y mantenía el código. Los errores en los documentos se pueden informar dentro de nuestro software de seguimiento de errores. Los cambios en los documentos podrían manejarse utilizando el proceso de revisión de código existente. Eventualmente, los ingenieros comenzaron a arreglar los documentos ellos mismos o enviar los cambios a los redactores técnicos (que a menudo eran los propietarios).

³ Cuando descartamos GooWiki, descubrimos que alrededor del 90 % de los documentos no tenían vistas ni actualizaciones en el pocos meses anteriores.

Trasladar la documentación al control de código fuente generó inicialmente mucha controversia. Muchos ingenieros estaban convencidos de que acabar con GooWiki, ese bastión de la libertad de información, conduciría a una mala calidad porque el listón de la documentación (requerir una revisión, requerir a los propietarios de los documentos, etc.) sería más alto. Pero ese no fue el caso. Los documentos mejoraron.

La introducción de Markdown como un lenguaje de formato de documentación común también ayudó porque facilitó a los ingenieros entender cómo editar documentos sin necesidad de conocimientos especializados en HTML o CSS. Google finalmente introdujo su propio marco para incrustar documentación dentro del código:[g3doc](#). Con ese marco, la documentación mejoró aún más, ya que los documentos coexistían con el código fuente dentro del entorno de desarrollo del ingeniero. Ahora, los ingenieros podrían actualizar el código y su documentación asociada en el mismo cambio (una práctica cuya adopción todavía estamos tratando de mejorar).

La diferencia clave fue que el mantenimiento de la documentación se convirtió en una experiencia similar al mantenimiento del código: los ingenieros archivaron errores, realizaron cambios en los documentos en las listas de cambios, enviaron los cambios a las revisiones de los expertos, etc. Aprovechar los flujos de trabajo de los desarrolladores existentes, en lugar de crear otros nuevos, fue un beneficio clave.

Conozca a su audiencia

Uno de los errores más importantes que cometen los ingenieros al escribir documentación es escribir solo para ellos mismos. Es natural hacerlo, y escribir por sí mismo no deja de tener valor: después de todo, es posible que deba mirar este código en unos años e intentar averiguar qué quiso decir una vez. También puede tener aproximadamente el mismo conjunto de habilidades que alguien que lee su documento. Pero si escribe solo para usted mismo, va a hacer ciertas suposiciones, y dado que su documento puede ser leído por una audiencia muy amplia (todos los ingenieros, desarrolladores externos), incluso unos pocos lectores perdidos es un gran costo. A medida que crece una organización, los errores en la documentación se vuelven más prominentes y sus suposiciones a menudo no se aplican.

En cambio, antes de comenzar a escribir, debe (formal o informalmente) identificar la(s) audiencia(s) que sus documentos necesitan satisfacer. Un documento de diseño podría necesitar persuadir a los tomadores de decisiones. Es posible que un tutorial deba proporcionar instrucciones muy explícitas a alguien que no esté familiarizado con su base de código. Es posible que una API deba proporcionar información de referencia completa y precisa para cualquier usuario de esa API, ya sean expertos o novatos. Siempre trate de identificar una audiencia principal y escriba para esa audiencia.

La buena documentación no necesita ser pulida o “perfecta”. Un error que cometen los ingenieros cuando escriben documentación es asumir que deben ser mucho mejores escritores. Según esa medida, pocos ingenieros de software escribirían. Piense en escribir como lo hace sobre las pruebas o cualquier otro proceso que necesite hacer como ingeniero. Escriba a su audiencia, con la voz y el estilo que esperan. Si puedes leer, puedes escribir. Recuerda eso

tu audiencia está donde tú estabas, pero *sin su nuevo conocimiento de dominio*. Así que no necesitas ser un gran escritor; solo necesita que alguien como usted esté tan familiarizado con el dominio como lo está ahora. (Y siempre que tenga una estaca en el suelo, puede mejorar este documento con el tiempo).

Tipos de audiencias

Hemos señalado que debe escribir en el nivel de habilidad y conocimiento de dominio apropiado para su audiencia. Pero, ¿quién es exactamente tu audiencia? Lo más probable es que tenga varias audiencias en función de uno o más de los siguientes criterios:

- Nivel de experiencia (programadores expertos o ingenieros jóvenes que tal vez ni siquiera estén familiarizados con el lenguaje).
- Conocimiento del dominio (miembros del equipo u otros ingenieros de su organización que solo estén familiarizados con los puntos finales de la API).
- Propósito (usuarios finales que podrían necesitar su API para realizar una tarea específica y necesitan encontrar esa información rápidamente, o gurús de software que son responsables de las entrañas de una implementación particularmente compleja que espera que nadie más necesite mantener).

En algunos casos, diferentes audiencias requieren diferentes estilos de escritura, pero en la mayoría de los casos, el truco consiste en escribir de una manera que se aplique lo más ampliamente posible a los diferentes grupos de audiencia. A menudo, deberá explicar un tema complejo tanto a un experto como a un novato. Escribir para el experto con conocimiento del dominio puede permitirle tomar atajos, pero confundirá al novato; por el contrario, explicar todo en detalle al novato sin duda molestará al experto.

Obviamente, escribir dichos documentos es un acto de equilibrio y no existe una fórmula milagrosa, pero una cosa que descubrimos es que ayuda a mantener sus documentos *pequeño*. Escriba de manera suficientemente descriptiva para explicar temas complejos a personas que no están familiarizadas con el tema, pero no pierda ni moleste a los expertos. Escribir un documento corto a menudo requiere que escriba uno más largo (obteniendo toda la información) y luego haga un pase de edición, eliminando la información duplicada donde pueda. Esto puede parecer tedioso, pero tenga en cuenta que este gasto se distribuye entre todos los lectores de la documentación. Como dijo una vez Blaise Pascal: "Si tuviera más tiempo, te habría escrito una carta más corta". Al mantener un documento breve y claro, se asegurará de que satisfaga tanto a un experto como a un novato.

Otra distinción importante de la audiencia se basa en cómo un usuario encuentra un documento:

- *buscadores* son ingenieros que *saber lo que quieren* y quieren saber si lo que están viendo encaja a la perfección. Un dispositivo pedagógico clave para esta audiencia es *consistencia*. Si está escribiendo documentación de referencia para este grupo, dentro de un

archivo de código, por ejemplo, querrá que sus comentarios sigan un formato similar para que los lectores puedan escanear rápidamente una referencia y ver si encuentran lo que están buscando.

- *Tropezonespuede* que no sepa exactamente lo que quiere. Es posible que solo tengan una vaga idea de cómo implementar aquello con lo que están trabajando. La clave para esta audiencia es *claridad*. Proporcione resúmenes o introducciones (en la parte superior de un archivo, por ejemplo) que expliquen el propósito del código que están viendo. También es útil para identificar cuándo un documento está *noapropiado* para una audiencia. Muchos documentos en Google comienzan con una "declaración TL; DR" como "TL; DR: si no está interesado en los compiladores de C ++ en Google, puede dejar de leer ahora".

Finalmente, una distinción de audiencia importante es entre la de un cliente (p. ej., un usuario de una API) y la de un proveedor (p. ej., un miembro del equipo del proyecto). En la medida de lo posible, los documentos destinados a uno deben mantenerse separados de los documentos destinados al otro. Los detalles de implementación son importantes para un miembro del equipo con fines de mantenimiento; los usuarios finales no deberían necesitar leer dicha información. A menudo, los ingenieros denotan decisiones de diseño dentro de la API de referencia de una biblioteca que publican. Dichos razonamientos pertenecen más apropiadamente a documentos específicos (documentos de diseño) o, en el mejor de los casos, dentro de los detalles de implementación del código oculto detrás de una interfaz.

Tipos de documentación

Los ingenieros escriben varios tipos diferentes de documentación como parte de su trabajo: documentos de diseño, comentarios de código, documentos de procedimientos, páginas de proyectos y más. Todo esto cuenta como "documentación". Pero es importante conocer los diferentes tipos, y *no mezclar tipos*. Un documento debe tener, en general, un propósito singular y ceñirse a él. Así como una API debe hacer una cosa y hacerlo bien, evite intentar hacer varias cosas dentro de un documento. En su lugar, separa esas piezas de forma más lógica.

Hay varios tipos principales de documentos que los ingenieros de software a menudo necesitan escribir:

- Documentación de referencia, incluidos los comentarios del código
- Documentos de diseño
- Tutoriales
- Documentación conceptual
- Páginas de destino

En los primeros días de Google, era común que los equipos tuvieran páginas wiki monolíticas con montones de enlaces (muchos rotos u obsoletos), alguna información conceptual sobre cómo funcionaba el sistema, una referencia de API, etc., todo salpicado. Dichos documentos fallan porque no sirven para un solo propósito (y también se alargan tanto

que nadie los leerá; algunas páginas wiki notorias se desplazaron a través de varias docenas de pantallas). En su lugar, asegúrese de que su documento tenga un único propósito, y si agregar algo a esa página no tiene sentido, probablemente desee encontrar, o incluso crear, otro documento para ese propósito.

Documentación de referencia

La documentación de referencia es el tipo más común que los ingenieros necesitan escribir; de hecho, a menudo necesitan escribir algún tipo de documento de referencia todos los días. Por documentación de referencia, nos referimos a cualquier cosa que documente el uso del código dentro del código base. Los comentarios de código son la forma más común de documentación de referencia que un ingeniero debe mantener. Dichos comentarios se pueden dividir en dos campos básicos: comentarios de API versus comentarios de implementación. Recuerde las diferencias de audiencia entre estos dos: los comentarios de la API no necesitan discutir detalles de implementación o decisiones de diseño y no pueden asumir que un usuario está tan versado en la API como el autor. Los comentarios de implementación, por otro lado, pueden asumir mucho más conocimiento del dominio del lector, aunque tenga cuidado al asumir demasiado: las personas abandonan los proyectos,

La mayor parte de la documentación de referencia, incluso cuando se proporciona como documentación separada del código, se genera a partir de comentarios dentro del propio código base. (Como debe ser; la documentación de referencia debe ser de una sola fuente tanto como sea posible). Algunos lenguajes como Java o Python tienen marcos de comentarios específicos (Javadoc, PyDoc, GoDoc) destinados a facilitar la generación de esta documentación de referencia. Otros lenguajes, como C++, no tienen una implementación estándar de "documentación de referencia", pero debido a que C++ separa su superficie API (en encabezado o.harchivos) de la implementación (.ccarchivos), los archivos de encabezado suelen ser un lugar natural para documentar una API de C++.

Google adopta este enfoque: una API de C++ merece tener su documentación de referencia en vivo dentro del archivo de encabezado. Otra documentación de referencia también está integrada directamente en el código fuente de Java, Python y Go. Debido a que el navegador Code Search de Google (ver[capítulo 17](#)) es tan sólido que hemos encontrado pocos beneficios en proporcionar documentación de referencia generada por separado. Los usuarios de Code Search no solo buscan código fácilmente, sino que generalmente pueden encontrar la definición original de ese código como el resultado principal. Tener la documentación junto con las definiciones del código también hace que la documentación sea más fácil de descubrir y mantener.

Todos sabemos que los comentarios de código son esenciales para una API bien documentada. Pero, ¿qué es exactamente un "buen" comentario? Anteriormente en este capítulo, identificamos dos audiencias principales para la documentación de referencia: buscadores y tropiezos. Los buscadores saben lo que quieren; los tropiezones no. La ventaja clave para los buscadores es una base de código comentada constantemente para que puedan escanear rápidamente una API y encontrar lo que están buscando. La victoria clave para los tropiezones es identificar claramente el propósito de una API, a menudo en la parte superior de un archivo.

encabezamiento. Veremos algunos comentarios de código en las subsecciones que siguen. Las pautas para comentar el código que se encuentran a continuación se aplican a C++, pero existen reglas similares en Google para otros idiomas.

Comentarios del archivo

Casi todos los archivos de código de Google deben contener un comentario de archivo. (Algunos archivos de encabezado que contienen solo una función de utilidad, etc., pueden desviarse de este estándar). Los comentarios del archivo deben comenzar con un encabezado de la siguiente forma:

```
// ----- // str_cat.h
```

```
// ----- //
```

```
// Este archivo de encabezado contiene funciones para concatenar y agregar eficientemente //
cadenas: StrCat() y StrAppend(). La mayor parte del trabajo dentro de estas rutinas // en realidad se
maneja mediante el uso de un tipo AlphaNum especial, que fue diseñado // para usarse como un
tipo de parámetro que administra de manera eficiente la conversión a
// cadenas y evita copias en las operaciones anteriores. ...
```

Por lo general, un comentario de archivo debe comenzar con un resumen del contenido del código que está leyendo. Debe identificar los principales casos de uso del código y la audiencia prevista (en el caso anterior, los desarrolladores que desean concatenar cadenas). Cualquier API que no se pueda describir sucintamente en el primer párrafo o en el segundo suele ser el signo de una API que no está bien pensada. Considere dividir la API en componentes separados en esos casos.

Comentarios de clase

La mayoría de los lenguajes de programación modernos están orientados a objetos. Los comentarios de clase son, por lo tanto, importantes para definir los "objetos" de la API en uso en una base de código. Todas las clases (y estructuras) públicas en Google deben contener un comentario de clase que describa la clase/estructura, los métodos importantes de esa clase y el propósito de la clase. En general, los comentarios de clase deben ser "sustantivos" con documentación que enfatice su aspecto de objeto. Es decir, "La clase Foo contiene x, y, z, te permite hacer Bar y tiene los siguientes aspectos de Baz", y así sucesivamente.

Los comentarios de clase generalmente deben comenzar con un comentario de la siguiente forma:

```
// ----- // AlfaNum
```

```
// ----- //
```

```
// La clase AlphaNum actúa como el tipo de parámetro principal para StrCat() y //
StrAppend(), proporcionando una conversión eficiente de valores numéricos,
booleanos y // hexadecimales (a través del tipo Hex) en cadenas.
```

Comentarios de función

Todas las funciones gratuitas, o métodos públicos de una clase, en Google también deben contener un comentario de función que describa cuál es la función.*lo hace*. Los comentarios de función deben enfatizar la *actividad/naturaleza* de su uso, comenzando con un verbo indicativo que describe lo que hace la función y lo que devuelve.

Los comentarios de función generalmente deben comenzar con un comentario de la siguiente forma:

```
// StrCat()  
//  
// Fusiona las cadenas o números dados, sin usar delimitador(es), //  
devuelve el resultado combinado como una cadena.  
...
```

Tenga en cuenta que comenzar un comentario de función con un verbo declarativo introduce coherencia en un archivo de encabezado. Un buscador puede escanear rápidamente una API y leer solo el verbo para tener una idea de si la función es apropiada: "Fusionar, Eliminar, Crear", etc.

Algunos estilos de documentación (y algunos generadores de documentación) requieren varias formas de repetitivo en los comentarios de funciones, como "Devoluciones:", "Lanzamientos:", etc., pero en Google no hemos encontrado que sean necesarios. A menudo es más claro presentar dicha información en un solo comentario en prosa que no se divide en límites de secciones artificiales:

```
// Crea un nuevo registro para un cliente con el nombre y la dirección  
proporcionados, // y devuelve el ID del registro, o lanza `DuplicateEntryError` si  
ya existe un // registro con ese nombre.  
int AddCustomer(nombre de cadena, dirección de cadena);
```

Observe cómo la condición posterior, los parámetros, el valor de retorno y los casos excepcionales se documentan juntos de forma natural (en este caso, en una sola oración), porque no son independientes entre sí. Agregar secciones repetitivas explícitas haría que el comentario fuera más detallado y repetitivo, pero no más claro (y posiblemente menos claro).

Documentos de diseño

La mayoría de los equipos de Google requieren un documento de diseño aprobado antes de comenzar a trabajar en cualquier proyecto importante. Un ingeniero de software normalmente escribe el documento de diseño propuesto utilizando una plantilla de documento de diseño específica aprobada por el equipo. Dichos documentos están diseñados para ser colaborativos, por lo que a menudo se comparten en Google Docs, que tiene buenas herramientas de colaboración. Algunos equipos requieren que dichos documentos de diseño se discutan y debatan en reuniones específicas del equipo, donde los expertos pueden discutir o criticar los puntos más finos del diseño. En algunos aspectos, estas discusiones de diseño actúan como una forma de revisión del código antes de escribir cualquier código.

Debido a que el desarrollo de un documento de diseño es uno de los primeros procesos que emprende un ingeniero antes de implementar un nuevo sistema, también es un lugar conveniente para garantizar

que varias preocupaciones están cubiertas. Las plantillas de documentos de diseño canónico en Google requieren que los ingenieros consideren aspectos de su diseño, como las implicaciones de seguridad, la internacionalización, los requisitos de almacenamiento y las preocupaciones de privacidad, etc. En la mayoría de los casos, esas partes de esos documentos de diseño son revisadas por expertos en esos dominios.

Un buen documento de diseño debe cubrir los objetivos del diseño, su estrategia de implementación y proponer decisiones de diseño clave con énfasis en sus compensaciones individuales. Los mejores documentos de diseño sugieren objetivos de diseño y cubren diseños alternativos, indicando sus puntos fuertes y débiles.

Un buen documento de diseño, una vez aprobado, también actúa no solo como un registro histórico, sino como una medida de si el proyecto logró con éxito sus objetivos. La mayoría de los equipos archivan sus documentos de diseño en una ubicación adecuada dentro de sus documentos de equipo para que puedan revisarlos más adelante. Suele ser útil revisar un documento de diseño antes de lanzar un producto para asegurarse de que los objetivos establecidos cuando se escribió el documento de diseño sigan siendo los objetivos establecidos en el lanzamiento (y si no lo son, el documento o el producto se pueden ajustar en consecuencia).

Tutoriales

Cada ingeniero de software, cuando se une a un nuevo equipo, querrá ponerse al día lo más rápido posible. Tener un tutorial que guíe a alguien a través de la configuración de un nuevo proyecto es invaluable; "Hello World" se ha consolidado como una de las mejores maneras de garantizar que todos los miembros del equipo comienzen con el pie derecho. Esto se aplica tanto a los documentos como al código. La mayoría de los proyectos merecen un documento de "Hola mundo" que no asuma nada y haga que el ingeniero haga que suceda algo "real".

A menudo, el mejor momento para escribir un tutorial, si aún no existe, es cuando se une a un equipo por primera vez. (También es el mejor momento para encontrar errores en cualquier tutorial existente que esté siguiendo). Obtenga un bloc de notas u otra forma de tomar notas y escriba todo lo que necesita hacer en el camino, asumiendo que no tiene conocimiento del dominio o restricciones de configuración especiales; una vez que haya terminado, probablemente sabrá qué errores cometió durante el proceso y por qué – y luego puede editar sus pasos para obtener un tutorial más simplificado. Lo importante es escribir *todo* que necesitas hacer en el camino; trate de no asumir ninguna configuración, permisos o conocimiento del dominio en particular. Si necesita asumir alguna otra configuración, indíquelo claramente al comienzo del tutorial como un conjunto de requisitos previos.

La mayoría de los tutoriales requieren que realice una serie de pasos en orden. En esos casos, numere esos pasos explícitamente. Si el enfoque del tutorial está en el *usuario* (por ejemplo, para la documentación del desarrollador externo), luego enumere cada acción que un usuario debe realizar. No enumere las acciones que el sistema puede realizar en respuesta a tales acciones del usuario. Es crítico e importante enumerar explícitamente cada paso al hacer esto. Nada es más molesto que un error en el paso 4 porque olvida decirle a alguien que autorice correctamente su nombre de usuario, por ejemplo.

Ejemplo: un mal tutorial

1. Descargue el paquete de nuestro servidor en <http://example.com>
2. Copie el script de shell en su directorio de inicio
3. Ejecute el script de shell
4. El sistema foobar se comunicará con el sistema de autenticación.
5. Una vez autenticado, foobar iniciará una nueva base de datos llamada "baz"
6. Pruebe "baz" ejecutando un comando SQL en la línea de comando
7. Escriba: CREAR BASE DE DATOS my_foobar_db;

En el procedimiento anterior, los pasos 4 y 5 ocurren en el extremo del servidor. No está claro si el usuario necesita hacer algo, pero no es así, por lo que esos efectos secundarios se pueden mencionar como parte del paso 3. Además, no está claro si el paso 6 y el paso 7 son diferentes. (No lo son). Combine todas las operaciones atómicas del usuario en pasos individuales para que el usuario sepa que necesita hacer algo en cada paso del proceso. Además, si su tutorial tiene entrada o salida visible para el usuario, indíquelo en líneas separadas (a menudo usando la convención de unnegrita monoespaciadafuente).

Ejemplo: un mal tutorial mejorado

1. Descargue el paquete de nuestro servidor en <http://ejemplo.com>:

```
$ rizo -Ihttp://ejemplo.com
```

2. Copie el script de shell en su directorio de inicio:

```
$ cp foobar.sh ~
```

3. Ejecute el script de shell en su directorio de inicio:

```
$ disco ~; foobar.sh
```

El sistema foobar se comunicará primero con el sistema de autenticación. Una vez autenticado, foobar iniciará una nueva base de datos llamada "baz" y abrirá un shell de entrada.

4. Pruebe "baz" ejecutando un comando SQL en la línea de comandos:

```
baz:$ CREAR BASE DE DATOS my_foobar_db;
```

Tenga en cuenta cómo cada paso requiere la intervención específica del usuario. Si, en cambio, el tutorial se centró en algún otro aspecto (por ejemplo, un documento sobre la "vida de un servidor"), enumere esos pasos desde la perspectiva de ese enfoque (lo que hace el servidor).

Documentación Conceptual

Algunos códigos requieren explicaciones o conocimientos más profundos que los que se pueden obtener simplemente leyendo la documentación de referencia. En esos casos, necesitamos documentación conceptual para proporcionar una descripción general de las API o los sistemas. Algunos ejemplos de documentación conceptual pueden ser una descripción general de la biblioteca para una API popular, un documento que describe el ciclo de vida de los datos dentro de un servidor, etc. En casi todos los casos, un documento conceptual pretende aumentar, no reemplazar, un conjunto de documentación de referencia. A menudo, esto lleva a la duplicación de alguna información, pero con un propósito: promover la claridad. En esos casos, no es necesario que un documento conceptual cubra todos los casos límite (aunque una referencia debería cubrir esos casos religiosamente). En este caso, es aceptable sacrificar algo de precisión por claridad.

Los documentos de “concepto” son las formas de documentación más difíciles de escribir. Como resultado, a menudo son el tipo de documento más descuidado dentro de la caja de herramientas de un ingeniero de software. Un problema al que se enfrentan los ingenieros al escribir documentación conceptual es que, a menudo, no se puede incrustar directamente en el código fuente porque no hay una ubicación canónica para colocarla. Algunas API tienen un área superficial de API relativamente amplia, en cuyo caso, un comentario de archivo podría ser un lugar apropiado para una explicación “conceptual” de la API. Pero a menudo, una API funciona junto con otras API y/o módulos. El único lugar lógico para documentar un comportamiento tan complejo es a través de un documento conceptual separado. Si los comentarios son las pruebas unitarias de la documentación, los documentos conceptuales son las pruebas de integración.

Incluso cuando una API tiene un alcance adecuado, a menudo tiene sentido proporcionar un documento conceptual separado. Por ejemplo, el de `Abseilformato` de `cadenaLa` biblioteca cubre una variedad de conceptos que los usuarios avanzados de la API deben comprender. En esos casos, tanto a nivel interno como externo, brindamos [undocumento de conceptos de formato](#).

Un documento conceptual debe ser útil para una amplia audiencia: tanto expertos como novatos. Además, es necesario enfatizar *claridad*, por lo que a menudo necesita sacrificar la integridad (algo que es mejor reservar para una referencia) y (a veces) la precisión estricta. Eso no quiere decir que un documento conceptual deba ser intencionalmente inexacto; simplemente significa que debe centrarse en el uso común y dejar los usos raros o los efectos secundarios para la documentación de referencia.

Páginas de destino

La mayoría de los ingenieros son miembros de un equipo, y la mayoría de los equipos tienen una “página de equipo” en algún lugar de la intranet de su empresa. A menudo, estos sitios son un poco complicados: una página de destino típica puede contener algunos enlaces interesantes, a veces varios documentos titulados “¡Lea esto primero!” y alguna información tanto para el equipo como para sus clientes. Dichos documentos comienzan siendo útiles pero rápidamente se convierten en desastres; porque se convierten

tan engorrosos de mantener, eventualmente se volverán tan obsoletos que solo los valientes o los desesperados los arreglarán.

Afortunadamente, estos documentos parecen intimidantes, pero en realidad son fáciles de arreglar: asegúrese de que una página de destino identifique claramente su propósito y luego incluya *solo* Enlaces a otras páginas para más información. Si algo en una página de destino está haciendo más que ser un policía de tráfico, es *no* *está haciendo su trabajo*. Si tiene un documento de configuración separado, vincúlelo desde la página de destino como un documento separado. Si tiene demasiados enlaces en la página de destino (su página no debe desplazarse por varias pantallas), considere dividir las páginas por taxonomía, en diferentes secciones.

La mayoría de las páginas de destino mal configuradas tienen dos propósitos diferentes: son la página de "ir a" para alguien que es un usuario de su producto o API, o son la página de inicio para un equipo. No haga que la página sirva a ambos maestros, se volverá confuso. Cree una "página de equipo" separada como una página interna aparte de la página de destino principal. Lo que el equipo necesita saber a menudo es bastante diferente de lo que un cliente de su API necesita saber.

Revisiones de documentación

En Google, todo el código debe revisarse, y nuestro proceso de revisión de código se entiende y acepta bien. En general, la documentación también necesita revisión (aunque esto es menos universalmente aceptado). Si desea "probar" si su documentación funciona, generalmente debe hacer que otra persona la revise.

Un documento técnico se beneficia de tres tipos diferentes de revisiones, cada una de las cuales enfatiza diferentes aspectos:

- Una revisión técnica, para la precisión. Esta revisión generalmente la realiza un experto en la materia, a menudo otro miembro de su equipo. A menudo, esto es parte de una revisión de código en sí.
- Una revisión de la audiencia, para mayor claridad. Suele ser alguien que no está familiarizado con el dominio. Puede ser alguien nuevo en su equipo o un cliente de su API.
- Una revisión escrita, por consistencia. Suele ser un escritor técnico o un voluntario.

Por supuesto, algunas de estas líneas a veces son borrosas, pero si su documento es de alto perfil o podría terminar siendo publicado externamente, probablemente quiera asegurarse de que reciba más tipos de revisiones. (Hemos utilizado un proceso de revisión similar para este libro). Cualquier documento tiende a beneficiarse de las revisiones antes mencionadas, incluso si algunas de esas revisiones son ad hoc. Dicho esto, incluso conseguir que un revisor revise tu texto es preferible a que nadie lo revise.

Es importante destacar que si la documentación está vinculada al flujo de trabajo de ingeniería, a menudo mejorará con el tiempo. La mayoría de los documentos en Google ahora implícitamente pasan por una revisión de la audiencia porque en algún momento, su audiencia los usará y, con suerte, le informará cuando no estén funcionando (a través de errores u otras formas de comentarios).

Estudio de caso: la biblioteca de guías para desarrolladores

Como se mencionó anteriormente, hubo problemas asociados con tener la mayor parte (casi toda) la documentación de ingeniería contenida en un wiki compartido: poca propiedad de la documentación importante, documentación en competencia, información obsoleta y dificultad para archivar errores o problemas con la documentación. Pero este problema no se vio en algunos documentos: la guía de estilo de Google C++ era propiedad de un grupo selecto de ingenieros senior (árbitros de estilo) que la administraban. El documento se mantuvo en buen estado porque ciertas personas se preocuparon por él. Implícitamente poseían ese documento. El documento también era canónico: solo había una guía de estilo de C++.

Como se mencionó anteriormente, la documentación que se encuentra directamente dentro del código fuente es una forma de promover el establecimiento de documentos canónicos; si la documentación se encuentra junto al código fuente, por lo general debería ser la más aplicable (con suerte). En Google, cada API suele tener un `ag3doc` directorio donde residen dichos documentos (escritos como archivos Markdown y legibles dentro de nuestro navegador Code Search). El hecho de que la documentación exista junto con el código fuente no solo establece la propiedad de facto, sino que hace que la documentación parezca más completamente "parte" del código.

Sin embargo, algunos conjuntos de documentación no pueden existir de manera muy lógica dentro del código fuente. Una "guía para desarrolladores de C++" para Googlers, por ejemplo, no tiene un lugar obvio para ubicarse dentro del código fuente. No existe un directorio maestro "C++" donde las personas busquen dicha información. En este caso (y en otros que cruzaron los límites de la API), resultó útil crear conjuntos de documentación independientes en su propio depósito. Muchos de estos, seleccionados, asociaron documentos existentes en un conjunto común, con una navegación y una apariencia comunes. Dichos documentos se anotaron como "Guías para desarrolladores" y, al igual que el código en el código base, estaban bajo control de fuente en un depósito de documentación específico, con este depósito organizado por tema en lugar de API. A menudo, los escritores técnicos administraron estas guías para desarrolladores porque eran mejores para explicar los temas a través de los límites de la API.

Con el tiempo, estas guías para desarrolladores se volvieron canónicas. Los usuarios que escribieron documentos alternativos o complementarios se mostraron dispuestos a agregar sus documentos al conjunto de documentos canónicos después de que se estableciera, y luego desaprobar sus documentos alternativos. Eventualmente, la guía de estilo de C++ se convirtió en parte de una "Guía para desarrolladores de C++" más grande. A medida que el conjunto de documentación se volvió más completo y autorizado, su calidad también mejoró. Los ingenieros comenzaron a registrar errores porque sabían que alguien estaba manteniendo estos documentos. Debido a que los documentos estaban bloqueados bajo control de código fuente, con los propietarios adecuados, los ingenieros también comenzaron a enviar listas de cambios directamente a los redactores técnicos.

La introducción de enlaces go/ (ver [Capítulo 3](#)) permitió que la mayoría de los documentos, en efecto, se establecieran más fácilmente como canónicos sobre cualquier tema dado. Nuestra Guía para desarrolladores de C++ se estableció en "go/cpp", por ejemplo. Con una mejor búsqueda interna, enlaces go/ y la integración de múltiples documentos en un conjunto de documentación común, dichos conjuntos de documentación canónica se volvieron más fidedignos y sólidos con el tiempo.

Filosofía de la documentación

Advertencia: la siguiente sección es más un tratado sobre las mejores prácticas de escritura técnica (y la opinión personal) que sobre "cómo lo hace Google". Considérelo opcional para que los ingenieros de software lo comprendan por completo, aunque comprender estos conceptos probablemente le permitirá escribir información técnica más fácilmente.

QUIÉN, QUÉ, CUÁNDO, DÓNDE y POR QUÉ

La mayoría de la documentación técnica responde a una pregunta de "CÓMO". ¿Como funciona esto? ¿Cómo programo para esta API? ¿Cómo configuro este servidor? Como resultado, hay una tendencia de los ingenieros de software a saltar directamente al "CÓMO" en cualquier documento dado e ignorar las otras preguntas asociadas con él: QUIÉN, QUÉ, CUÁNDO, DÓNDE y POR QUÉ. Es cierto que ninguno de ellos es generalmente tan importante como el CÓMO (un documento de diseño es una excepción porque un aspecto equivalente suele ser el POR QUÉ), pero sin un marco adecuado de documentación técnica, los documentos terminan siendo confusos. Trate de abordar las otras preguntas en los dos primeros párrafos de cualquier documento:

- QUIÉN se discutió anteriormente: esa es la audiencia. Pero a veces también necesita llamar y dirigirse explícitamente a la audiencia en un documento. Ejemplo: "Este documento es para nuevos ingenieros en el proyecto Secret Wizard".
- QUÉ identifica el propósito de este documento: "Este documento es un tutorial diseñado para iniciar un servidor Frobber en un entorno de prueba". A veces, simplemente escribir el QUÉ lo ayuda a enmarcar el documento de manera adecuada. Si comienza a agregar información que no se aplica al QUÉ, es posible que desee mover esa información a un documento separado.
- CUÁNDO identifica cuándo se creó, revisó o actualizó este documento. Los documentos en el código fuente tienen esta fecha anotada implícitamente, y algunos otros esquemas de publicación también automatizan esto. Pero, si no, asegúrese de anotar la fecha en que se escribió el documento (o se revisó por última vez) en el documento mismo.
- DÓNDE suele estar implícito también, pero decide dónde debe residir el documento. Por lo general, la preferencia debe estar bajo algún tipo de control de versión, idealmente *con el código fuente que documenta*. Pero otros formatos también funcionan para diferentes propósitos. En Google, a menudo usamos Google Docs para facilitar la colaboración, especialmente en

problemas de diseño. En algún momento, sin embargo, cualquier documento compartido se vuelve menos una discusión y más un registro histórico estable. En ese momento, muévalo a un lugar más permanente, con propiedad clara, control de versiones y responsabilidad.

- **POR QUÉ** establece el propósito del documento. Resuma lo que espera que alguien se lleve del documento después de leerlo. Una buena regla general es establecer el **POR QUÉ** en la introducción de un documento. Cuando escriba el resumen, verifique si cumplió con sus expectativas originales (y modifique en consecuencia).

El principio, el medio y el final

Todos los documentos, de hecho, todas las partes de los documentos, tienen un principio, un medio y un final. Aunque suene increíblemente tonto, la mayoría de los documentos deberían tener, como mínimo, esas tres secciones. Un documento con una sola sección tiene una sola cosa que decir, y muy pocos documentos tienen una sola cosa que decir. No tenga miedo de agregar secciones a su documento; dividen el flujo en partes lógicas y brindan a los lectores una hoja de ruta de lo que cubre el documento.

Incluso el documento más simple suele tener más de una cosa que decir. Nuestros populares "Consejos de la semana de C++" han sido tradicionalmente muy breves y se centran en un pequeño consejo. Sin embargo, incluso aquí, tener secciones ayuda. Tradicionalmente, la primera sección denota el problema, la sección del medio repasa las soluciones recomendadas y la conclusión resume las conclusiones. Si el documento hubiera constado de una sola sección, algunos lectores sin duda tendrían dificultades para desentrañar los puntos importantes.

La mayoría de los ingenieros detestan la redundancia, y con razón. Pero en la documentación, la redundancia suele ser útil. Un punto importante enterrado dentro de una pared de texto puede ser difícil de recordar o de descifrar. Por otro lado, colocar ese punto en una ubicación más prominente al principio puede perder el contexto proporcionado más adelante. Por lo general, la solución es introducir y resumir el punto dentro de un párrafo introductorio y luego usar el resto de la sección para presentar su caso de manera más detallada. En este caso, la redundancia ayuda al lector a comprender la importancia de lo que se está enunciando.

Los parámetros de una buena documentación

Por lo general, hay tres aspectos de una buena documentación: integridad, precisión y claridad. Rara vez obtiene los tres en el mismo documento; a medida que intenta hacer un documento más "completo", por ejemplo, la claridad puede comenzar a sufrir. Si intenta documentar todos los casos de uso posibles de una API, podría terminar con un lío incomprensible. Para los lenguajes de programación, ser completamente preciso en todos los casos (y documentar todos los posibles efectos secundarios) también puede afectar la claridad. Para otros documentos, tratar de ser claro sobre un tema complicado puede afectar sutilmente la precisión del documento; puede decidir ignorar algunos efectos secundarios raros en un documento conceptual, por ejemplo,

porque el objetivo del documento es familiarizar a alguien con el uso de una API, no proporcionar una descripción dogmática de todo el comportamiento previsto.

En cada caso, un “buen documento” se define como el documento que *es haciendo su trabajo previsto*. Como resultado, rara vez querrá que un documento haga más de un trabajo. Para cada documento (y para cada tipo de documento), decida su enfoque y ajuste la escritura apropiadamente. ¿Escribir un documento conceptual? Probablemente no necesite cubrir cada parte de la API. ¿Escribir una referencia? Probablemente quieras que esté completo, pero tal vez debas sacrificar algo de claridad. ¿Escribir una página de aterrizaje? Concéntrate en la organización y mantenga la discusión al mínimo. Todo esto se suma a la calidad, que, sin duda, es obstinadamente difícil de medir con precisión.

¿Cómo se puede mejorar rápidamente la calidad de un documento? Centrarse en las necesidades de la audiencia. A menudo, menos es más. Por ejemplo, un error que suelen cometer los ingenieros es agregar decisiones de diseño o detalles de implementación a un documento de API. Al igual que idealmente debería separar la interfaz de una implementación dentro de una API bien diseñada, debe evitar discutir las decisiones de diseño en un documento de API. Los usuarios no necesitan conocer esta información. En su lugar, ponga esas decisiones en un documento especializado para ese propósito (generalmente un documento de diseño).

Documentos obsoletos

Al igual que el código antiguo puede causar problemas, también lo pueden hacer los documentos antiguos. Con el tiempo, los documentos se vuelven obsoletos, obsoletos o (a menudo) abandonados. Intente en la medida de lo posible evitar los documentos abandonados, pero cuando un documento ya no sirve para ningún propósito, elimínelo o identifíquelo como obsoleto (y, si está disponible, indique a dónde acudir para obtener nueva información). Incluso para documentos sin dueño, alguien agrega una nota que dice “[Esto ya no funciona]” es más útil que no decir nada y dejar algo que parece autoritario pero que ya no funciona.

En Google, a menudo adjuntamos “fechas de actualización” a la documentación. Dichos documentos indican la última vez que se revisó un documento, y los metadatos en el conjunto de documentación enviarán recordatorios por correo electrónico cuando el documento no se haya tocado, por ejemplo, tres meses. Tales fechas de actualización, como se muestra en el siguiente ejemplo, y el seguimiento de sus documentos como errores, pueden ayudar a que un conjunto de documentación sea más fácil de mantener a lo largo del tiempo, que es la principal preocupación de un documento:

```
<!--*
# Actualización del documento: para obtener más información, consulte ir/fresh-source.
frescura: { propietario: 'nombre de usuario' revisado: '2019-02-27' }
*-->
```

Los usuarios que poseen dicho documento tienen un incentivo para mantener actualizada la fecha de actualización (y si el documento está bajo control de fuente, eso requiere una revisión del código). Como resultado, es un medio de bajo costo para garantizar que un documento se revise de vez en cuando. En Google, descubrimos que incluir el propietario de un documento en esta actualización

fecha dentro del propio documento con una línea de "Última revisión por..." condujo a una mayor adopción también.

¿Cuándo necesita escritores técnicos?

Cuando Google era joven y estaba creciendo, no había suficientes escritores técnicos en ingeniería de software. (Ese sigue siendo el caso). Esos proyectos que se consideraban importantes tendían a recibir un redactor técnico, independientemente de si ese equipo realmente necesitaba uno. La idea era que el escritor pudiera aliviar al equipo de parte de la carga de escribir y mantener documentos y (teóricamente) permitir que el importante proyecto alcanzara una mayor velocidad. Esto resultó ser una mala suposición.

Aprendimos que la mayoría de los equipos de ingeniería pueden escribir documentación para ellos mismos (su equipo) perfectamente bien; solo cuando están escribiendo documentos para otra audiencia tienden a necesitar ayuda porque es difícil escribir para otra audiencia. El circuito de retroalimentación dentro de su equipo con respecto a los documentos es más inmediato, el conocimiento del dominio y las suposiciones son más claras y las necesidades percibidas son más obvias. Por supuesto, un escritor técnico a menudo puede hacer un mejor trabajo con la gramática y la organización, pero apoyar a un solo equipo no es el mejor uso de un recurso limitado y especializado; no escala. Introdujo un incentivo perverso: conviértase en un proyecto importante y sus ingenieros de software no necesitarán escribir documentos. Desanimar a los ingenieros de escribir documentos resulta ser lo contrario de lo que quieras hacer.

Debido a que son un recurso limitado, los redactores técnicos generalmente deben enfocarse en tareas que los ingenieros de software *no* necesitan hacer como parte de sus deberes normales. Por lo general, esto implica escribir documentos que cruzan los límites de la API. Project Foo puede saber claramente qué documentación necesita Project Foo, pero probablemente tenga una idea menos clara de lo que necesita Project Bar. Un escritor técnico está mejor capacitado para actuar como una persona que no está familiarizada con el dominio. De hecho, es una de sus funciones fundamentales: desafiar las suposiciones que hace su equipo sobre la utilidad de su proyecto. Es una de las razones por las que muchos, si no la mayoría, de los escritores técnicos de ingeniería de software tienden a centrarse en este tipo específico de documentación API.

Conclusión

Google ha hecho grandes avances en el tratamiento de la calidad de la documentación durante la última década, pero para ser sincero, la documentación en Google aún no es un ciudadano de primera clase. En comparación, los ingenieros han aceptado gradualmente que las pruebas son necesarias para cualquier cambio de código, sin importar cuán pequeño sea. Además, las herramientas de prueba son robustas, variadas y están conectadas a un flujo de trabajo de ingeniería en varios puntos. La documentación no está arraigada casi al mismo nivel.

Para ser justos, no existe necesariamente la misma necesidad de abordar la documentación que con las pruebas. Las pruebas se pueden hacer atómicas (pruebas unitarias) y pueden seguir la forma y función prescritas. Los documentos, en su mayor parte, no pueden. Las pruebas se pueden automatizar y, a menudo, faltan esquemas para automatizar la documentación. Los documentos son necesariamente subjetivos; la calidad del documento no la mide el escritor, sino el lector, y a menudo de manera bastante asincrónica. Dicho esto, se reconoce que la documentación es importante y los procesos en torno al desarrollo de documentos están mejorando. En opinión de este autor, la calidad de la documentación en Google es mejor que en la mayoría de las tiendas de ingeniería de software.

Para cambiar la calidad de la documentación de ingeniería, los ingenieros, y toda la organización de ingeniería, deben aceptar que ellos son tanto el problema como la solución. En lugar de darse por vencidos ante el estado de la documentación, deben darse cuenta de que producir documentación de calidad es parte de su trabajo y les ahorra tiempo y esfuerzo a largo plazo. Para cualquier pieza de código que espere vivir más de unos pocos meses, los ciclos adicionales que dedica a documentar ese código no solo ayudarán a otros, sino que también lo ayudarán a mantener ese código.

TL; DR

- La documentación es muy importante con el tiempo y la escala.
- Los cambios en la documentación deben aprovechar el flujo de trabajo del desarrollador existente.
- Mantenga los documentos enfocados en un propósito.
- Escriba para su público, no para usted mismo.

Descripción general de las pruebas

*Escrito por Adam Bender
Editado por Tom Mansreck*

Las pruebas siempre han sido parte de la programación. De hecho, la primera vez que escribió un programa de computadora, casi con seguridad le envió algunos datos de muestra para ver si funcionaba como esperaba. Durante mucho tiempo, el estado del arte en las pruebas de software se parecía a un proceso muy similar, en gran parte manual y propenso a errores. Sin embargo, desde principios de la década de 2000, el enfoque de las pruebas de la industria del software ha evolucionado drásticamente para hacer frente al tamaño y la complejidad de los sistemas de software modernos. El centro de esa evolución ha sido la práctica de pruebas automatizadas impulsadas por desarrolladores.

Las pruebas automatizadas pueden evitar que los errores se escapen y afecten a sus usuarios. Cuanto más tarde en el ciclo de desarrollo se detecta un error, más caro es; exponencialmente en muchos casos.¹ Sin embargo, “atrinar bichos” es solo una parte de la motivación. Una razón igualmente importante por la que desea probar su software es para admitir la capacidad de cambio. Ya sea que esté agregando nuevas funciones, haciendo una refactorización centrada en el estado del código o realizando un rediseño más grande, las pruebas automatizadas pueden detectar errores rápidamente y esto hace posible cambiar el software con confianza.

Las empresas que pueden iterar más rápido pueden adaptarse más rápidamente a las tecnologías cambiantes, las condiciones del mercado y los gustos de los clientes. Si tiene una sólida práctica de pruebas, no debe temer el cambio: puede aceptarlo como una cualidad esencial del desarrollo de software. Cuanto más y más rápido desee cambiar sus sistemas, más necesitará una forma rápida de probarlos.

1 Ver “Prevención de defectos: reducción de costos y mejora de la calidad”.

El acto de escribir pruebas también mejora el diseño de sus sistemas. Como los primeros clientes de su código, una prueba puede brindarle mucha información sobre sus opciones de diseño. ¿Su sistema está demasiado acoplado a una base de datos? ¿La API es compatible con los casos de uso requeridos? ¿Su sistema maneja todos los casos extremos? Escribir pruebas automatizadas lo obliga a enfrentar estos problemas al principio del ciclo de desarrollo. Hacerlo generalmente conduce a un software más modular que permite una mayor flexibilidad más adelante.

Mucha tinta se ha derramado sobre el tema de las pruebas de software, y por una buena razón: para una práctica tan importante, hacerlo bien todavía parece ser un oficio misterioso para muchos. En Google, si bien hemos recorrido un largo camino, todavía enfrentamos problemas difíciles para que nuestros procesos se escalen de manera confiable en toda la empresa. En este capítulo, compartiremos lo que hemos aprendido para ayudar a promover la conversación.

¿Por qué escribimos pruebas?

Para comprender mejor cómo aprovechar al máximo las pruebas, comenzemos desde el principio. Cuando hablamos de pruebas automatizadas, ¿de qué estamos hablando realmente?

La prueba más simple se define por:

- Un solo comportamiento que está probando, generalmente un método o API que está llamando
- Una entrada específica, algún valor que pasa a la API
- Un resultado o comportamiento observable
- Un entorno controlado, como un único proceso aislado

Cuando ejecuta una prueba como esta, pasando la entrada al sistema y verificando la salida, sabrá si el sistema se comporta como espera. Tomados en conjunto, cientos o miles de pruebas simples (generalmente llamadas *Banco de pruebas*) puede decirle qué tan bien se ajusta todo su producto al diseño previsto y, lo que es más importante, cuándo no.

Crear y mantener un conjunto de pruebas saludable requiere un gran esfuerzo. A medida que crece la base de código, también lo hará el conjunto de pruebas. Comenzará a enfrentar desafíos como la inestabilidad y la lentitud. Si no se abordan estos problemas, se paralizará un conjunto de pruebas. Tenga en cuenta que las pruebas derivan su valor de la confianza que los ingenieros depositan en ellas. Si las pruebas se convierten en un sumidero de productividad, provocando constantemente trabajo e incertidumbre, los ingenieros perderán la confianza y comenzarán a encontrar soluciones alternativas. Un mal conjunto de pruebas puede ser peor que ningún conjunto de pruebas.

Además de empoderar a las empresas para que construyan excelentes productos rápidamente, las pruebas se están volviendo fundamentales para garantizar la seguridad de productos y servicios importantes en nuestras vidas. El software está más involucrado en nuestras vidas que nunca antes, y los defectos pueden causar

más que una pequeña molestia: pueden costar enormes cantidades de dinero, pérdida de propiedad o, lo peor de todo, pérdida de vidas.²

En Google, hemos determinado que las pruebas no pueden ser una ocurrencia tardía. Centrarse en la calidad y las pruebas es parte de cómo hacemos nuestro trabajo. Hemos aprendido, a veces dolorosamente, que no incorporar calidad a nuestros productos y servicios conduce inevitablemente a malos resultados. Como resultado, hemos integrado las pruebas en el corazón de nuestra cultura de ingeniería.

La historia del servidor web de Google

En los primeros días de Google, a menudo se suponía que las pruebas dirigidas por ingenieros tenían poca importancia. Los equipos dependían regularmente de personas inteligentes para obtener el software correcto. Algunos sistemas realizaron grandes pruebas de integración, pero en su mayoría fue el Salvaje Oeste. Un producto en particular pareció sufrir lo peor: se llamaba Google Web Server, también conocido como GWS.

GWS es el servidor web responsable de atender las consultas de la Búsqueda de Google y es tan importante para la Búsqueda de Google como lo es el control del tráfico aéreo para un aeropuerto. En 2005, a medida que el proyecto crecía en tamaño y complejidad, la productividad se redujo drásticamente. Los lanzamientos se estaban volviendo más problemáticos y tomaba más y más tiempo sacarlos. Los miembros del equipo tenían poca confianza al realizar cambios en el servicio y, a menudo, solo descubrían que algo andaba mal cuando las características dejaban de funcionar en producción. (En un momento, más del 80 % de los impulsos de producción contenían errores que afectaban al usuario y que tenían que revertirse).

Para abordar estos problemas, el líder tecnológico (TL) de GWS decidió instituir una política de pruebas automatizadas impulsadas por ingenieros. Como parte de esta política, se requería que todos los cambios de código nuevos incluyeran pruebas, y esas pruebas se ejecutarían continuamente. Dentro de un año de instituir esta política, el número de empujones de emergencia cayó a la mitad. Esta caída se produjo a pesar de que el proyecto experimentaba un número récord de nuevos cambios cada trimestre. Incluso ante un crecimiento y un cambio sin precedentes, las pruebas trajeron productividad y confianza renovadas a uno de los proyectos más críticos de Google. En la actualidad, GWS tiene decenas de miles de pruebas y se lanza casi todos los días con relativamente pocas fallas visibles para el cliente.

Los cambios en GWS marcaron un punto de inflexión para la cultura de pruebas en Google, ya que los equipos de otras partes de la empresa vieron los beneficios de las pruebas y adoptaron tácticas similares.

2 Ver “Fracaso en Dhahran”.

Una de las ideas clave que nos enseñó la experiencia de GWS fue que no se puede confiar únicamente en la capacidad del programador para evitar defectos en el producto. Incluso si cada ingeniero escribe solo un error ocasional, una vez que haya suficientes personas trabajando en el mismo proyecto, se verá abrumado por la lista cada vez mayor de defectos. Imagine un equipo hipotético de 100 personas cuyos ingenieros son tan buenos que cada uno escribe solo un error al mes. Colectivamente, este grupo de increíbles ingenieros todavía produce cinco errores nuevos cada día laboral. Peor aún, en un sistema complejo, corregir un error a menudo puede causar otro, ya que los ingenieros se adaptan a los errores conocidos y codifican a su alrededor.

Los mejores equipos encuentran formas de convertir la sabiduría colectiva de sus miembros en un beneficio para todo el equipo. Eso es exactamente lo que hacen las pruebas automatizadas. Después de que un ingeniero del equipo escribe una prueba, se agrega al grupo de recursos comunes disponibles para otros. Todos los demás miembros del equipo ahora pueden ejecutar la prueba y se beneficiarán cuando detecte un problema. Compare esto con un enfoque basado en la depuración, en el que cada vez que ocurre un error, un ingeniero debe pagar el costo de investigarlo con un depurador. El costo de los recursos de ingeniería es constante y fue la razón fundamental por la que GWS pudo cambiar su suerte.

Pruebas a la velocidad del desarrollo moderno

Los sistemas de software son cada vez más grandes y más complejos. Una aplicación o servicio típico de Google se compone de miles o millones de líneas de código. Utiliza cientos de bibliotecas o marcos y debe entregarse a través de redes poco confiables a un número creciente de plataformas que se ejecutan con un número incontable de configuraciones. Para empeorar las cosas, las nuevas versiones se envían a los usuarios con frecuencia, a veces varias veces al día. Esto está muy lejos del mundo del software empaquetado que vio actualizaciones solo una o dos veces al año.

La capacidad de los humanos para validar manualmente cada comportamiento en un sistema no ha podido seguir el ritmo de la explosión de funciones y plataformas en la mayoría del software. Imagínese lo que se necesitaría para probar manualmente todas las funciones de la Búsqueda de Google, como encontrar vuelos, horarios de películas, imágenes relevantes y, por supuesto, resultados de búsqueda web (ver [Figura 11-1](#)). Incluso si puede determinar cómo resolver ese problema, debe multiplicar esa carga de trabajo por cada idioma, país y dispositivo que la Búsqueda de Google debe admitir, y no olvide verificar cosas como la accesibilidad y la seguridad. Intentar evaluar la calidad del producto pidiéndoles a los humanos que interactúen manualmente con cada característica simplemente no escala. Cuando se trata de pruebas, hay una respuesta clara: automatización.

Google Microphone Search

All Flights Maps Images Shopping More Settings Tools

About 15,500,000 results (1.25 seconds)

Flights from San Francisco, CA (SFO) to London, United Kingdom (all airports) Sponsored

www.google.com/flights

From	To
San Francisco, CA (SFO)	London, United Kingdom (all airports)
Sun, September 15	Mon, September 30

Airline	Flight Time	Nonstop	Price
Multiple airlines	10h 10m+	Connecting	from \$353
British Airways	13h 35m+	Connecting	from \$365
Multiple airlines	10h 15m	Nonstop	from \$422
United	10h 30m	Nonstop	from \$422
Delta	10h 15m	Nonstop	from \$628
Virgin Atlantic	10h 15m	Nonstop	from \$628
Air France	10h 15m	Nonstop	from \$629
KLM	10h 15m	Nonstop	from \$629
Lufthansa	10h 30m	Nonstop	from \$692
Austrian	10h 30m	Nonstop	from \$692
Brussels Airlines	10h 30m	Nonstop	from \$692
Norwegian Air UK	10h 10m	Nonstop	from \$760
American	10h 25m	Nonstop	from \$939
British Airways	10h 25m	Nonstop	from \$939
Iberia	10h 25m	Nonstop	from \$939
Other airlines	10h 15m+	Connecting	from \$415

[Search flights](#)

Cheap Flights from San Francisco to London from \$347 - KAYAK
<https://www.kayak.com/Flights/Worldwide/Europe/UnitedKingdom/England>
 Fly from San Francisco to London on Air Canada from \$347, Finnair from \$348, Lufthansa from \$363...
 Search ... SFO - LHR San Francisco - London Heathrow ...

How does KAYAK find such low flight prices? ▼

How can Hacker Fares save me money? ▼

Does KAYAK query more flight providers than competitors? ▼

▼ Show more

Google Microphone Search

All Shopping News Images Videos More Settings Tools

About 32,300 results (0.69 seconds)

Spaceballs / On TV soon

All times are in Pacific Time

Date	Channel
Today, 9:25 AM	Starz Comedy (West)
Today, 5 PM	Starz Comedy (East)
Today, 8 PM	Starz Comedy (West)
Tomorrow, 10:30 PM	Encore (East)
Fri, 1/31, 1:30 AM	Encore (West)

[Feedback](#)

www.fandango.com/spaceballs-836/movie-overview ▾
Spaceballs | Fandango
 ★★★★ ½ Rating: 83% - 434,978 votes
 Showtimes are not available near 95101, watch Spaceballs anytime, anywhere with FandangoNow. Watch Spaceballs anytime, anywhere with FandangoNow.

www.fandango.com/spaceballs-836/movie-times ▾
Spaceballs Times - Movie Tickets + Showtimes | Fandango
 ★★★★ ½ Rating: 83% - 434,979 votes
 SPACEBALLS, 1987, Park Circus/MGM, 96 min. Dir. Mel Brooks. Bill Pullman, John Candy and Rick Moranis head the cast in Mel Brooks' hilarious riff on STAR ...



Spaceballs
PG 1987 - Fantasy/Parody film - 1h 36m

[Play trailer on YouTube](#)

<small>83%</small> Fandango	<small>7.1/10</small> IMDb	<small>46%</small> Metacritic
--------------------------------	-------------------------------	----------------------------------

92% liked this movie
Google users

In a distant galaxy, planet Spaceball has depleted its air supply, leaving its citizens reliant on a product called "Perri-Air." In desperation, Spaceball's leader President Skroob (Mel Brooks) orders the evil Dark Helmet (Rick Moranis) to kidnap Princess Vespa (Daphne Zuniga) of oxygen-rich Druidia... [MORE](#) ▾

Figura 11-1. Capturas de pantalla de dos complejos resultados de búsqueda de Google

Escribir, ejecutar, reaccionar

En su forma más pura, la automatización de pruebas consta de tres actividades: escribir pruebas, ejecutar pruebas y reaccionar ante fallas en las pruebas. Una prueba automatizada es un pequeño fragmento de código, generalmente una sola función o método, que llama a una parte aislada de un sistema más grande que desea probar. El código de prueba establece un entorno esperado, llama al sistema, generalmente con una entrada conocida, y verifica el resultado. Algunas de las pruebas son muy pequeñas y ejercen una sola ruta de código; otros son mucho más grandes y pueden involucrar sistemas completos, como un sistema operativo móvil o un navegador web.

Ejemplo 11-1 presenta una prueba deliberadamente simple en Java que no utiliza marcos ni bibliotecas de prueba. No es así como escribiría un conjunto de pruebas completo, pero en esencia, cada prueba automatizada se parece a este ejemplo muy simple.

Ejemplo 11-1. Una prueba de ejemplo

```
// Verifica que una clase Calculadora pueda manejar resultados negativos. público
vacioprincipal(Cuerda[] argumentos) {
    calculadora calculadora=nuevoCalculadora(); En t
    Resultado Esperado= -3;
    En tresultado actual=calculadora.sustraer(2,5); //Dado 2, resta 5. afirmar
    Resultado Esperado==resultado actual);
}
```

A diferencia de los procesos de control de calidad de antaño, en los que las salas de evaluadores de software dedicados analizaban minuciosamente las nuevas versiones de un sistema, ejerciendo todos los comportamientos posibles, los ingenieros que construyen sistemas hoy en día desempeñan un papel activo e integral en la escritura y ejecución de pruebas automatizadas para su propio código. Incluso en empresas donde QA es una organización prominente, las pruebas escritas por desarrolladores son comunes. A la velocidad y escala a la que se desarrollan los sistemas actuales, la única forma de mantenerse al día es compartir el desarrollo de las pruebas con todo el personal de ingeniería.

Por supuesto, escribir exámenes es diferente de escribir *buenas pruebas*. Puede ser bastante difícil capacitar a decenas de miles de ingenieros para que escriban buenas pruebas. Discutiremos lo que hemos aprendido acerca de escribir buenas pruebas en los capítulos siguientes.

Escribir pruebas es solo el primer paso en el proceso de pruebas automatizadas. Después de haber escrito las pruebas, debe ejecutarlas. Frecuentemente. En esencia, las pruebas automatizadas consisten en repetir la misma acción una y otra vez, y solo requieren la atención humana cuando algo se rompe. Discutiremos esta integración continua (CI) y las pruebas en [capítulo 23](#). Al expresar las pruebas como código en lugar de una serie manual de pasos, podemos ejecutarlas cada vez que cambia el código, fácilmente miles de veces al día. A diferencia de los probadores humanos, las máquinas nunca se cansan ni se aburren.

Otro beneficio de tener pruebas expresadas como código es que es fácil modularizarlas para su ejecución en varios entornos. Probando el comportamiento de Gmail en Firefox

no requiere más esfuerzo que hacerlo en Chrome, siempre que tenga configuraciones para ambos sistemas.³ La ejecución de pruebas para una interfaz de usuario (UI) en japonés o alemán se puede realizar utilizando el mismo código de prueba que para el inglés.

Los productos y servicios en desarrollo activo inevitablemente experimentarán fallas en las pruebas. Lo que realmente hace que un proceso de prueba sea efectivo es cómo aborda las fallas de prueba. Permitir que las pruebas fallidas se acumulen rápidamente anula cualquier valor que estuvieran brindando, por lo que es imperativo no dejar que eso suceda. Los equipos que priorizan la reparación de una prueba rota a los pocos minutos de una falla pueden mantener un alto nivel de confianza y un rápido aislamiento de fallas y, por lo tanto, obtener más valor de sus pruebas.

En resumen, una cultura saludable de pruebas automatizadas alienta a todos a compartir el trabajo de escribir pruebas. Tal cultura también asegura que las pruebas se realicen con regularidad. Por último, y quizás lo más importante, pone énfasis en reparar rápidamente las pruebas rotas para mantener una alta confianza en el proceso.

Beneficios del código de prueba

Para los desarrolladores que provienen de organizaciones que no tienen una fuerte cultura de pruebas, la idea de escribir pruebas como un medio para mejorar la productividad y la velocidad puede parecer antitética. Después de todo, el acto de escribir pruebas puede llevar tanto tiempo (¡si no más!) que implementar una función en primer lugar. Por el contrario, en Google descubrimos que invertir en pruebas de software proporciona varios beneficios clave para la productividad de los desarrolladores:

Menos depuración

Como era de esperar, el código probado tiene menos defectos cuando se envía. Críticamente, también tiene menos defectos a lo largo de su existencia; la mayoría de ellos serán capturados antes de que se envíe el código. Se espera que una pieza de código en Google se modifique docenas de veces durante su vida útil. Será cambiado por otros equipos e incluso sistemas automatizados de mantenimiento de código. Una prueba escrita una vez continúa pagando dividendos y evitando costosos defectos y molestas sesiones de depuración a lo largo de la vida útil del proyecto. Los cambios en un proyecto, o las dependencias de un proyecto, que rompen una prueba pueden ser rápidamente detectados por la infraestructura de prueba y revertidos antes de que el problema pase a producción.

Mayor confianza en los cambios.

Todos los cambios de software. Los equipos con buenas pruebas pueden revisar y aceptar cambios en su proyecto con confianza porque todos los comportamientos importantes de su proyecto se verifican continuamente. Dichos proyectos fomentan la refactorización. Cambios que refactorizan

³ ¡Conseguir el comportamiento correcto en diferentes navegadores e idiomas es una historia diferente! Pero, idealmente, el final-La experiencia del usuario debe ser la misma para todos.

El código mientras se preserva el comportamiento existente (idealmente) no debería requerir cambios en las pruebas existentes.

Documentación mejorada

La documentación del software es notoriamente poco confiable. Desde requisitos obsoletos hasta casos extremos que faltan, es común que la documentación tenga una relación tenue con el código. Las pruebas claras y enfocadas que ejercitan un comportamiento a la vez funcionan como documentación ejecutable. Si desea saber qué hace el código en un caso particular, mire la prueba para ese caso. Aún mejor, cuando los requisitos cambian y el nuevo código rompe una prueba existente, recibimos una señal clara de que la "documentación" ahora está desactualizada. Tenga en cuenta que las pruebas funcionan mejor como documentación solo si se tiene cuidado de mantenerlas claras y concisas.

Revisões más simples

Todo el código en Google es revisado por al menos otro ingeniero antes de que pueda enviarse (ver [Capítulo 9](#) para más detalles). Un revisor de código dedica menos esfuerzo a verificar que el código funcione como se espera si la revisión del código incluye pruebas exhaustivas que demuestran la corrección del código, los casos límite y las condiciones de error. En lugar del tedioso esfuerzo necesario para recorrer mentalmente cada caso a través del código, el revisor puede verificar que cada caso haya pasado la prueba.

Diseño reflexivo

Escribir pruebas para código nuevo es un medio práctico de ejercitarse en el diseño API del propio código. Si el nuevo código es difícil de probar, a menudo se debe a que el código que se está probando tiene demasiadas responsabilidades o dependencias difíciles de administrar. El código bien diseñado debe ser modular, evitando el acoplamiento estrecho y centrándose en responsabilidades específicas. La solución temprana de problemas de diseño a menudo significa menos reelaboración posterior.

Lanzamientos rápidos y de alta calidad.

Con un conjunto de pruebas automatizado saludable, los equipos pueden lanzar nuevas versiones de su aplicación con confianza. Muchos proyectos en Google lanzan una nueva versión a producción todos los días, incluso proyectos grandes con cientos de ingenieros y miles de cambios de código enviados todos los días. Esto no sería posible sin las pruebas automatizadas.

Diseño de un conjunto de pruebas

Hoy, Google opera a gran escala, pero no siempre hemos sido tan grandes, y las bases de nuestro enfoque se establecieron hace mucho tiempo. A lo largo de los años, a medida que nuestra base de código ha crecido, hemos aprendido mucho sobre cómo abordar el diseño y la ejecución de un conjunto de pruebas, a menudo cometiendo errores y limpiando después.

Una de las lecciones que aprendimos bastante pronto es que los ingenieros preferían escribir pruebas más grandes a escala del sistema, pero que estas pruebas eran más lentas, menos confiables y más difíciles de depurar que las pruebas más pequeñas. Los ingenieros, hartos de depurar las pruebas a escala del sistema,

se preguntaron: "¿Por qué no podemos probar un servidor a la vez?" o "¿Por qué necesitamos probar un servidor completo a la vez? Podríamos probar módulos más pequeños individualmente". Eventualmente, el deseo de reducir el dolor llevó a los equipos a desarrollar pruebas cada vez más pequeñas, que resultaron ser más rápidas, más estables y, en general, menos dolorosas.

Esto generó mucha discusión en la empresa sobre el significado exacto de "pequeño". ¿La prueba de unidad de media pequeña? ¿Qué pasa con las pruebas de integración, qué tamaño son esas? Hemos llegado a la conclusión de que hay dos dimensiones distintas para cada caso de prueba: tamaño y alcance. El tamaño se refiere a los recursos que se requieren para ejecutar un caso de prueba: elementos como la memoria, los procesos y el tiempo. El alcance se refiere a las rutas de código específicas que estamos verificando. Tenga en cuenta que ejecutar una línea de código es diferente de verificar que funcionó como se esperaba. El tamaño y el alcance son conceptos interrelacionados pero distintos.

Tamaño de prueba

En Google, clasificamos cada una de nuestras pruebas en un tamaño y alentamos a los ingenieros a escribir siempre la prueba más pequeña posible para una funcionalidad determinada. El tamaño de una prueba no está determinado por su número de líneas de código, sino por cómo se ejecuta, qué puede hacer y cuántos recursos consume. De hecho, en algunos casos, nuestras definiciones de pequeño, mediano y grande en realidad están codificadas como restricciones que la infraestructura de pruebas puede imponer en una prueba. Entraremos en detalles en un momento, pero en resumen, *pequeñas pruebas* ejecutar en un solo proceso, *pruebas medianas* ejecutar en una sola máquina, y *grandes pruebas* correr donde quieran, como se demuestra en [Figura 11-2](#).⁴

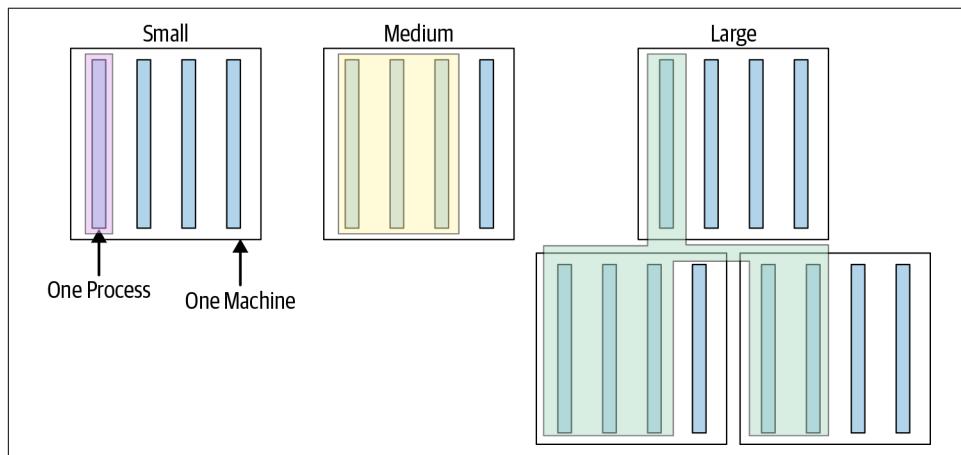


Figura 11-2. Tamaños de prueba

4 Técnicamente, tenemos cuatro tamaños de prueba en Google: pequeña, mediana, grande y *enorme*. La diferencia interna entre grande y enorme es en realidad sutil e histórico; por lo tanto, en este libro, la mayoría de las descripciones de grande se aplican a nuestra noción de enorme.

Hacemos esta distinción, a diferencia de la "unidad" o "integración" más tradicional, porque las cualidades más importantes que queremos de nuestro conjunto de pruebas son la velocidad y el determinismo, independientemente del alcance de la prueba. Las pruebas pequeñas, independientemente del alcance, casi siempre son más rápidas y más deterministas que las pruebas que involucran más infraestructura o consumen más recursos. Poner restricciones en pruebas pequeñas hace que la velocidad y el determinismo sean mucho más fáciles de lograr. A medida que crece el tamaño de las pruebas, muchas de las restricciones se relajan. Las pruebas medianas tienen más flexibilidad pero también más riesgo de no determinismo. Las pruebas más grandes se guardan solo para los escenarios de prueba más complejos y difíciles. Echemos un vistazo más de cerca a las restricciones exactas impuestas en cada tipo de prueba.

Pequeñas pruebas

Las pruebas pequeñas son las más limitadas de los tres tamaños de prueba. La restricción principal es que las pruebas pequeñas deben ejecutarse en un solo proceso. En muchos idiomas, restringimos esto aún más para decir que deben ejecutarse en un solo hilo. Esto significa que el código que realiza la prueba debe ejecutarse en el mismo proceso que el código que se está probando. No puede ejecutar un servidor y tener un proceso de prueba separado que se conecte a él. También significa que no puede ejecutar un programa de terceros, como una base de datos, como parte de su prueba.

Las otras limitaciones importantes de las pruebas pequeñas son que no se les permite dormir, realizar operaciones de E/S,⁵⁰ realizar cualquier otro bloqueo de llamadas. Esto significa que las pruebas pequeñas no pueden acceder a la red o al disco. Probar el código que se basa en este tipo de operaciones requiere el uso de dobles de prueba (ver[Capítulo 13](#)) para reemplazar la dependencia pesada con una dependencia ligera en proceso.

El propósito de estas restricciones es garantizar que las pruebas pequeñas no tengan acceso a las principales fuentes de lentitud o no determinismo de las pruebas. Una prueba que se ejecuta en un solo proceso y nunca hace llamadas de bloqueo puede ejecutarse de manera efectiva tan rápido como la CPU puede manejar. Es difícil (pero ciertamente no imposible) hacer accidentalmente que una prueba de este tipo sea lenta o no determinista. Las limitaciones de las pruebas pequeñas proporcionan una caja de arena que evita que los ingenieros se disparen en el pie.

Estas restricciones pueden parecer excesivas al principio, pero considere un conjunto modesto de un par de cientos de pequeños casos de prueba que se ejecutan durante todo el día. Si incluso algunos de ellos fallan de manera no determinista (a menudo llamado[pruebas escamosas](#)), rastrear la causa se convierte en una grave pérdida de productividad. A la escala de Google, tal problema podría paralizar nuestra infraestructura de pruebas.

En Google, alentamos a los ingenieros a que traten de escribir pruebas pequeñas siempre que sea posible, independientemente del alcance de la prueba, porque mantiene todo el conjunto de pruebas funcionando de manera rápida y confiable. Para obtener más información sobre pruebas pequeñas versus pruebas unitarias, consulte[Capítulo 12](#).

5 Hay un pequeño margen de maniobra en esta política. Las pruebas pueden acceder a un sistema de archivos si utilizan un sistema hermético e integrado. implementación de la memoria.

Pruebas medianas

Las restricciones impuestas a las pruebas pequeñas pueden ser demasiado restrictivas para muchos tipos de pruebas interesantes. El siguiente peldaño en la escala de tamaños de prueba es la prueba mediana. Las pruebas medianas pueden abarcar múltiples procesos, usar subprocessos y pueden realizar llamadas de bloqueo, incluidas llamadas de red, paraservidor local. La única restricción restante es que las pruebas medianas no pueden realizar llamadas de red a ningún sistema que no sea servidor local. En otras palabras, la prueba debe estar contenida dentro de una sola máquina.

La capacidad de ejecutar múltiples procesos abre muchas posibilidades. Por ejemplo, podría ejecutar una instancia de base de datos para validar que el código que está probando se integra correctamente en una configuración más realista. O puede probar una combinación de interfaz de usuario web y código de servidor. Las pruebas de aplicaciones web a menudo involucran herramientas como [WebDriver](#) que inician un navegador real y lo controlan de forma remota a través del proceso de prueba.

Desafortunadamente, con una mayor flexibilidad viene un mayor potencial para que las pruebas se vuelvan lentas y no deterministas. Las pruebas que abarcan procesos o que pueden realizar llamadas de bloqueo dependen de que el sistema operativo y los procesos de terceros sean rápidos y deterministas, algo que no podemos garantizar en general. Las pruebas medianas aún brindan un poco de protección al evitar el acceso a máquinas remotas a través de la red, que es, con mucho, la mayor fuente de lentitud y no determinismo en la mayoría de los sistemas. Aún así, al escribir pruebas medianas, la "seguridad" está desactivada y los ingenieros deben ser mucho más cuidadosos.

Pruebas grandes

Por último, tenemos grandes pruebas. Las pruebas grandes eliminan la restricción impuesta a las pruebas medianas, lo que permite que la prueba y el sistema que se está probando abarquen varias máquinas. Por ejemplo, la prueba podría ejecutarse en un sistema en un clúster remoto.

Como antes, una mayor flexibilidad conlleva un mayor riesgo. Tener que lidiar con un sistema que abarca varias máquinas y la red que las conecta aumenta significativamente la posibilidad de lentitud y no determinismo en comparación con la ejecución en una sola máquina. En su mayoría, reservamos las pruebas grandes para las pruebas de extremo a extremo del sistema completo que tienen más que ver con la validación de la configuración que con las piezas de código, y para las pruebas de componentes heredados para los que es imposible usar dobles de prueba. Hablaremos más sobre casos de uso para pruebas grandes [en capítulo 14](#). Los equipos de Google con frecuencia aislarán sus pruebas grandes de sus pruebas pequeñas o medianas, ejecutándolas solo durante el proceso de compilación y lanzamiento para no afectar el flujo de trabajo del desarrollador.

Estudio de caso: las pruebas de escamas son caras

Si tiene unos cuantos miles de pruebas, cada una con un poquito de no determinismo, ejecutándose todo el día, de vez en cuando una probablemente fallará (flake). A medida que crece el número de pruebas, estadísticamente también lo hará el número de escamas. Si cada prueba tiene incluso un 0,1% de falla cuando no debería, y ejecuta 10,000 pruebas por día, estará investigando 10 fallas por día. Cada investigación le quita tiempo a algo más productivo que su equipo podría estar haciendo.

En algunos casos, puede limitar el impacto de las pruebas irregulares volviéndolas a ejecutar automáticamente cuando fallan. Esto es efectivamente intercambiar ciclos de CPU por tiempo de ingeniería. En niveles bajos de descamación, esta compensación tiene sentido. Solo tenga en cuenta que volver a ejecutar una prueba solo retrasa la necesidad de abordar la causa raíz de la descamación.

Si la descamación de las pruebas continúa creciendo, experimentará algo mucho peor que la pérdida de productividad: una pérdida de confianza en las pruebas. No es necesario investigar muchas fallas antes de que un equipo pierda la confianza en el conjunto de pruebas. Después de que eso suceda, los ingenieros dejarán de reaccionar ante las fallas de las pruebas, eliminando cualquier valor proporcionado por el conjunto de pruebas. Nuestra experiencia sugiere que a medida que se acerca al 1% de descamación, las pruebas comienzan a perder valor. En Google, nuestra tasa de escamas ronda el 0,15 %, lo que implica miles de escamas cada día. Luchamos arduamente para mantener las escamas bajo control, lo que incluye invertir activamente horas de ingeniería para repararlas.

En la mayoría de los casos, las escamas aparecen debido a un comportamiento no determinista en las pruebas mismas. El software proporciona muchas fuentes de no determinismo: tiempo de reloj, programación de subprocessos, latencia de red y más. Aprender a aislar y estabilizar los efectos de la aleatoriedad no es fácil. A veces, los efectos están vinculados a preocupaciones de bajo nivel como interrupciones de hardware o motores de renderizado del navegador. Una buena infraestructura de pruebas automatizadas debería ayudar a los ingenieros a identificar y mitigar cualquier comportamiento no determinista.

Propiedades comunes a todos los tamaños de prueba

Todas las pruebas deben esforzarse por ser herméticas: una prueba debe contener toda la información necesaria para configurar, ejecutar y desmantelar su entorno. Las pruebas deben suponer lo menos posible sobre el entorno exterior, como el orden en que se ejecutan las pruebas. Por ejemplo, no deben depender de una base de datos compartida. Esta restricción se vuelve más desafiante con pruebas más grandes, pero aún se debe hacer un esfuerzo para garantizar el aislamiento.

Una prueba debe contener *solo* la información necesaria para el ejercicio de la conducta de que se trata. Mantener las pruebas claras y simples ayuda a los revisores a verificar que el código hace lo que dice que hace. El código claro también ayuda a diagnosticar fallas cuando fallan. Nos gusta decir que "una prueba debe ser obvia tras la inspección". Debido a que no hay pruebas para las pruebas en sí mismas, requieren una revisión manual como una verificación importante de la corrección. Como corolario de esto, también **desalentar enfáticamente el uso del estado de flujo de control**.

mentos como condicionales y bucles en una prueba. Los flujos de prueba más complejos corren el riesgo de contener errores y hacen que sea más difícil determinar la causa de una falla en la prueba.

Recuerde que las pruebas a menudo se revisan solo cuando algo se rompe. Cuando lo llamen para arreglar una prueba rota que nunca antes había visto, estará agradecido de que alguien se haya tomado el tiempo para que sea fácil de entender. El código se lee mucho más de lo que se escribe, ¡así que asegúrese de escribir la prueba que le gustaría leer!

Tamaños de prueba en la práctica. Tener definiciones precisas de los tamaños de las pruebas nos ha permitido crear herramientas para aplicarlas. La aplicación nos permite escalar nuestros conjuntos de pruebas y seguir ofreciendo ciertas garantías sobre la velocidad, la utilización de recursos y la estabilidad. La medida en que estas definiciones se aplican en Google varía según el idioma. Por ejemplo, ejecutamos todas las pruebas de Java con un administrador de seguridad personalizado que hará que todas las pruebas etiquetadas como pequeñas fallen si intentan hacer algo prohibido, como establecer una conexión de red.

Alcance de prueba

Aunque en Google ponemos mucho énfasis en el tamaño de la prueba, otra propiedad importante a considerar es el alcance de la prueba. El alcance de la prueba se refiere a la cantidad de código que se valida mediante una prueba determinada. Las pruebas de alcance limitado (comúnmente llamadas "pruebas unitarias") están diseñadas para validar la lógica en una parte pequeña y enfocada de la base de código, como una clase o método individual. Pruebas de mediano alcance (comúnmente llamadas *pruebas de integración*) están diseñados para verificar las interacciones entre un pequeño número de componentes; por ejemplo, entre un servidor y su base de datos. Pruebas de gran alcance (comúnmente denominadas con nombres como *pruebas funcionales, de extremo a extremo* o *pruebas del sistema*) están diseñados para validar la interacción de varias partes distintas del sistema, o comportamientos emergentes que no se expresan en una sola clase o método.

Es importante tener en cuenta que cuando hablamos de pruebas unitarias como de alcance limitado, nos referimos al código que se está *validado*, no el código que está siendo *ejecutado*. Es bastante común que una clase tenga muchas dependencias u otras clases a las que se refiere, y estas dependencias se invocarán naturalmente al probar la clase de destino. Aunque algunos **otras estrategias de prueba** haga un uso intensivo de pruebas dobles (falsas o simulacros) para evitar la ejecución de código fuera del sistema bajo prueba, en Google, preferimos mantener las dependencias reales en su lugar cuando sea factible hacerlo. [Capítulo 13](#) trata este tema con más detalle.

Las pruebas de alcance limitado tienden a ser pequeñas y las pruebas de alcance amplio tienden a ser medianas o grandes, pero no siempre es así. Por ejemplo, es posible escribir una prueba de amplio alcance de un extremo del servidor que cubra todo su análisis normal, validación de solicitudes y lógica comercial, que sin embargo es pequeña porque usa dobles para reemplazar todas las dependencias fuera del proceso, como una base de datos o un sistema de archivos. De manera similar, es posible escribir una prueba de alcance limitado de un solo método que debe ser de tamaño mediano. Para

Por ejemplo, los marcos web modernos a menudo combinan HTML y JavaScript, y probar un componente de interfaz de usuario como un selector de fechas a menudo requiere ejecutar un navegador completo, incluso para validar una sola ruta de código.

Así como alentamos las pruebas de tamaño más pequeño, en Google también alentamos a los ingenieros a escribir pruebas de alcance más limitado. Como pauta muy aproximada, tendemos a tener una combinación de alrededor del 80 % de nuestras pruebas que son pruebas unitarias de alcance limitado que validan la mayoría de nuestra lógica comercial; 15% pruebas de integración de mediano alcance que validan las interacciones entre dos o más componentes; y 5% pruebas de extremo a extremo que validan todo el sistema.[Figura 11-3](#)muestra cómo podemos visualizar esto como una pirámide.

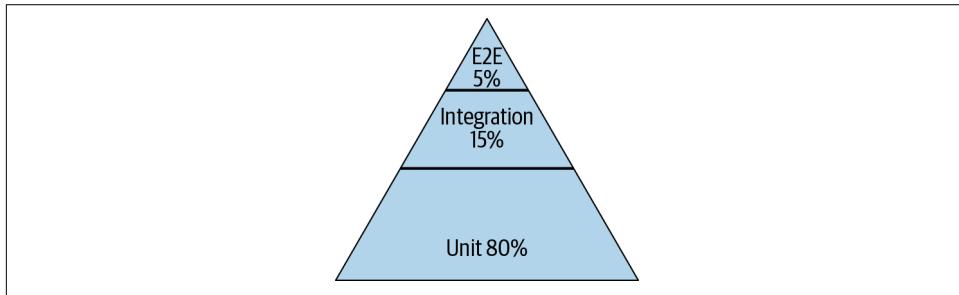


Figura 11-3. la versión de Google de la pirámide de prueba de Mike Cohn; los porcentajes son por recuento de casos de prueba, y la combinación de cada equipo será un poco diferente

Las pruebas unitarias forman una base excelente porque son rápidas, estables y limitan drásticamente el alcance y reducen la carga cognitiva requerida para identificar todos los comportamientos posibles que tiene una clase o función. Además, hacen que el diagnóstico de fallas sea rápido e indoloro. Dos antipatrones a tener en cuenta son el "cono de helado" y el "reloj de arena", como se ilustra en[Figura 11-4](#).

Con el cono de helado, los ingenieros escriben muchas pruebas de extremo a extremo, pero pocas pruebas de integración o unitarias. Dichos conjuntos tienden a ser lentos, poco confiables y difíciles de trabajar. Este patrón a menudo aparece en proyectos que comienzan como prototipos y se apresuran rápidamente a la producción, sin detenerse nunca para abordar la deuda de prueba.

El reloj de arena implica muchas pruebas de extremo a extremo y muchas pruebas unitarias, pero pocas pruebas de integración. No es tan malo como el cono de helado, pero aun así da como resultado muchas fallas de prueba de extremo a extremo que podrían haberse detectado más rápido y más fácilmente con un conjunto de pruebas de alcance medio. El patrón de reloj de arena se produce cuando el acoplamiento estrecho dificulta la creación de instancias de dependencias individuales de forma aislada.

6Mike Cohn, *Tener éxito con Agile: desarrollo de software usando Scrum*(Nueva York: Addison-Wesley Profesional, 2009).

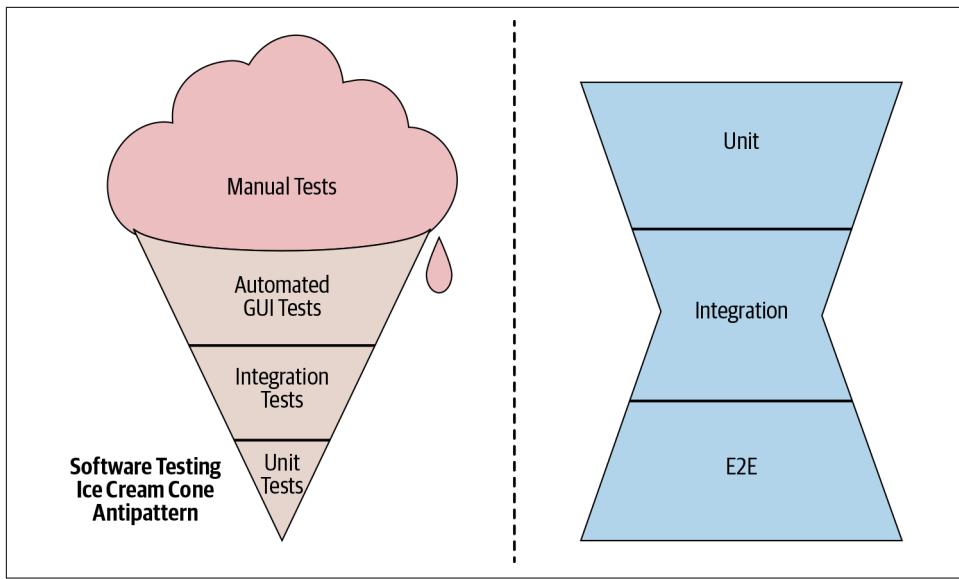


Figura 11-4. Conjunto de pruebas antipatrones

Nuestra combinación recomendada de pruebas está determinada por nuestros dos objetivos principales: productividad de ingeniería y confianza en el producto. Favorecer las pruebas unitarias nos da una gran confianza rápidamente y al principio del proceso de desarrollo. Las pruebas más grandes actúan como controles de cordura a medida que se desarrolla el producto; no deben verse como un método principal para atrapar errores.

Al considerar su propia mezcla, es posible que desee un equilibrio diferente. Si enfatiza las pruebas de integración, es posible que descubra que sus conjuntos de pruebas tardan más en ejecutarse pero detectan más problemas entre los componentes. Cuando enfatiza las pruebas unitarias, sus conjuntos de pruebas pueden completarse muy rápidamente y detectarán muchos errores lógicos comunes. Pero las pruebas unitarias no pueden verificar las interacciones entre los componentes, como **un contrato entre dos sistemas desarrollados por diferentes equipos**. Un buen conjunto de pruebas contiene una combinación de diferentes tamaños y alcances de prueba que son apropiados para las realidades organizativas y arquitectónicas locales.

La regla de Beyoncé

A menudo nos preguntan, cuando entrenamos a nuevos empleados, ¿qué comportamientos o propiedades realmente necesitan ser probados? La respuesta directa es: prueba todo lo que no quieras romper. En otras palabras, si quiere estar seguro de que un sistema exhibe un comportamiento particular, la única manera de estar seguro es escribir una prueba automatizada para él. Esto incluye todos los sospechosos habituales, como el rendimiento de las pruebas, la corrección del comportamiento, la accesibilidad y la seguridad. También incluye propiedades menos obvias, como probar cómo un sistema maneja las fallas.

Tenemos un nombre para esta filosofía general: la llamamos laRegla de Beyoncé. Sucintamente, se puede decir de la siguiente manera: "Si te gustó, entonces deberías haberlo probado". La Regla de Beyoncé a menudo es invocada por equipos de infraestructura que son responsables de realizar cambios en todo el código base. Si los cambios de infraestructura no relacionados pasan todas sus pruebas pero aún así rompen el producto de su equipo, está obligado a arreglarlo y agregar pruebas adicionales.

Prueba de falla

Una de las situaciones más importantes que un sistema debe tener en cuenta es la falla. El fracaso es inevitable, pero esperar una catástrofe real para saber qué tan bien responde un sistema a una catástrofe es una receta para el dolor. En lugar de esperar una falla, escriba pruebas automatizadas que simulen tipos comunes de fallas. Esto incluye simular excepciones o errores en las pruebas unitarias e injectar errores o latencia de llamada a procedimiento remoto (RPC) en las pruebas de integración y de extremo a extremo. También puede incluir interrupciones mucho más grandes que afectan la red de producción real utilizando técnicas comoIngeniería del caos. Una respuesta predecible y controlada a condiciones adversas es un sello distintivo de un sistema confiable.

Una nota sobre la cobertura del código

La cobertura de código es una medida de qué líneas de código de característica se ejercen mediante qué pruebas. Si tiene 100 líneas de código y sus pruebas ejecutan 90 de ellas, tiene una cobertura de código del 90 %.⁷ La cobertura de código a menudo se presenta como la métrica estándar de oro para comprender la calidad de la prueba, y eso es algo desafortunado. Es posible ejercitar muchas líneas de código con unas pocas pruebas, sin verificar nunca que cada línea esté haciendo algo útil. Esto se debe a que la cobertura de código solo mide que se invocó una línea, no lo que sucedió como resultado. (Recomendamos solo medir la cobertura de pruebas pequeñas para evitar la inflación de cobertura que ocurre al ejecutar pruebas más grandes).

Un problema aún más insidioso con la cobertura de código es que, al igual que otras métricas, rápidamente se convierte en un objetivo en sí mismo. Es común que los equipos establezcan una barra para la cobertura de código esperada, por ejemplo, 80%. Al principio, eso suena eminentemente razonable; seguramente querrás tener al menos esa cobertura. En la práctica, lo que sucede es que en lugar de tratar el 80% como un piso, los ingenieros lo tratan como un techo. Pronto, los cambios comienzan a aterrizar con una cobertura de no más del 80 %. Después de todo, ¿por qué hacer más trabajo del que requiere la métrica?

7 Tenga en cuenta que hay diferentes tipos de cobertura (línea, camino, ramal, etc.), y cada uno dice algo diferente sobre qué código ha sido probado. En este ejemplo simple, se utiliza cobertura de línea.

Una mejor manera de abordar la calidad de su conjunto de pruebas es pensar en los comportamientos que se prueban. ¿Confía en que todo lo que sus clientes esperan que funcione funcionará? ¿Se siente seguro de poder detectar cambios importantes en sus dependencias? ¿Son sus pruebas estables y confiables? Preguntas como estas son una forma más holística de pensar en un conjunto de pruebas. Cada producto y equipo va a ser diferente; algunos tendrán interacciones difíciles de probar con el hardware, algunos involucran conjuntos de datos masivos. Tratando de responder a la pregunta "¿tenemos suficientes pruebas?" con un solo número ignora mucho contexto y es poco probable que sea útil. La cobertura de código puede proporcionar una idea del código no probado, pero no reemplaza el pensamiento crítico sobre qué tan bien se prueba su sistema.

Pruebas a escala de Google

Gran parte de la guía hasta este punto se puede aplicar a bases de código de casi cualquier tamaño. Sin embargo, deberíamos dedicar algún tiempo a probar lo que hemos aprendido a gran escala. Para comprender cómo funcionan las pruebas en Google, necesita comprender nuestro entorno de desarrollo, cuyo hecho más importante es que la mayor parte del código de Google se guarda en un repositorio único y monolítico ([monorepo](#)). Casi todas las líneas de código de cada producto y servicio que operamos se almacenan en un solo lugar. Actualmente tenemos más de dos mil millones de líneas de código en el repositorio.

El código base de Google experimenta cerca de 25 millones de líneas de cambio cada semana. Aproximadamente la mitad de ellos son realizados por las decenas de miles de ingenieros que trabajan en nuestro monorepo, y la otra mitad por nuestros sistemas automatizados, en forma de actualizaciones de configuración o cambios a gran escala ([capítulo 22](#)). Muchos de esos cambios se inician desde fuera del proyecto inmediato. No imponemos muchas limitaciones a la capacidad de los ingenieros para reutilizar el código.

La apertura de nuestro código base fomenta un nivel de copropiedad que permite que todos asuman la responsabilidad del código base. Un beneficio de tal apertura es la capacidad de corregir errores directamente en un producto o servicio que utiliza (sujeto a aprobación, por supuesto) en lugar de quejarse. Esto también implica que muchas personas harán cambios en una parte del código base que pertenece a otra persona.

Otra cosa que hace que Google sea un poco diferente es que casi ningún equipo usa la bifurcación del repositorio. Todos los cambios se confirman en el encabezado del repositorio y son inmediatamente visibles para que todos los vean. Además, todas las compilaciones de software se realizan utilizando el último cambio confirmado que ha validado nuestra infraestructura de prueba. Cuando se construye un producto o servicio, casi todas las dependencias requeridas para ejecutarlo también se construyen desde la fuente, también desde el encabezado del repositorio. Google gestiona las pruebas a esta escala mediante el uso de un sistema CI. Uno de los componentes clave de nuestro sistema CI es nuestra Plataforma Automatizada de Pruebas (TAP).



Para obtener más información sobre TAP y nuestra filosofía de IC, consulte [capítulo 23](#).

Ya sea que esté considerando nuestro tamaño, nuestro monorepo o la cantidad de productos que ofrecemos, el entorno de ingeniería de Google es complejo. Cada semana experimenta millones de líneas cambiantes, se ejecutan miles de millones de casos de prueba, se construyen decenas de miles de binarios y se actualizan cientos de productos. ¡Hablando de complicado!

Las trampas de una gran suite de pruebas

A medida que crece la base de código, inevitablemente necesitará realizar cambios en el código existente. Cuando están mal escritas, las pruebas automatizadas pueden dificultar la realización de esos cambios. Las pruebas frágiles, aquellas que especifican en exceso los resultados esperados o se basan en un modelo extenso y complicado, en realidad pueden resistir el cambio. Estas pruebas mal escritas pueden fallar incluso cuando se realizan cambios no relacionados.

Si alguna vez ha realizado un cambio de cinco líneas en una característica solo para encontrar docenas de pruebas rotas no relacionadas, ha sentido la fricción de las pruebas frágiles. Con el tiempo, esta fricción puede hacer que un equipo se muestre reticente a realizar la refactorización necesaria para mantener una base de código saludable. Los capítulos siguientes cubrirán las estrategias que puede utilizar para mejorar la solidez y la calidad de sus pruebas.

Algunos de los peores infractores de las pruebas frágiles provienen del mal uso de objetos simulados. El código base de Google ha sufrido tanto por el abuso de los marcos de burlas que ha llevado a algunos ingenieros a declarar "¡no más simulacros!" Aunque esa es una declaración fuerte, comprender las limitaciones de los objetos simulados puede ayudarlo a evitar el mal uso de ellos.



Para obtener más información sobre cómo trabajar eficazmente con objetos simulados, consulte [Capítulo 13](#).

Además de la fricción causada por las pruebas frágiles, un conjunto más grande de pruebas será más lento de ejecutar. Cuanto más lento sea un conjunto de pruebas, con menos frecuencia se ejecutarán y menos beneficios proporcionará. Utilizamos una serie de técnicas para acelerar nuestro conjunto de pruebas, incluida la ejecución en paralelo y el uso de hardware más rápido. Sin embargo, este tipo de trucos eventualmente se ven inundados por una gran cantidad de casos de prueba individualmente lentos.

Las pruebas pueden volverse lentas por muchas razones, como arrancar partes significativas de un sistema, iniciar un emulador antes de la ejecución, procesar grandes conjuntos de datos o esperar a que se sincronicen sistemas dispares. Las pruebas a menudo comienzan lo suficientemente rápido, pero se ralentizan a medida que el

el sistema crece. Por ejemplo, tal vez tenga una prueba de integración que ejerza una sola dependencia que tarde cinco segundos en responder, pero con los años crece hasta depender de una docena de servicios, y ahora las mismas pruebas tardan cinco minutos.

Las pruebas también pueden volverse lentas debido a límites de velocidad innecesarios introducidos por funciones como `dormir()` y `establecerTiempoDeEspera()`. Las llamadas a estas funciones a menudo se usan como heurísticas ingenuas antes de verificar el resultado del comportamiento no determinista. Dormir medio segundo aquí o allá no parece demasiado peligroso al principio; sin embargo, si una función de "esperar y verificar" está integrada en una utilidad ampliamente utilizada, muy pronto habrá agregado minutos de tiempo de inactividad a cada ejecución de su conjunto de pruebas. Una mejor solución es buscar activamente una transición de estado con una frecuencia más cercana a los microsegundos. Puede combinar esto con un valor de tiempo de espera en caso de que una prueba no alcance un estado estable.

No mantener un conjunto de pruebas determinista y rápido asegura que se convertirá en un obstáculo para la productividad. En Google, los ingenieros que se encuentran con estas pruebas han encontrado formas de evitar las ralentizaciones, y algunos llegan a omitir las pruebas por completo al enviar cambios. Obviamente, esta es una práctica arriesgada y debe desaconsejarse, pero si un conjunto de pruebas está causando más daño que bien, eventualmente los ingenieros encontrarán una manera de hacer su trabajo, con o sin pruebas.

El secreto para vivir con un gran conjunto de pruebas es tratarlo con respeto. Incentivar a los ingenieros para que se preocupen por sus pruebas; recompénsalos tanto por tener pruebas sólidas como lo harías por tener un gran lanzamiento de funciones. Establezca objetivos de rendimiento apropiados y refactorice las pruebas lentas o marginales. Básicamente, trate sus pruebas como código de producción. Cuando los cambios simples comienzan a tomar tiempo no trivial, haga un esfuerzo para que sus pruebas sean menos frágiles.

Además de desarrollar la cultura adecuada, invierta en su infraestructura de prueba mediante el desarrollo de linters, documentación u otra asistencia que dificulte la redacción de malas pruebas. Reduzca la cantidad de marcos y herramientas que necesita admitir para aumentar la eficiencia del tiempo que invierte para mejorar las cosas.⁸ Si no invierte en facilitar la administración de sus pruebas, eventualmente los ingenieros decidirán que no vale la pena tenerlas.

Historia de las pruebas en Google

Ahora que hemos discutido cómo Google aborda las pruebas, podría ser esclarecedor saber cómo llegamos aquí. Como se mencionó anteriormente, los ingenieros de Google no siempre aceptaron el valor de las pruebas automatizadas. De hecho, hasta 2005, las pruebas estaban más cerca de una curiosidad que de una práctica disciplinada. La mayoría de las pruebas se realizaron manualmente, si es que se hicieron. Sin embargo, de 2005 a 2006, se produjo una revolución en las pruebas y cambió

⁸ Cada idioma admitido en Google tiene un marco de prueba estándar y una simulación/aplicación estándar.

biblioteca. Un conjunto de infraestructura ejecuta la mayoría de las pruebas en todos los idiomas en todo el código base.

la forma en que abordamos la ingeniería de software. Sus efectos continúan repercutiendo dentro de la empresa hasta el día de hoy.

La experiencia del proyecto GWS, que discutimos al comienzo de este capítulo, actuó como catalizador. Dejó en claro cuán poderosas podrían ser las pruebas automatizadas. Tras las mejoras de GWS en 2005, las prácticas comenzaron a extenderse por toda la empresa. El utilaje era primitivo. Sin embargo, los voluntarios, que llegaron a ser conocidos como Testing Grouplet, no permitieron que eso los detuviera.

Tres iniciativas clave ayudaron a introducir las pruebas automatizadas en la conciencia de la empresa: Clases de orientación, el programa Test Certified y Testing on the Toilet. Cada uno tuvo influencia de una manera completamente diferente y juntos remodelaron la cultura de ingeniería de Google.

Clases de orientación

Aunque gran parte del personal de ingeniería inicial de Google evitaba las pruebas, los pioneros de las pruebas automatizadas en Google sabían que al ritmo de crecimiento de la empresa, los nuevos ingenieros rápidamente superarían en número a los miembros existentes del equipo. Si pudieran llegar a todos los nuevos empleados de la empresa, podría ser una vía extremadamente eficaz para introducir un cambio cultural. Afortunadamente, había, y todavía hay, un único cuello de botella por el que pasan todas las nuevas contrataciones de ingeniería: la orientación.

La mayor parte del programa de orientación inicial de Google se refería a aspectos como los beneficios médicos y cómo funcionaba la Búsqueda de Google, pero a partir de 2005 también comenzó a incluir una discusión de una hora sobre el valor de las pruebas automatizadas.⁹ La clase cubrió los diversos beneficios de las pruebas, como una mayor productividad, una mejor documentación y soporte para la refactorización. También cubrió cómo escribir una buena prueba. Para muchos Nooglers (nuevos Googleers) en ese momento, esa clase fue su primera exposición a este material. Lo que es más importante, todas estas ideas se presentaron como si fueran una práctica habitual en la empresa. Los nuevos empleados no tenían idea de que estaban siendo utilizados como caballos de Troya para infiltrar esta idea en sus equipos desprevenidos.

Cuando los Nooglers se unieron a sus equipos siguiendo la orientación, comenzaron a escribir pruebas y a cuestionar a los del equipo que no lo hicieron. En solo uno o dos años, la población de ingenieros a los que se les había enseñado a probar superó en número a los ingenieros de la cultura de prueba previa. Como resultado, muchos proyectos nuevos comenzaron con el pie derecho.

Las pruebas ahora se han practicado más ampliamente en la industria, por lo que la mayoría de los nuevos empleados llegan con las expectativas de pruebas automatizadas firmemente establecidas. No obstante, las clases de orientación continúan estableciendo expectativas sobre las pruebas y conectan lo que los Nooglers

⁹ Esta clase tuvo tanto éxito que todavía hoy se enseña una versión actualizada. De hecho, es uno de los más largos. impartiendo clases de orientación en la historia de la empresa.

saber acerca de las pruebas fuera de Google a los desafíos de hacerlo en nuestra base de código muy grande y muy compleja.

Prueba certificada

Inicialmente, las partes más grandes y complejas de nuestro código base parecían resistentes a las buenas prácticas de prueba. Algunos proyectos tenían una calidad de código tan baja que era casi imposible probarlos. Para dar a los proyectos un camino claro hacia adelante, Testing Grouplet ideó un programa de certificación al que llamaron Test Certified. Test Certified tenía como objetivo brindar a los equipos una forma de comprender la madurez de sus procesos de prueba y, lo que es más importante, instrucciones de libros de cocina sobre cómo mejorarlo.

El programa se organizó en cinco niveles, y cada nivel requería algunas acciones concretas para mejorar la higiene de las pruebas en el equipo. Los niveles se diseñaron de tal manera que cada paso hacia arriba se pudiera lograr en un trimestre, lo que lo convirtió en un ajuste conveniente para la cadencia de planificación interna de Google.

Test Certified Level 1 cubrió los conceptos básicos: configurar una compilación continua; comenzar a rastrear la cobertura del código; clasifique todas sus pruebas como pequeñas, medianas o grandes; identificar (pero no necesariamente corregir) pruebas escamosas; y cree un conjunto de pruebas rápidas (no necesariamente completas) que se puedan ejecutar rápidamente. Cada nivel subsiguiente agregó más desafíos como "sin lanzamientos con pruebas rotas" o "eliminar todas las pruebas no deterministas". Para el nivel 5, todas las pruebas estaban automatizadas, se ejecutaban pruebas rápidas antes de cada compromiso, se había eliminado todo el no determinismo y se cubría cada comportamiento. Un tablero interno aplicaba presión social al mostrar el nivel de cada equipo. No pasó mucho tiempo antes de que los equipos compitieran entre sí para subir la escalera.

Cuando el programa Test Certified fue reemplazado por un enfoque automatizado en 2015 (más sobre el pH más adelante), había ayudado a más de 1500 proyectos a mejorar su cultura de pruebas.

Prueba en el inodoro

De todos los métodos que Testing Grouplet usó para tratar de mejorar las pruebas en Google, quizás ninguno fue más original que Testing on the Toilet (TotT). El objetivo de TotT era bastante simple: crear conciencia de forma activa sobre las pruebas en toda la empresa. La pregunta es, ¿cuál es la mejor forma de hacerlo en una empresa con empleados repartidos por todo el mundo?

The Testing Grouplet consideró la idea de un boletín informativo por correo electrónico regular, pero dado el gran volumen de correo electrónico que todos manejan en Google, era probable que se perdiera en el ruido. Después de un poco de lluvia de ideas, alguien propuso la idea de colocar volantes en los baños como una broma. Rápidamente reconocimos su genialidad: el baño es un lugar que todos deben visitar al menos una vez al día, pase lo que pase. Broma o no, la idea era tan barata de implementar que había que probarla.

En abril de 2006, apareció en los baños de Google un breve artículo sobre cómo mejorar las pruebas en Python. Este primer episodio fue publicado por un pequeño grupo de voluntarios. Decir que la reacción fue polarizada es quedarse corto; algunos lo vieron como una invasión del espacio personal y objetaron fuertemente. Las listas de correo se llenaron de quejas, pero los creadores de TotT estaban contentos: las personas que se quejaban todavía hablaban de pruebas.

En última instancia, el alboroto disminuyó y TotT se convirtió rápidamente en un elemento básico de la cultura de Google. Hasta la fecha, los ingenieros de toda la empresa han producido varios cientos de episodios, cubriendo casi todos los aspectos imaginables de las pruebas (además de una variedad de otros temas técnicos). Los nuevos episodios se esperan con impaciencia y algunos ingenieros incluso se ofrecen como voluntarios para publicar los episodios en sus propios edificios. Limitamos intencionalmente cada episodio a exactamente una página, desafiando a los autores a centrarse en los consejos más importantes y prácticos. Un buen episodio contiene algo que un ingeniero puede llevar al escritorio de inmediato y probar.

Irónicamente para una publicación que aparece en uno de los lugares más privados, TotT ha tenido un impacto público descomunal. La mayoría de los visitantes externos ven un episodio en algún momento de su visita, y dichos encuentros a menudo conducen a conversaciones divertidas sobre cómo los usuarios de Google siempre parecen estar pensando en el código. Además, los episodios de TotT son excelentes publicaciones de blog, algo que los autores originales de TotT reconocieron desde el principio. comenzaron a publicar [versiones ligeramente editadas públicamente](#), ayudando a compartir nuestra experiencia con la industria en general.

A pesar de comenzar como una broma, TotT ha tenido la carrera más larga y el impacto más profundo de cualquiera de las iniciativas de prueba iniciadas por Testing Grouplet.

Probando la cultura hoy

Hoy en día, la cultura de pruebas en Google ha recorrido un largo camino desde 2005. Los nooglers aún asisten a clases de orientación sobre pruebas y TotT continúa distribuyéndose casi todas las semanas. Sin embargo, las expectativas de las pruebas se han integrado más profundamente en el flujo de trabajo diario de los desarrolladores.

Cada cambio de código en Google debe pasar por una revisión de código. Y se espera que cada cambio incluya tanto el código de función como las pruebas. Se espera que los revisores revisen la calidad y corrección de ambos. De hecho, es perfectamente razonable bloquear un cambio si faltan pruebas.

Como reemplazo de Test Certified, uno de nuestros equipos de productividad de ingeniería lanzó recientemente una herramienta llamada Project Health (pH). La herramienta de pH recopila continuamente docenas de métricas sobre la salud de un proyecto, incluida la cobertura de prueba y la latencia de prueba, y las pone a disposición internamente. El pH se mide en una escala de uno (peor) a cinco (mejor). Un proyecto de pH-1 se considera un problema que el equipo debe abordar. Casi todos los equipos que ejecutan una compilación continua obtienen automáticamente una puntuación de pH.

Con el tiempo, las pruebas se han convertido en una parte integral de la cultura de ingeniería de Google. Tenemos innumerables formas de reforzar su valor para los ingenieros de toda la empresa. A través de una combinación de capacitación, pequeños empujones, tutoría y, sí, incluso un poco de competencia amistosa, hemos creado la clara expectativa de que las pruebas son trabajo de todos.

¿Por qué no empezamos por exigir la redacción de pruebas?

El Grupo de Pruebas había considerado pedir un mandato de prueba a los líderes senior, pero rápidamente decidió no hacerlo. Cualquier orden sobre cómo desarrollar código sería seriamente contraria a la cultura de Google y probablemente ralentizaría el progreso, independientemente de la idea que se ordene. La creencia era que las ideas exitosas se difundirían, por lo que el enfoque se convirtió en demostrar el éxito.

Si los ingenieros estaban decidiendo escribir pruebas por su cuenta, significaba que habían aceptado completamente la idea y probablemente seguirían haciendo lo correcto, incluso si nadie los obligaba a hacerlo.

Los límites de las pruebas automatizadas

Las pruebas automatizadas no son adecuadas para todas las tareas de prueba. Por ejemplo, probar la calidad de los resultados de búsqueda a menudo implica el juicio humano. Realizamos estudios internos específicos utilizando evaluadores de calidad de búsqueda que ejecutan consultas reales y registran sus impresiones. Del mismo modo, es difícil capturar los matices de la calidad de audio y video en una prueba automatizada, por lo que a menudo usamos el juicio humano para evaluar el rendimiento de los sistemas de telefonía o videollamadas.

Además de los juicios cualitativos, hay ciertas evaluaciones creativas en las que sobresalen los humanos. Por ejemplo, buscar vulnerabilidades de seguridad complejas es algo que los humanos hacen mejor que los sistemas automatizados. Una vez que un ser humano ha descubierto y entendido una falla, se puede agregar a un sistema de prueba de seguridad automatizado como el de Google.

Escáner de seguridad en la nube donde se puede ejecutar de forma continua y a escala.

Un término más generalizado para esta técnica es prueba exploratoria. Las pruebas exploratorias son un esfuerzo fundamentalmente creativo en el que alguien trata la aplicación que se está probando como un rompecabezas que se debe resolver, tal vez mediante la ejecución de una serie de pasos inesperados o la inserción de datos inesperados. Al realizar una prueba exploratoria, los problemas específicos que se van a encontrar se desconocen al principio. Se descubren gradualmente al sondar las rutas de código que comúnmente se pasan por alto o las respuestas inusuales de la aplicación. Al igual que con la detección de vulnerabilidades de seguridad, tan pronto como una prueba exploratoria descubra un problema, se debe agregar una prueba automatizada para evitar futuras regresiones.

El uso de pruebas automatizadas para cubrir comportamientos bien entendidos permite que los esfuerzos costosos y cualitativos de los probadores humanos se centren en las partes de sus productos para las que pueden proporcionar el mayor valor y evitar aburrirlos hasta las lágrimas en el proceso.

Conclusión

La adopción de pruebas automatizadas dirigidas por desarrolladores ha sido una de las prácticas de ingeniería de software más transformadoras en Google. Nos ha permitido construir sistemas más grandes con equipos más grandes, más rápido de lo que nunca creímos posible. Nos ha ayudado a mantenernos al día con el ritmo cada vez mayor del cambio tecnológico. Durante los últimos 15 años, hemos transformado con éxito nuestra cultura de ingeniería para convertir las pruebas en una norma cultural. A pesar de que la compañía creció casi 100 veces desde que comenzó el viaje, nuestro compromiso con la calidad y las pruebas es más fuerte hoy que nunca.

Este capítulo ha sido escrito para ayudarlo a orientarse sobre cómo piensa Google acerca de las pruebas. En los próximos capítulos, profundizaremos aún más en algunos temas clave que han ayudado a dar forma a nuestra comprensión de lo que significa escribir pruebas buenas, estables y confiables. Discutiremos el qué, por qué y cómo de las pruebas unitarias, el tipo de prueba más común en Google. Nos adentraremos en el debate sobre cómo usar efectivamente los dobles de prueba en las pruebas a través de técnicas como falsificación, stubing y pruebas de interacción. Finalmente, discutiremos los desafíos de probar sistemas más grandes y complejos, como muchos de los que tenemos en Google.

Al final de estos tres capítulos, debería tener una imagen mucho más profunda y clara de las estrategias de prueba que usamos y, lo que es más importante, por qué las usamos.

TL; DR

- Las pruebas automatizadas son fundamentales para permitir que el software cambie.
- Para que las pruebas se escalen, deben estar automatizadas.
- Es necesario un conjunto de pruebas equilibrado para mantener una cobertura de pruebas saludable.
- “Si te gustó, deberías haberlo probado”.
- Cambiar la cultura de las pruebas en las organizaciones lleva tiempo.

CAPÍTULO 12

Examen de la unidad

*Escrito por Erik Kuefler
Editado por Tom Mansreck*

El capítulo anterior introdujo dos de los ejes principales a lo largo de los cuales Google clasifica las pruebas: *Tamaño* y *alcance*. En resumen, el tamaño se refiere a los recursos consumidos por una prueba y lo que se le permite hacer, y el alcance se refiere a la cantidad de código que una prueba pretende validar. Aunque Google tiene definiciones claras para el tamaño de la prueba, el alcance tiende a ser un poco más confuso. Usamos el término *prueba de unidad* para referirnos a pruebas de alcance relativamente limitado, como de una sola clase o método. Las pruebas unitarias suelen ser de tamaño pequeño, pero no siempre es así.

Después de prevenir errores, el propósito más importante de una prueba es mejorar la productividad de los ingenieros. En comparación con las pruebas de alcance más amplio, las pruebas unitarias tienen muchas propiedades que las convierten en una forma excelente de optimizar la productividad:

- Tienden a ser pequeños de acuerdo con las definiciones de tamaño de prueba de Google. Las pruebas pequeñas son rápidas y deterministas, lo que permite a los desarrolladores ejecutarlas con frecuencia como parte de su flujo de trabajo y obtener comentarios inmediatos.
- Tienden a ser fáciles de escribir al mismo tiempo que el código que están probando, lo que permite a los ingenieros centrar sus pruebas en el código en el que están trabajando sin tener que configurar y comprender un sistema más grande.
- Promueven altos niveles de cobertura de prueba porque son rápidos y fáciles de escribir. La alta cobertura de prueba permite a los ingenieros realizar cambios con la confianza de que no están rompiendo nada.
- Tienden a facilitar la comprensión de lo que está mal cuando fallan porque cada prueba es conceptualmente simple y está enfocada en una parte particular del sistema.
- Pueden servir como documentación y ejemplos, mostrando a los ingenieros cómo usar la parte del sistema que se está probando y cómo se pretende que funcione ese sistema.

Debido a sus muchas ventajas, la mayoría de las pruebas escritas en Google son pruebas unitarias y, como regla general, animamos a los ingenieros a apuntar a una combinación de alrededor del 80 % de pruebas unitarias y un 20 % de pruebas de alcance más amplio. Este consejo, junto con la facilidad de escribir pruebas unitarias y la velocidad con la que se ejecutan, significa que los ingenieros ejecutan un *o* de pruebas unitarias: no es inusual que un ingeniero ejecute miles de pruebas unitarias (directa o indirectamente) durante una jornada laboral promedio.

Debido a que constituyen una parte tan importante de la vida de los ingenieros, Google pone mucho énfasis en *prueba de mantenibilidad*. Las pruebas mantenibles son aquellas que “simplemente funcionan”: después de escribirlas, los ingenieros no necesitan volver a pensar en ellas hasta que fallan, y esas fallas indican errores reales con causas claras. La mayor parte de este capítulo se centra en explorar la idea de la mantenibilidad y las técnicas para lograrla.

La importancia de la mantenibilidad

Imagine este escenario: Mary quiere agregar una nueva característica simple al producto y puede implementarla rápidamente, tal vez requiriendo solo un par de docenas de líneas de código. Pero cuando va a verificar su cambio, recibe una pantalla llena de errores del sistema de prueba automatizado. Pasa el resto del día revisando esos fracasos uno por uno. En cada caso, el cambio no introdujo ningún error real, pero rompió algunas de las suposiciones que hizo la prueba sobre la estructura interna del código, lo que requirió que esas pruebas se actualizaran. A menudo, tiene dificultades para averiguar qué intentaban hacer las pruebas en primer lugar, y los trucos que agrega para corregirlas hacen que esas pruebas sean aún más difíciles de entender en el futuro. En última instancia, lo que debería haber sido un trabajo rápido termina tomando horas o incluso días de mucho trabajo,

En este caso, las pruebas tuvieron el efecto contrario al deseado al agotar la productividad en lugar de mejorarla sin aumentar significativamente la calidad del código bajo prueba. Este escenario es demasiado común y los ingenieros de Google luchan con él todos los días. No existe una varita mágica, pero muchos ingenieros de Google han estado trabajando para desarrollar conjuntos de patrones y prácticas para aliviar estos problemas, que alentamos al resto de la empresa a seguir.

Los problemas con los que se encontró Mary no fueron su culpa, y no había nada que pudiera haber hecho para evitarlos: las pruebas incorrectas deben corregirse antes de que se registren, para que no supongan una carga para los futuros ingenieros. En términos generales, los problemas que encontró se dividen en dos categorías. Primero, las pruebas con las que estaba trabajando eran *frágiles*: se rompieron en respuesta a un cambio inofensivo y no relacionado que no introdujo errores reales. En segundo lugar, las pruebas fueron *poco claras*: después de que estaban fallando, era difícil determinar qué estaba mal, cómo solucionarlo y qué se suponía que estaban haciendo esas pruebas en primer lugar.

Prevención de pruebas frágiles

Como se acaba de definir, una prueba frágil es aquella que falla frente a un cambio no relacionado con el código de producción que no introduce ningún error real.¹ Dichas pruebas deben ser diagnosticadas y corregidas por ingenieros como parte de su trabajo. En bases de código pequeñas con solo unos pocos ingenieros, tener que modificar algunas pruebas para cada cambio puede no ser un gran problema. Pero si un equipo escribe regularmente pruebas frágiles, el mantenimiento de las pruebas inevitablemente consumirá una proporción cada vez mayor del tiempo del equipo, ya que se ven obligados a analizar una cantidad cada vez mayor de fallas en un conjunto de pruebas en constante crecimiento. Si los ingenieros deben ajustar manualmente un conjunto de pruebas para cada cambio, llamarlo "conjunto de pruebas automatizado" es un poco exagerado.

Las pruebas frágiles causan dolor en las bases de código de cualquier tamaño, pero se vuelven particularmente agudas en la escala de Google. Un ingeniero individual podría ejecutar fácilmente miles de pruebas en un solo día durante el curso de su trabajo y un solo cambio a gran escala (ver [capítulo 22](#)) puede desencadenar cientos de miles de pruebas. A esta escala, las roturas espurias que afectan incluso a un pequeño porcentaje de las pruebas pueden desperdiciar una gran cantidad de tiempo de ingeniería. Los equipos de Google varían bastante en cuanto a la fragilidad de sus conjuntos de pruebas, pero hemos identificado algunas prácticas y patrones que tienden a hacer que las pruebas sean más resistentes al cambio.

Esforzarse por las pruebas inmutables

Antes de hablar de patrones para evitar pruebas frágiles, debemos responder una pregunta: ¿con qué frecuencia deberíamos esperar cambiar una prueba después de escribirla? Cualquier tiempo dedicado a actualizar pruebas antiguas es tiempo que no se puede dedicar a un trabajo más valioso. Por lo tanto, *la prueba ideal es invariable*: una vez escrito, nunca necesita cambiar a menos que cambien los requisitos del sistema bajo prueba.

¿Cómo se ve esto en la práctica? Necesitamos pensar en los tipos de cambios que los ingenieros hacen en el código de producción y cómo debemos esperar que las pruebas respondan a esos cambios. Fundamentalmente, hay cuatro tipos de cambios:

Refactorizaciones puras

Cuando un ingeniero refatoriza las partes internas de un sistema sin modificar su interfaz, ya sea por rendimiento, claridad o cualquier otra razón, las pruebas del sistema no deberían cambiar. El papel de las pruebas en este caso es asegurar que la refactorización no cambie el comportamiento del sistema. Las pruebas que deben cambiarse durante una refactorización indican que el cambio está afectando el comportamiento del sistema y no es una refactorización pura, o que las pruebas no se escribieron en un nivel adecuado.

1 Tenga en cuenta que esto es ligeramente diferente de una *prueba escamosa*, que falla de manera no determinista sin ningún cambio en Código de producción.

de abstracción La dependencia de Google de los cambios a gran escala (descritos en [capítulo 22](#)) para hacer tales refactorizaciones hace que este caso sea particularmente importante para nosotros.

Nuevas características

Cuando un ingeniero agrega nuevas funciones o comportamientos a un sistema existente, los comportamientos existentes del sistema no deberían verse afectados. El ingeniero debe escribir nuevas pruebas para cubrir los nuevos comportamientos, pero no debería necesitar cambiar ninguna prueba existente. Al igual que con las refactorizaciones, un cambio en las pruebas existentes al agregar nuevas funciones sugiere consecuencias no deseadas de esa función o pruebas inapropiadas.

Corrección de errores

Reparar un error es muy parecido a agregar una nueva función: la presencia del error sugiere que faltaba un caso en el conjunto de pruebas inicial, y la corrección del error debe incluir ese caso de prueba faltante. Una vez más, las correcciones de errores normalmente no deberían requerir actualizaciones de las pruebas existentes.

Cambios de comportamiento

Cambiar el comportamiento existente de un sistema es el único caso en el que esperamos tener que realizar actualizaciones en las pruebas existentes del sistema. Tenga en cuenta que tales cambios tienden a ser significativamente más costosos que los otros tres tipos. Es probable que los usuarios de un sistema confíen en su comportamiento actual, y los cambios en ese comportamiento requieren coordinación con esos usuarios para evitar confusiones o interrupciones. Cambiar una prueba en este caso indica que estamos incumpliendo un contrato explícito del sistema, mientras que los cambios en los casos anteriores indican que estamos incumpliendo un contrato no deseado. Las bibliotecas de bajo nivel a menudo invertirán un esfuerzo significativo para evitar la necesidad de realizar un cambio de comportamiento para no perjudicar a sus usuarios.

La conclusión es que después de escribir una prueba, no debería necesitar tocar esa prueba nuevamente a medida que refactoriza el sistema, corrige errores o agrega nuevas funciones. Esta comprensión es lo que hace posible trabajar con un sistema a escala: expandirlo requiere escribir solo una pequeña cantidad de pruebas nuevas relacionadas con el cambio que está realizando en lugar de tener que tocar todas las pruebas que se han escrito alguna vez en el sistema. Solo romper los cambios en el comportamiento de un sistema debería requerir volver atrás para cambiar sus pruebas y, en tales situaciones, el costo de actualizar esas pruebas tiende a ser pequeño en relación con el costo de actualizar a todos los usuarios del sistema.

Prueba a través de API públicas

Ahora que entendemos nuestro objetivo, veamos algunas prácticas para asegurarnos de que las pruebas no necesitan cambiar a menos que cambien los requisitos del sistema que se está probando. Con mucho, la forma más importante de garantizar esto es escribir pruebas que invoquen el sistema que se está probando de la misma manera que lo harían sus usuarios; es decir, realizar llamadas contra su API pública [en lugar de sus detalles de implementación](#). Si las pruebas funcionan de la misma manera que los usuarios del sistema, por definición, el cambio que interrumpe una prueba también podría interrumpir a un usuario. Como bono adicional, tales pruebas pueden servir como ejemplos y documentación útiles para los usuarios.

Considerar **Ejemplo 12-1**, que valida una transacción y la guarda en una base de datos.

Ejemplo 12-1. Una API de transacción

```
públicovacióprocesoTransacción(transacción transacción) {  
    si(es válida(transacción)) {  
        guardar en la base de datos(transacción);  
    }  
}  
  
privadobooloñoes válida(Transacción t) {  
    return not.obtenerCantidad() <t.obtenerRemitente().obtenersaldo();  
}  
  
privadovacióguardar en la base de datos(Transacción t) {  
    cadena=t.obtenerRemitente() +"," +t.getRecipient() +"," +t.obtenerCantidad(); base de  
    datos.poner(t.obtenerId(),s);  
}  
  
públicovacióestablecer el saldo de la cuenta(Cadena nombre de cuenta,En tbalance) {  
    // Escribir el saldo directamente en la base de datos  
}  
  
públicovacióobtenersaldodecuenta(Cadena nombre de cuenta) {  
    // Leer transacciones de la base de datos para determinar el saldo de la cuenta  
}
```

Una forma tentadora de probar este código sería eliminar los modificadores de visibilidad "privados" y probar la lógica de implementación directamente, como se demuestra en **Ejemplo 12-2**.

Ejemplo 12-2. Una prueba ingenua de la implementación de una API de transacción

```
@Prueba  
públicovacióLa cuenta vacía no debe ser válida() {  
    afirmar que(procesador.es válida(nuevaTransacción().establecerRemitente(EMPTY_ACCOUNT)))  
        . Es falso();  
}  
  
@Prueba  
públicovaciódebería guardar datos serializados() {  
    procesador.guardar en la base de datos(nuevaTransacción()  
        • Pon la identificación(123)  
        • establecerRemitente("a mí")  
        • establecerDestinatario("usted")  
        . fijar cantidad(100));  
    afirmar que(base de datos.conseguir(123)).es igual a("yo, tu, 100");  
}
```

Esta prueba interactúa con el procesador de transacciones de una manera muy diferente a como lo harían sus usuarios reales: examina el estado interno del sistema y llama a métodos que no están publicados.

expuestas lícitamente como parte de la API del sistema. Como resultado, la prueba es frágil y casi cualquier refactorización del sistema bajo prueba (como cambiar el nombre de sus métodos, factorizarlos en una clase auxiliar o cambiar el formato de serialización) haría que la prueba fallara, incluso si tal el cambio sería invisible para los usuarios reales de la clase.

En cambio, se puede lograr la misma cobertura de prueba probando solo con la API pública de la clase, como se muestra en [Ejemplo 12-3](#).²

Ejemplo 12-3. Probando la API pública

```
@Prueba


públicovaciódebe transferir fondos() {


    procesador.establecer el saldo de la cuenta("a mí",150);
    procesador.establecer el saldo de la cuenta("usted",20);

    procesador.procesoTransacción(nuevaTransacción()
        . establecerRemitente("a mí")
        . establecerDestinatario("usted")
        . fijar cantidad(100));

    afirmar que(procesador.obtenersaldodecuenta("a mí")).es igual a(50);
    afirmar que(procesador.obtenersaldodecuenta("usted")).es igual a(120);
}
```



```
@Prueba


públicovacióno debe realizar transacciones no válidas() {


    procesador.establecer el saldo de la cuenta("a mí",50);
    procesador.establecer el saldo de la cuenta("usted",20);

    procesador.procesoTransacción(nuevaTransacción()
        . establecerRemitente("a mí")
        . establecerDestinatario("usted")
        . fijar cantidad(100));

    afirmar que(procesador.obtenersaldodecuenta("a mí")).es igual a(50);
    afirmar que(procesador.obtenersaldodecuenta("usted")).es igual a(20);
}
```

Las pruebas que utilizan solo API públicas, por definición, acceden al sistema bajo prueba de la misma manera que lo harían sus usuarios. Tales pruebas son más realistas y menos frágiles porque forman contratos explícitos: si una prueba de este tipo se rompe, implica que un usuario existente del sistema también se romperá. Probar solo estos contratos significa que puede hacer cualquier refactorización interna del sistema que desee sin tener que preocuparse por realizar cambios tediosos en las pruebas.

² Esto a veces se llama el "[Utilice el primer principio de la puerta de entrada.](#)"

No siempre está claro qué constituye una "API pública", y la pregunta realmente llega al corazón de qué es una "unidad" en las pruebas unitarias. Las unidades pueden ser tan pequeñas como una función individual o tan amplias como un conjunto de varios paquetes/módulos relacionados. Cuando decimos "API pública" en este contexto, en realidad estamos hablando de la API expuesta por esa unidad a terceros fuera del equipo propietario del código. Esto no siempre se alinea con la noción de visibilidad proporcionada por algunos lenguajes de programación; por ejemplo, las clases en Java pueden definirse a sí mismas como "públicas" para que otros paquetes en la misma unidad puedan acceder a ellas, pero no están destinadas a que las usen otras partes fuera de la unidad. Algunos lenguajes como Python no tienen una noción integrada de visibilidad (a menudo se basan en convenciones como el prefijo de nombres de métodos privados con guiones bajos), y construyen sistemas como [bazel](#) puede restringir aún más quién puede depender de las API declaradas públicas por el lenguaje de programación.

Definir un alcance apropiado para una unidad y, por lo tanto, lo que debe considerarse la API pública es más un arte que una ciencia, pero aquí hay algunas reglas generales:

- Si un método o clase existe solo para admitir una o dos clases más (es decir, es una "clase auxiliar"), probablemente no debería considerarse su propia unidad, y su funcionalidad debería probarse a través de esas clases en su lugar, de directamente
- Si un paquete o clase está diseñado para que cualquiera pueda acceder a él sin tener que consultar con sus propietarios, es casi seguro que constituye una unidad que debe probarse directamente, donde sus pruebas acceden a la unidad de la misma manera que lo harían los usuarios.
- Si solo las personas que lo poseen pueden acceder a un paquete o clase, pero está diseñado para proporcionar una funcionalidad general útil en una variedad de contextos (es decir, es una "biblioteca de soporte"), también debe ser considerado una unidad y probado directamente. Por lo general, esto creará cierta redundancia en las pruebas dado que el código de la biblioteca de soporte estará cubierto tanto por sus propias pruebas como por las pruebas de sus usuarios. Sin embargo, dicha redundancia puede ser valiosa: sin ella, se podría introducir una brecha en la cobertura de la prueba si alguna vez se eliminara uno de los usuarios de la biblioteca (y sus pruebas).

En Google, descubrimos que a veces es necesario convencer a los ingenieros de que las pruebas a través de las API públicas son mejores que las pruebas con los detalles de implementación. La renuencia es comprensible porque a menudo es mucho más fácil escribir pruebas centradas en el código que acaba de escribir en lugar de averiguar cómo afecta ese código al sistema en su conjunto. Sin embargo, hemos encontrado valioso fomentar tales prácticas, ya que el esfuerzo inicial adicional se paga por sí mismo muchas veces en la reducción de la carga de mantenimiento. Probar contra las API públicas no evitará por completo la fragilidad, pero es lo más importante que puede hacer para asegurarse de que sus pruebas fallen solo en el caso de cambios significativos en su sistema.

Estado de prueba, no interacciones

Otra forma en que las pruebas comúnmente dependen de los detalles de implementación no implica qué métodos del sistema llama la prueba, sino cómo se verifican los resultados de esas llamadas. En general, hay dos formas de verificar que un sistema bajo prueba se comporta como se espera. Con *prueba estatal*, observa el sistema en sí para ver cómo se ve después de invocarlo. Con *pruebas de interacción*, en su lugar, verifica que el sistema tomó una secuencia esperada de acciones en sus colaboradores en respuesta a invocarlo. Muchas pruebas realizarán una combinación de validación de estado e interacción.

Las pruebas de interacción tienden a ser más frágiles que las pruebas estatales por la misma razón que es más frágil probar un método privado que probar un método público: las pruebas de interacción verifican cómo un sistema llegó a su resultado, mientras que por lo general sólo debería preocuparse qué el resultado es. [Ejemplo 12-4](#) ilustra una prueba que utiliza un doble de prueba (explicado más adelante en [Capítulo 13](#)) para verificar cómo un sistema interactúa con una base de datos.

Ejemplo 12-4. Una prueba de interacción frágil

```
@Prueba
público void crearUsuario() {
    cuentas.create_usuario("foobar");
    verificar
        (base_de_datos).poner("foobar");
}
```

La prueba verifica que se realizó una llamada específica contra una API de base de datos, pero hay un par de formas diferentes en que podría salir mal:

- Si un error en el sistema bajo prueba hace que el registro se elimine de la base de datos poco tiempo después de haber sido escrito, la prueba pasará aunque hubiésemos querido que fallara.
- Si el sistema bajo prueba se refactoriza para llamar a una API ligeramente diferente para escribir un registro equivalente, la prueba fallará aunque hubiéramos querido que pasara.

Es mucho menos frágil probar directamente contra el estado del sistema, como se demuestra en [Ejemplo 12-5](#).

Ejemplo 12-5. Prueba contra el estado

```
@Prueba
público void deberíaCrearUsuarios() {
    cuentas.create_usuario("foobar");
    afirmar que(cuentas.obtenerUsuario("foobar")).No es nulo();
}
```

Esta prueba expresa con mayor precisión lo que nos importa: el estado del sistema bajo prueba después de interactuar con él.

La razón más común de las pruebas de interacción problemáticas es una dependencia excesiva de los marcos de trabajo simulados. Estos marcos facilitan la creación de dobles de prueba que registran y verifican cada llamada realizada contra ellos, y el uso de esos dobles en lugar de objetos reales en las pruebas. Esta estrategia conduce directamente a pruebas de interacción frágiles, por lo que tendemos a preferir el uso de objetos reales en favor de objetos simulados, siempre que los objetos reales sean rápidos y deterministas.



Para obtener una discusión más extensa sobre los dobles de prueba y los marcos de simulación, cuándo deben usarse y las alternativas más seguras, consulte [Capítulo 13](#)

Escribir pruebas claras

Tarde o temprano, incluso si hemos evitado por completo la fragilidad, nuestras pruebas fallarán. La falla es algo bueno: las fallas en las pruebas brindan señales útiles a los ingenieros y son una de las principales formas en que una prueba unitaria proporciona valor.

Las fallas en las pruebas ocurren por una de dos razones:³

- El sistema bajo prueba tiene un problema o está incompleto. Este resultado es exactamente para lo que están diseñadas las pruebas: alertarlo sobre errores para que pueda corregirlos.
- La prueba en sí es defectuosa. En este caso, no hay ningún problema con el sistema bajo prueba, pero la prueba se especificó incorrectamente. Si se trata de una prueba existente en lugar de una que acaba de escribir, esto significa que la prueba es frágil. La sección anterior discutió cómo evitar las pruebas frágiles, pero rara vez es posible eliminarlas por completo.

Cuando falla una prueba, el primer trabajo de un ingeniero es identificar en cuál de estos casos cae la falla y luego diagnosticar el problema real. La velocidad a la que el ingeniero puede hacerlo depende de la prueba *claridad*. Una prueba clara es aquella cuyo propósito para existir y la razón para fallar es inmediatamente clara para el ingeniero que diagnostica una falla. Las pruebas no logran la claridad cuando las razones del fracaso no son obvias o cuando es difícil averiguar por qué se escribieron originalmente. Las pruebas claras también aportan otros beneficios, como documentar el sistema bajo prueba y servir más fácilmente como base para nuevas pruebas.

La claridad de la prueba se vuelve significativa con el tiempo. Las pruebas a menudo durarán más que los ingenieros que las escribieron, y los requisitos y la comprensión de un sistema cambiarán sutilmente a medida que envejezca. Es muy posible que una prueba fallida haya sido escrita hace años por un

³ Estas son también las mismas dos razones por las que una prueba puede ser "inestable". O bien el sistema bajo prueba tiene un fallo ístico, o la prueba es defectuosa de tal manera que a veces falla cuando debería pasar.

ingeniero ya no está en el equipo, sin dejar forma de averiguar su propósito o cómo solucionarlo. Esto contrasta con el código de producción poco claro, cuyo propósito generalmente se puede determinar con suficiente esfuerzo observando qué lo llama y qué se rompe cuando se elimina. Con una prueba poco clara, es posible que nunca comprenda su propósito, ya que eliminar la prueba no tendrá otro efecto que (potencialmente) introducir un agujero sutil en la cobertura de la prueba.

En el peor de los casos, estas pruebas oscuras acaban siendo eliminadas cuando los ingenieros no saben cómo solucionarlas. La eliminación de tales pruebas no solo introduce un agujero en la cobertura de la prueba, sino que también indica que la prueba ha estado proporcionando un valor cero quizás durante todo el período que ha existido (que podría haber sido años).

Para que un conjunto de pruebas se amplíe y sea útil con el tiempo, es importante que cada prueba individual en ese conjunto sea lo más clara posible. Esta sección explora técnicas y formas de pensar acerca de las pruebas para lograr claridad.

Haga sus pruebas completas y concisas

Dos propiedades de alto nivel que ayudan a que las pruebas logren claridad son **integridad y concisión**. Una prueba es **incompleta** cuando su cuerpo contiene toda la información que un lector necesita para comprender cómo llega a su resultado. Una prueba es **desordenada** cuando no contenga otra información irrelevante o que distraiga. **Ejemplo 12-6** muestra una prueba que no es ni completa ni concisa:

Ejemplo 12-6. Una prueba incompleta y desordenada

```
@Prueba
públicovacióndebe realizar la adición() {
    calculadora calculadora=nuevoCalculadora(nuevoRedondeoEstrategia(),
        "no usado",ENABLE_COSINE_FEATURE,0.01,motor de cálculo,falso); En t
    resultado=calculadora.calcular(newTestCalculation()); afirmar que(resultado).
        es igual a(5); //¿De dónde salió este número?
}
```

La prueba pasa mucha información irrelevante al constructor, y las partes realmente importantes de la prueba están ocultas dentro de un método auxiliar. La prueba se puede hacer más completa aclarando las entradas del método auxiliar y más concisa utilizando otro auxiliar para ocultar los detalles irrelevantes de la construcción de la calculadora, como se ilustra en **Ejemplo 12-7**.

Ejemplo 12-7. Una prueba completa y concisa

```
@Prueba
públicovacióndebe realizar la adición() {
    calculadora calculadora=nuevaCalculadora();
    En resultado=calculadora.calcular(nuevoCálculo(2,Operación.MÁS,3));
```

```
    afirmar que(resultado).es igual a(5);
}
```

Las ideas que discutimos más adelante, especialmente sobre el código compartido, se relacionarán con la integridad y la concisión. En particular, a menudo puede valer la pena violar el principio DRY (Don't Repeat Yourself) si conduce a pruebas más claras. Recuerda: un *El cuerpo de la prueba debe contener toda la información necesaria para comprenderla sin contener información irrelevante o que distraiga..*

Comportamientos de prueba, no métodos

El primer instinto de muchos ingenieros es tratar de hacer coincidir la estructura de sus pruebas con la estructura de su código, de modo que cada método de producción tenga un método de prueba correspondiente. Este patrón puede ser conveniente al principio, pero con el tiempo conduce a problemas: a medida que el método que se prueba se vuelve más complejo, su prueba también crece en complejidad y se vuelve más difícil razonar sobre ella. Por ejemplo, considere el fragmento de código en [Ejemplo 12-8](#), que muestra los resultados de una transacción.

Ejemplo 12-8. Un fragmento de transacción

```
públicovacióMostrar resultados de transacciones(usuario usuario,transacción transacción) {
    interfaz de usuario.Mostrar mensaje("Tú compraste un"+transacción.getItemName()
()); si(usuario.obtenersaldo() <BAJO_BALANCE_UMBRAL) {
    interfaz de usuario.Mostrar mensaje("Advertencia: ¡su saldo es bajo!");
}
}
```

No sería raro encontrar una prueba que cubra los dos mensajes que podría mostrar el método, como se presenta en [Ejemplo 12-9](#).

Ejemplo 12-9. Una prueba basada en métodos

```
@Prueba
públicovaciótestDisplayTransactionResults() {
    procesador de transacciones.Mostrar resultados de transacciones(
        nuevoUsuarioConSaldo(
            BAJO_BALANCE_UMBRAL.más(dolares(2))),
        nuevoTransacción("Algún artículo",dolares(3)));
    afirmar que(interfaz de usuario.obtenerTexto()).contiene("Has comprado un artículo");
    afirmar que(interfaz de usuario.obtenerTexto()).contiene("tu saldo es bajo");
}
```

Con tales pruebas, es probable que la prueba comenzara cubriendo solo el primer método. Posteriormente, un ingeniero amplió la prueba cuando se agregó el segundo mensaje (violando la idea de pruebas inmutables que discutimos anteriormente). Esta modificación sienta un mal precedente: a medida que el método bajo prueba se vuelve más complejo e implementa más

funcionalidad, su prueba de unidad se volverá cada vez más complicada y será cada vez más difícil trabajar con ella.

El problema es que enmarcar las pruebas en torno a los métodos puede, naturalmente, fomentar pruebas poco claras porque un solo método a menudo hace algunas cosas diferentes debajo del capó y puede tener varios casos complicados. Hay una mejor manera: en lugar de escribir una prueba para cada método, escriba una prueba para cada *conducta*.⁴ Un comportamiento es cualquier garantía que hace un sistema acerca de cómo responderá a una serie de entradas mientras se encuentra en un estado particular.⁵ Los comportamientos a menudo se pueden expresar usando las palabras “*dado*”, “*cuándo*” y “*entonces*”: “*Dado* que una cuenta bancaria está vacía, *cuándo* intentar retirar dinero de él, *después* la transacción es rechazada.” El mapeo entre métodos y comportamientos es de muchos a muchos: la mayoría de los métodos no triviales implementan múltiples comportamientos, y algunos comportamientos se basan en la interacción de múltiples métodos. El ejemplo anterior se puede reescribir utilizando pruebas basadas en el comportamiento, como se presenta en [Ejemplo 12-10](#).

Ejemplo 12-10. Una prueba basada en el comportamiento

```
@Prueba


úplicovaciódisplayTransactionResults_showsItemName() {
    procesador de transacciones.Mostrar resultados de transacciones(
        nuevoUsuario(),nuevoTransacción("Algún artículo")); afirmar que(interfaz de
        usuario.obtenerTexto()).contiene("Has comprado un artículo");
}


```



```
@Prueba


úplicovaciódisplayTransactionResults_showsLowBalanceWarning() {
    procesador de transacciones.Mostrar resultados de transacciones(
        nuevoUsuarioConSaldo(
            BAJO_BALANCE_UMBRAL.más(dolares(2))), nuevoTransacción(
            "Algún artículo",dolares(3))); afirmar que(interfaz de usuario.
        obtenerTexto()).contiene("tu saldo es bajo");
}


```

El texto estándar adicional requerido para dividir la prueba individual es *más que vale la pena*, y las pruebas resultantes son mucho más claras que la prueba original. Las pruebas basadas en el comportamiento tienden a ser más claras que las pruebas orientadas a métodos por varias razones. En primer lugar, se leen más como un lenguaje natural, lo que permite que se entiendan de forma natural en lugar de requerir un análisis mental laborioso. En segundo lugar, expresan más claramente *causa y efecto* porque cada prueba tiene un alcance más limitado. Finalmente, el hecho de que cada prueba sea corta y descriptiva hace que sea más fácil ver qué funcionalidad ya se ha probado y alienta a los ingenieros a agregar nuevos métodos de prueba simplificados en lugar de acumularlos en los métodos existentes.

4 Ver <https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html> <https://dannorth.net/introduciendo-bdd>.

5 Además, un *rasgo*(en el sentido del producto de la palabra) se puede expresar como una colección de comportamientos.

Pruebas de estructura para enfatizar comportamientos

Pensar en las pruebas como acopladas a comportamientos en lugar de métodos afecta significativamente la forma en que deben estructurarse. Recuerde que cada comportamiento tiene tres partes: un componente "dado" que define cómo se configura el sistema, un componente "cuándo" que define la acción que se tomará en el sistema y un componente "entonces" que valida el resultado.⁶ Las pruebas son más claras cuando esta estructura es explícita. Algunos marcos como **PepinoySpock** hornean directamente en dado/cuando/entonces. Otros idiomas pueden usar espacios en blanco y comentarios optionales para resaltar la estructura, como la que se muestra en [Ejemplo 12-11](#).

Ejemplo 12-11. Una prueba bien estructurada.

```
@Prueba
públicovacióTransferir fondos debe mover dinero entre cuentas() {
    // Dadas dos cuentas con saldos iniciales de $150 y $20 cuenta cuenta1=
    nuevaCuentaConSaldo(Dólar estadounidense(150)); cuenta cuenta2=
    nuevaCuentaConSaldo(Dólar estadounidense(20));

    // Al transferir $100 de la primera a la segunda cuenta banco.Transferir
    fondos(cuenta1,cuenta2,Dólar estadounidense(100));

    // Entonces los nuevos saldos de cuenta deben reflejar la transferencia afirmar que(
    cuenta1.obtenersaldo().es igual a(Dólar estadounidense(50)); afirmar que(cuenta2.
    obtenersaldo().es igual a(Dólar estadounidense(120));
}
```

Este nivel de descripción no siempre es necesario en las pruebas triviales y, por lo general, es suficiente con omitir los comentarios y confiar en los espacios en blanco para aclarar las secciones. Sin embargo, los comentarios explícitos pueden hacer que las pruebas más sofisticadas sean más fáciles de entender. Este patrón permite leer las pruebas en tres niveles de granularidad:

1. Un lector puede comenzar mirando el nombre del método de prueba (discutido a continuación) para obtener una descripción aproximada del comportamiento que se está probando.
2. Si eso no es suficiente, el lector puede buscar en los comentarios dado/cuándo/entonces una descripción formal del comportamiento.
3. Finalmente, un lector puede mirar el código real para ver precisamente cómo se expresa ese comportamiento.

Este patrón se viola más comúnmente al intercalar afirmaciones entre múltiples llamadas al sistema bajo prueba (es decir, combinando los bloques "cuando" y "entonces"). fusión

6 Estos componentes a veces se denominan "arreglar", "actuar" y "afirmar".

los bloques "entonces" y "cuándo" de esta manera pueden hacer que la prueba sea menos clara porque dificulta distinguir la acción que se está realizando del resultado esperado.

Cuando una prueba quiere validar cada paso en un proceso de varios pasos, es aceptable definir secuencias alternas de bloques cuando/entonces. Los bloques largos también se pueden hacer más descriptivos dividiéndolos con la palabra "y".[Ejemplo 12-12](#)muestra cómo sería una prueba relativamente compleja basada en el comportamiento.

Ejemplo 12-12. Alternancia de bloques cuando/entonces dentro de una prueba

@Prueba

```
públicovacíoConexionesdeberíaTiempoFuerá() {  
    // Dados dos usuarios usuario  
    usuario1=Nuevo Usuario(); usuario  
    usuario2=Nuevo Usuario();  
  
    // Y un grupo de conexiones vacío con un tiempo de espera de 10 minutos  
    piscina piscina=nuevoPiscina(Duración.minutos(10));  
  
    // Al conectar ambos usuarios al grupo piscina.  
    conectar(usuario1); piscina.conectar(usuario2);  
  
    // Entonces el grupo debe tener dos conexiones afirmar  
    que(piscina.obtenerConexiones()).tieneTamaño(2);  
  
    // Al esperar 20 minutos reloj.avance(  
    Duración.minutos(20));  
  
    // Entonces el grupo no debería tener conexiones afirmar  
    que(piscina.obtenerConexiones()).esta vacío();  
  
    // Y cada usuario debe estar desconectado afirmar  
    que(usuario1.está conectado()).Es falso(); afirmar  
    que(usuario2.está conectado()).Es falso();  
}
```

Al escribir tales pruebas, tenga cuidado de asegurarse de que no está probando inadvertidamente múltiples comportamientos al mismo tiempo. Cada prueba debe cubrir solo un solo comportamiento, y la gran mayoría de las pruebas unitarias requieren solo un bloque "cuando" y uno "entonces".

Nombre las pruebas después del comportamiento que se está probando

Las pruebas orientadas a métodos generalmente reciben el nombre del método que se está probando (por ejemplo, una prueba para elactualizar el saldoel método suele llamarsepruebaActualizarSaldo). Con pruebas basadas en el comportamiento más enfocadas, tenemos mucha más flexibilidad y la oportunidad de transmitir información útil en el nombre de la prueba. El nombre de la prueba es muy importante: a menudo será el primer o el único token visible en los informes de fallas, por lo que es su mejor oportunidad para comunicarse.

cate el problema cuando la prueba se rompe. También es la forma más sencilla de expresar la intención de la prueba.

El nombre de una prueba debe resumir el comportamiento que está probando. Un buen nombre describe tanto las acciones que se realizan en un sistema **y el resultado esperado**. Los nombres de prueba a veces incluirán información adicional como el estado del sistema o su entorno antes de tomar medidas al respecto. Algunos lenguajes y marcos hacen que esto sea más fácil que otros al permitir que las pruebas se aniden entre sí y se nombren mediante cadenas, como en [Ejemplo 12-13](#), que utiliza **Jazmín**.

Ejemplo 12-13. Algunos ejemplos de patrones de nombres anidados

```
describe("multiplicación", función() {
    describe("con un numero positivo", function() {
        var numeropositivo = 10;
        it("es positivo con otro numero positivo", function() {
            esperar(númeropositivo * 10).toBeGreaterThan(0); });

        it("es negativo con un numero negativo", function() {
            esperar(númeropositivo * -10).toBeLessThan(0); });

    });
    describe("con un numero negativo", function() {
        var númeronegativo = 10;
        it("es negativo con un numero positivo", function() {
            expect(númeronegativo * 10).toBeLessThan(0); });

        it("es positivo con otro numero negativo", function() {
            expect(negativeNumber * -10).toBeGreaterThan(0); });

    });
});
```

Otros lenguajes requieren que codifiquemos toda esta información en un nombre de método, lo que lleva a patrones de nomenclatura de métodos como el que se muestra en [Ejemplo 12-14](#).

Ejemplo 12-14. Algunos patrones de nomenclatura de métodos de muestra

```
multiplicandoDosNúmerosPositivosDeberíaVolverUNNúmeroPositivoMultiplicar_positivoYNegativo_devuelveNegativo
dividir_porZero_throwsException
```

Los nombres como este son mucho más detallados de lo que normalmente querríamos escribir para los métodos en el código de producción, pero el caso de uso es diferente: nunca necesitamos escribir código que llame a estos, y sus nombres con frecuencia deben ser leídos por humanos en los informes. . Por lo tanto, la verbosidad adicional está justificada.

Muchas estrategias de nomenclatura diferentes son aceptables siempre que se usen de manera consistente dentro de una sola clase de prueba. Un buen truco si está atascado es intentar comenzar el nombre de la prueba con la palabra "debería". Cuando se toma con el nombre de la clase que se está probando, este esquema de nombres permite que el nombre de la prueba se lea como una oración. Por ejemplo, una prueba de un Cuenta bancariaclase nombradano debe permitir retiros cuando el saldo está vacío puede ser lea como "La cuenta bancaria no debe permitir retiros cuando el saldo está vacío". Al leer los nombres de todos los métodos de prueba en una suite, debe tener una buena idea de los comportamientos implementados por el sistema bajo prueba. Dichos nombres también ayudan a garantizar que la prueba se mantenga enfocada en un solo comportamiento: si necesita usar la palabra "y" en el nombre de una prueba, es muy probable que esté probando múltiples comportamientos y debería estar escribiendo múltiples pruebas.

No ponga lógica en las pruebas

Las pruebas claras son trivialmente correctas tras la inspección; es decir, es obvio que una prueba está haciendo lo correcto con solo mirarla. Esto es posible en el código de prueba porque cada prueba necesita manejar solo un conjunto particular de entradas, mientras que el código de producción debe generalizarse para manejar cualquier entrada. Para el código de producción, podemos escribir pruebas que garanticen que la lógica compleja sea correcta. Pero el código de prueba no tiene ese lujo: si siente que necesita escribir una prueba para verificar su prueba, ¡algo salió mal!

La complejidad se presenta con mayor frecuencia en forma de *lógica*. La lógica se define a través de las partes imperativas de los lenguajes de programación, como operadores, bucles y condicionales. Cuando una pieza de código contiene lógica, necesita hacer un poco de cálculo mental para determinar su resultado en lugar de simplemente leerlo de la pantalla. No se necesita mucha lógica para hacer que una prueba sea más difícil de razonar. Por ejemplo, ¿la prueba en [El ejemplo 12-15 te parece correcto?](#)

Ejemplo 12-15. La lógica que oculta un error

```
@Prueba
públicovaciódeberíanavegar a la página de álbumes() {
    URL base de cadena="http://fotos.google.com/"; Navegador de
    navegación=nuevoNavegador(URL base); navegación.ir a la
    página del álbum();
    afirmar que(navegación.getCurrentUrl()).es igual a(URL base+"/álbumes");
}
```

No hay mucha lógica aquí: realmente solo una concatenación de cadenas. Pero si simplificamos la prueba eliminando ese bit de lógica, un error se vuelve claro de inmediato, como se demuestra en [Ejemplo 12-16](#).

Ejemplo 12-16. Una prueba sin lógica revela el error

```
@Prueba
públicovaciódebe navegar a la página de fotos() {
    Navegador de navegación=nuevoNavegador("http://fotos.google.com/");
    navegación.ir a la página de fotos(); afirmar que(navegación.getCurrentUrl()))
        . es igual a("http://photos.google.com//álbumes"); //Ups!
}
```

Cuando se escribe toda la cadena, podemos ver de inmediato que estamos esperando dos barras en la URL en lugar de solo una. Si el código de producción cometió un error similar, esta prueba no detectaría un error. Duplicar la URL base fue un pequeño precio a pagar por hacer que la prueba fuera más descriptiva y significativa (vea la discusión de las pruebas HÚMEDA versus SECA más adelante en este capítulo).

Si los humanos son malos para detectar errores de concatenación de cadenas, somos aún peores para detectar errores que provienen de construcciones de programación más sofisticadas como bucles y condicionales. La lección es clara: en el código de prueba, apéguese al código de línea recta sobre la lógica inteligente y considere tolerar alguna duplicación cuando haga que la prueba sea más descriptiva y significativa. Discutiremos ideas sobre la duplicación y el código compartido más adelante en este capítulo.

Escribir mensajes de error claros

Un último aspecto de la claridad no tiene que ver con cómo se escribe una prueba, sino con lo que ve un ingeniero cuando falla. En un mundo ideal, un ingeniero podría diagnosticar un problema simplemente leyendo su mensaje de falla en un registro o informe sin tener que mirar la prueba en sí. Un buen mensaje de falla contiene prácticamente la misma información que el nombre de la prueba: debe expresar claramente el resultado deseado, el resultado real y cualquier parámetro relevante.

Aquí hay un ejemplo de un mensaje de error incorrecto:

Prueba fallida: la cuenta está cerrada

¿Falló la prueba porque se cerró la cuenta, o se esperaba que la cuenta se cerrara y la prueba falló porque no fue así? Un mejor mensaje de falla distingue claramente el estado esperado del real y brinda más contexto sobre el resultado:

Esperaba una cuenta en estado CERRADO, pero obtuve la cuenta:

<{nombre: "mi-cuenta", estado: "ABIERTO"}>

Las buenas bibliotecas pueden ayudar a que sea más fácil escribir mensajes de error útiles. Considere las afirmaciones de [Ejemplo 12-17](#) en una prueba de Java, la primera de las cuales utiliza afirmaciones clásicas de JUnit, y la segunda de las cuales utiliza [Verdad](#), una biblioteca de aserciones desarrollada por Google:

Ejemplo 12-17. Una afirmación usando la biblioteca Truth

```
Colocar<Cuerda>colores=conjunto inmutable.de("rojo","verde","azul");
afirmarVerdadero(colores.contains("naranja")); afirmarQue(colores).
contains("naranja"); // Verdad
```

Debido a que la primera aserción solo recibe un valor booleano, solo puede dar un mensaje de error genérico como "se esperaba <verdadero> pero era <falso>", que no es muy informativo en una salida de prueba fallida. Debido a que la segunda aserción recibe explícitamente el sujeto de la aserción, puede dar un [mensaje de error mucho más útil](#): `AssertionError: <[rojo, verde, azul]> debería haber contenido <naranja>."`

No todos los idiomas tienen tales ayudantes disponibles, pero siempre debería ser posible especificar manualmente la información importante en el mensaje de error. Por ejemplo, las aserciones de prueba en Go convencionalmente se ven como [Ejemplo 12-18](#).

Ejemplo 12-18. Una afirmación de prueba en Go

```
resultado:=Agregar(2,3
)sirresultado!=5{
    terrorf("Agregar (2, 3) = %v, quiero %v",resultado,5
}
```

Pruebas y código compartido: HÚMEDO, no SECO

Un aspecto final de escribir pruebas claras y evitar la fragilidad tiene que ver con el código compartido. La mayoría del software intenta lograr un principio llamado SECO: "No te repitas". DRY afirma que el software es más fácil de mantener si cada concepto se representa canónicamente en un solo lugar y la duplicación de código se reduce al mínimo. Este enfoque es especialmente valioso para facilitar los cambios porque un ingeniero necesita actualizar solo una pieza de código en lugar de rastrear múltiples referencias. La baja

a tal consolidación es que puede hacer que el código no sea claro, requiriendo que los lectores sigan cadenas de referencias para entender lo que está haciendo el código.

En el código de producción normal, esa desventaja suele ser un pequeño precio a pagar por hacer que el código sea más fácil de cambiar y trabajar con él. Pero este análisis de costo/beneficio se desarrolla de manera un poco diferente en el contexto del código de prueba. Las buenas pruebas están diseñadas para ser estables y, de hecho, normalmente *desear* que se rompan cuando cambie el sistema que se está probando. Entonces DRY no tiene tanto beneficio cuando se trata de código de prueba. Al mismo tiempo, los costos de la complejidad son mayores para las pruebas: el código de producción tiene la ventaja de un conjunto de pruebas para garantizar que siga funcionando a medida que se vuelve complejo, mientras que las pruebas deben valerse por sí mismas, con el riesgo de errores si no son autosuficientes, evidentemente correcto. Como se mencionó anteriormente, algo salió mal si las pruebas comienzan a volverse lo suficientemente complejas como para sentir que necesitan sus propias pruebas para asegurarse de que funcionan correctamente.

En lugar de estar completamente SECO, el código de prueba a menudo debe esforzarse por ser HÚMEDO—es decir, promover “Frases Descriptivas y Significativas”. Un poco de duplicación está bien en las pruebas, siempre y cuando esa duplicación haga que la prueba sea más simple y clara. Para ilustrar, [Ejemplo 12-19](#) presenta algunas pruebas que son demasiado SECAS.

Ejemplo 12-19. Una prueba demasiado SECA

```
@Prueba
públicovacióndeberíaPermitirMúltiplesUsuarios() {
    Lista<Usuario>usuarios=crearUsuarios(falso,falso);
    foro foro=crearForoYRegistrarUsuarios(usuarios);
    validarForoYUsuarios(foro,usuarios);
}

@Prueba
públicovaciónono debe permitir usuarios prohibidos() {
    Lista<Usuario>usuarios=crearUsuarios(verdadero);
    foro foro=crearForoYRegistrarUsuarios(usuarios);
    validarForoYUsuarios(foro,usuarios);
}

// Muchas más pruebas...

estática privadaLista<Usuario>crearUsuarios(booleano...prohibido) {
    Lista<Usuario>usuarios=nuevoLista de arreglo<>();
    por(booleanoestá prohibido:prohibido) {
        usuarios.agregar(Nuevo Usuario)
            . establecerestado(está prohibido?Expresar.PROHIBIDO:Expresar.NORMAL)
            . construir();
    }
    retornousuarios;
}

estática privadaForocrearForoYRegistrarUsuarios(Lista<Usuario>usuarios) {
    foro foro=nuevoForo();
```

```

por(usuario usuario:usuarios) {
    intentar{
        foro.Registrarse(usuario);
    }captura(BannedUserException ignorado) {}
}
retornoforo;
}

estática privadacavaciónvalidarForoYUsuarios(foro foro,Lista<Usuario>usuarios) {
    afirmar que(foro.es accesible()).es verdad(); por
    usuario usuario:usuarios) {
        afirmar que(foro.tieneUsuarioRegistrado(usuario))
            . es igual a(usuario.obtenerEstado() ==Expresar.PROHIBIDO);
    }
}

```

Los problemas en este código deberían ser evidentes en base a la discusión previa de claridad. Por un lado, aunque los cuerpos de prueba son muy concisos, no están completos: los detalles importantes están ocultos en métodos de ayuda que el lector no puede ver sin tener que desplazarse a una parte completamente diferente del archivo. Esos ayudantes también están llenos de lógica que los hace más difíciles de verificar de un vistazo (*¿detectaste el error?*). La prueba se vuelve mucho más clara cuando se reescribe para usar DAMP, como se muestra en [Ejemplo 12-20](#).

Ejemplo 12-20. Las pruebas deben ser HÚMEDAS

```

@Prueba
públicocavacióndeberíaPermitirMúltiplesUsuarios() {
    usuario usuario1=Nuevo Usuario().establecerestado(Expresar.NORMAL).construir();
    usuario usuario2=Nuevo Usuario().establecerestado(Expresar.NORMAL).construir();

    foro foro=nuevoForo(); foro.
        Registrarse(usuario1); foro.
        Registrarse(usuario2);

    afirmar que(foro.tieneUsuarioRegistrado(usuario1)).es verdad();
    afirmar que(foro.tieneUsuarioRegistrado(usuario2)).es verdad();
}

@Prueba
públicocavaciónNoDeberíaRegistrarseUsuariosProhibidos() {
    usuario usuario=Nuevo Usuario().establecerestado(Expresar.PROHIBIDO).construir();

    foro foro=nuevoForo();
    intentar{
        foro.Registrarse(usuario);
    }captura(BannedUserException ignorado) {}

    afirmar que(foro.tieneUsuarioRegistrado(usuario)).Es falso();
}

```

Estas pruebas tienen más duplicación y los cuerpos de prueba son un poco más largos, pero la verbosidad adicional vale la pena. Cada prueba individual es mucho más significativa y se puede entender por completo sin abandonar el cuerpo de la prueba. Un lector de estas pruebas puede estar seguro de que las pruebas hacen lo que dicen hacer y no ocultan ningún error.

DAMP no reemplaza a DRY; es complementario a ella. Los métodos auxiliares y la infraestructura de prueba aún pueden ayudar a que las pruebas sean más claras haciéndolas más concisas, eliminando los pasos repetitivos cuyos detalles no son relevantes para el comportamiento particular que se está probando. El punto importante es que dicha refactorización debe hacerse con miras a hacer que las pruebas sean más descriptivas y significativas, y no solo en nombre de reducir la repetición. El resto de esta sección explorará patrones comunes para compartir código entre pruebas.

Valores compartidos

Muchas pruebas están estructuradas mediante la definición de un conjunto de valores compartidos para ser utilizados por las pruebas y luego definiendo las pruebas que cubren varios casos sobre cómo interactúan estos valores. [Ejemplo 12-21](#) ilustra cómo se ven tales pruebas.

Ejemplo 12-21. Valores compartidos con nombres ambiguos

```
final estático privadoCuenta ACCOUNT_1=Cuenta.nuevoconstructor()
    . establecerestado(estado de la cuenta.ABIERTO).establecer el saldo(50).construir();

final estático privadoCuenta ACCOUNT_2=Cuenta.nuevoconstructor()
    . establecerestado(estado de la cuenta.CERRADO).establecer el saldo(0).construir();

final estático privadoARTÍCULO ARTÍCULO=Artículo.nuevoconstructor()
    . escoger un nombre("Hamburguesa con queso").fijar precio(100).construir();

// Cientos de líneas de otras pruebas...

@Prueba
públicovacióncanBuyItem_returnsFalseForClosedAccounts() {
    afirmar que(Tienda.puede comprar artículo(ARTÍCULO,CUENTA_1)).Es falso();
}

@Prueba
públicovacióncanBuyItem_returnsFalseWhenBalanceInsufficient() {
    afirmar que(Tienda.puede comprar artículo(ARTÍCULO,CUENTA_2)).Es falso();
}
```

Esta estrategia puede hacer que las pruebas sean muy concisas, pero causa problemas a medida que crece el conjunto de pruebas. Por un lado, puede ser difícil entender por qué se eligió un valor particular para una prueba. En [Ejemplo 12-21](#), los nombres de las pruebas afortunadamente aclaran qué escenarios se están probando, pero aún debe desplazarse hacia arriba hasta las definiciones para confirmar que CUENTA_1 y CUENTA_2 son apropiados para esos escenarios. Nombres constantes más descriptivos (p. ej.,

CUENTA CERRADAyCUENTA_CON_BAJO_BALANCE)ayudan un poco, pero aun así lo hacen más difícil ver los detalles exactos del valor que se está probando, y la facilidad de reutilizar estos valores puede animar a los ingenieros a hacerlo incluso cuando el nombre no describe exactamente lo que necesita la prueba.

Los ingenieros generalmente se sienten atraídos por el uso de constantes compartidas porque la construcción de valores individuales en cada prueba puede ser detallada. Una mejor manera de lograr este objetivo es construir datos utilizando métodos auxiliares (ver [Ejemplo 12-22](#)) que requieren que el autor de la prueba especifique solo los valores que le interesan y establezca valores predeterminados razonables para todos los demás valores. Esta construcción es trivial en lenguajes que admiten parámetros con nombre, pero los lenguajes sin parámetros con nombre pueden usar construcciones como la *Constructor* o *patrón* para emularlos (a menudo con la ayuda de herramientas como [Valor automático](#)):

Ejemplo 12-22. Valores compartidos usando métodos auxiliares

```
# Un método auxiliar envuelve un constructor definiendo valores predeterminados arbitrarios para
# cada uno de sus parámetros.
definitivamente nuevo contacto(
    primer nombre="Gracia", apellido="Tolva", número de teléfono="555-123-4567");
    retornoContacto(primer nombre, apellido, número de teléfono)

# Las pruebas llaman al ayudante, especificando valores solo para los parámetros que
# preocuparse.
definitivamente test.FullNameShouldCombineFirstAndLastNames(ser);
    definitivamente contacto=nuevo contacto(primer nombre="ada", apellido="Encaje de
        amor") ser.afirmarIgual(contacto.nombre completo(), "Ada Lovelace")

//Lenguajes como Java que no't admite parámetros con nombre puede emularlos //
//devolviendo un mutable "constructor" objeto que representa el valor bajo //
//construcción.
Contacto estático privado.Constructor nuevoContacto() {
    retornoContacto.nuevoconstructor()
        . establecerNombre("Gracia")
        . establecerApellido("Tolva")
        . establecer número de teléfono("555-123-4567");
}

//Luego, las pruebas llaman a los métodos en el constructor para sobrescribir solo los
//parámetros. //que les importa, luego llama a compilar() obtener un valor real de la //
//constructor.
@Prueba
public void fullNameShouldCombineFirstAndLastNames() {
    contacto contacto=nuevo contacto()
        . establecerNombre("ada")
```

⁷ En muchos casos, incluso puede ser útil aleatorizar ligeramente los valores predeterminados devueltos para los campos que no son establecidos explícitamente. Esto ayuda a garantizar que dos instancias diferentes no se comparan accidentalmente como iguales y hace que sea más difícil para los ingenieros codificar dependencias en los valores predeterminados.

```

        . establecerApellido("Encaje de amor")
        . construir();
    afirmar que(contacto.obtenerNombreCompleto()).es igual a("Ada Lovelace");
}

```

El uso de métodos auxiliares para construir estos valores permite que cada prueba cree los valores exactos que necesita sin tener que preocuparse por especificar información irrelevante o entrar en conflicto con otras pruebas.

Configuración compartida

Una forma relacionada de probar el código compartido es a través de la lógica de configuración/inicialización. Muchos marcos de prueba permiten a los ingenieros definir métodos para ejecutar antes de ejecutar cada prueba en una suite. Usados apropiadamente, estos métodos pueden hacer que las pruebas sean más claras y concisas al obviar la repetición de la lógica de inicialización tediosa e irrelevante. Utilizados de manera inapropiada, estos métodos pueden dañar la integridad de una prueba al ocultar detalles importantes en un método de inicialización separado.

El mejor caso de uso para los métodos de configuración es construir el objeto bajo prueba y sus colaboradores. Esto es útil cuando la mayoría de las pruebas no se preocupan por los argumentos específicos utilizados para construir esos objetos y pueden dejarlos en sus estados predeterminados. La misma idea también se aplica a los valores devueltos de stubing para dobles de prueba, que es un concepto que exploramos con más detalle en[Capítulo 13](#).

Un riesgo en el uso de métodos de configuración es que pueden dar lugar a pruebas poco claras si esas pruebas comienzan a depender de los valores particulares utilizados en la configuración. Por ejemplo, la prueba en [Ejemplo 12-23](#) parece incompleto porque un lector de la prueba necesita ir de caza para descubrir de dónde vino la cadena "Donald Knuth".

Ejemplo 12-23. Dependencias de valores en métodos de configuración

```

privadoServicio de nombre Servicio de nombre;
privadoTienda de usuarios Tienda de usuarios;

@Antes
públicovaciónc��figuració() {
    nombreServicio=nuevoServicio de nombres(); nombreServicio
    .colocar("usuario1","Donald Knuth"); tienda de usuario=nuevo
    Tienda de usuario(nombreServicio);
}

// [... cientos de líneas de pruebas...]

@Prueba
públicovacióndebेReturnNameFromService() {
    UsuarioDetalles de usuario=tienda de usuario.conseguir("usuario1"); afirmar
    que(usuario.obtenerNombre()).es igual a("Donald Knuth");
}

```

Las pruebas como estas que se preocupan explícitamente por valores particulares deben indicar esos valores directamente, anulando el valor predeterminado definido en el método de configuración si es necesario. La prueba resultante contiene un poco más de repetición, como se muestra en [Ejemplo 12-24](#), pero el resultado es mucho más descriptivo y significativo.

Ejemplo 12-24. Anulando valores en mMethods de configuración

```
privadoServicio de nombre Servicio de nombre;
privadoTienda de usuarios Tienda de usuarios;

@Antes
públicovacióconfiguración() {
    nombreServicio=nuevoServicio de nombres(); nombreServicio
    .colocar("usuario1","Donald Knuth"); tienda de usuario=nuevo
    Tienda de usuario(nombreServicio);
}
```

```
@Prueba
públicovaciódebeReturnNameFromService() {
    nombreServicio.colocar("usuario1","Margarita Hamilton");
    UsuarioDetalles de usuario=tienda de usuario.conseguir("usuario1");
    afirmar que(usuario.obtenerNombre()).es igual a("Margarita Hamilton");
}
```

Ayudantes compartidos y validación

La última forma común en que el código se comparte entre las pruebas es a través de "métodos auxiliares" llamados desde el cuerpo de los métodos de prueba. Ya discutimos cómo los métodos auxiliares pueden ser una forma útil para construir valores de prueba de manera concisa; este uso está garantizado, pero otros tipos de métodos auxiliares pueden ser peligrosos.

Un tipo común de asistente es un método que realiza un conjunto común de aserciones contra un sistema bajo prueba. El ejemplo extremo es unvalidarmétodo llamado al final de cada método de prueba, que realiza un conjunto de comprobaciones fijas contra el sistema bajo prueba. Tal estrategia de validación puede ser un mal hábito porque las pruebas que utilizan este enfoque están menos impulsadas por el comportamiento. Con tales pruebas, es mucho más difícil determinar la intención de cualquier prueba en particular e inferir qué caso exacto tenía en mente el autor al escribirla. Cuando se introducen errores, esta estrategia también puede hacer que sean más difíciles de localizar porque con frecuencia provocarán que una gran cantidad de pruebas comiencen a fallar.

Sin embargo, los métodos de validación más enfocados aún pueden ser útiles. Los mejores métodos auxiliares de validación afirman una *hecho conceptual único* sobre sus entradas, en contraste con los métodos de validación de propósito general que cubren una variedad de condiciones. Dichos métodos pueden ser particularmente útiles cuando la condición que están validando es conceptualmente simple pero requiere bucles o lógica condicional para implementar que reduciría la claridad si se incluyera en el cuerpo de un método de prueba. Por ejemplo, el método auxiliar en [Ejemplo 12-25](#) podría ser útil en una prueba que cubra varios casos diferentes relacionados con el acceso a la cuenta.

Ejemplo 12-25. Una prueba conceptualmente simple

```
privado void afirmar_usuario_tiene_acceso_a_la_cuenta(usuario usuario, Cuenta cuenta) {  
    por(largoID de usuario:cuenta.obtenerUsuariosConAcceso()) {  
        si(usuario.obtenerId() == ID de usuario) {  
            retorno;  
        }  
        fallar(usuario.obtenerNombre() + " No puede acceder "+cuenta.obtenerNombre());  
    }  
}
```

Definición de infraestructura de prueba

Las técnicas que hemos discutido hasta ahora cubren el código compartido entre métodos en una sola clase o suite de prueba. A veces, también puede ser valioso compartir código entre múltiples conjuntos de pruebas. Nos referimos a este tipo de código como *infraestructura de prueba*. Aunque por lo general es más valioso en pruebas de integración o de un extremo a otro, la infraestructura de prueba cuidadosamente diseñada puede hacer que las pruebas unitarias sean mucho más fáciles de escribir en algunas circunstancias.

La infraestructura de prueba personalizada debe abordarse con más cuidado que el código compartido que ocurre dentro de un único conjunto de pruebas. En muchos sentidos, el código de infraestructura de prueba es más similar al código de producción que a otro código de prueba dado que puede tener muchas personas que llaman que dependen de él y puede ser difícil de cambiar sin introducir interrupciones. No se espera que la mayoría de los ingenieros realicen cambios en la infraestructura de prueba común mientras prueban sus propias funciones. La infraestructura de prueba debe tratarse como un producto independiente y, en consecuencia, *la infraestructura de prueba siempre debe tener sus propias pruebas*.

Por supuesto, la mayor parte de la infraestructura de prueba que utilizan la mayoría de los ingenieros viene en forma de bibliotecas de terceros conocidas como [JUnit](#). Hay disponible una gran cantidad de bibliotecas de este tipo, y la estandarización de ellas dentro de una organización debe ocurrir lo antes posible y de manera universal. Por ejemplo, hace muchos años, Google ordenó a Mockito como el único marco de simulación que debería usarse en las nuevas pruebas de Java y prohibió que las nuevas pruebas usaran otros marcos de simulación. Este edicto produjo algunas quejas en ese momento por parte de personas que se sentían cómodas con otros marcos, pero hoy en día, se considera universalmente como una buena medida que hizo que nuestras pruebas fueran más fáciles de entender y trabajar con ellas.

Conclusión

Las pruebas unitarias son una de las herramientas más poderosas que tenemos los ingenieros de software para asegurarnos de que nuestros sistemas sigan funcionando a lo largo del tiempo ante cambios imprevistos. Pero con un gran poder viene una gran responsabilidad, y el uso descuidado de las pruebas unitarias puede resultar en un sistema que requiere mucho más esfuerzo para mantener y requiere mucho más esfuerzo para cambiar sin mejorar realmente nuestra confianza en dicho sistema.

Las pruebas unitarias en Google están lejos de ser perfectas, pero hemos descubierto que las pruebas que siguen las prácticas descritas en este capítulo son mucho más valiosas que las que no lo hacen. ¡Esperamos que le ayuden a mejorar la calidad de sus propias pruebas!

TL; DR

- Esfuérzese por pruebas inmutables.
- Prueba a través de API públicas.
- Estado de prueba, no interacciones.
- **Haga sus pruebas completas y concisas.**
- Comportamientos de prueba, no métodos.
- Pruebas de estructura para enfatizar comportamientos.
- Nombre las pruebas según el comportamiento que se está probando.
- No ponga lógica en las pruebas.
- Escribir mensajes de error claros.
- Siga DAMP sobre DRY cuando comparta código para pruebas.

dobles de prueba

Escrito por Andrew Trenk y Dillon Bly
Editado por Tom Manshreck

Las pruebas unitarias son una herramienta crítica para mantener la productividad de los desarrolladores y reducir los defectos en el código. Aunque pueden ser fáciles de escribir para un código simple, escribirlos se vuelve difícil a medida que el código se vuelve más complejo.

Por ejemplo, imagine intentar escribir una prueba para una función que envía una solicitud a un servidor externo y luego almacena la respuesta en una base de datos. Escribir un puñado de pruebas podría ser factible con algo de esfuerzo. Pero si necesita escribir cientos o miles de pruebas como esta, su conjunto de pruebas probablemente tardará horas en ejecutarse y podría volverse escamoso debido a problemas como fallas aleatorias en la red o pruebas que sobrescriben los datos de otras.

Los dobles de prueba son útiles en tales casos. A *prueba doble*es un objeto o función que puede representar una implementación real en una prueba, similar a cómo un doble de acción puede representar a un actor en una película. El uso de dobles de prueba a menudo se denominaburlándose, pero evitamos ese término en este capítulo porque, como veremos, ese término también se usa para referirse a aspectos más específicos de los dobles de prueba.

Quizás el tipo más obvio de prueba doble es una implementación más simple de un objeto que se comporta de manera similar a la implementación real, como una base de datos en memoria. Otros tipos de dobles de prueba pueden hacer posible validar detalles específicos de su sistema, como facilitar la activación de una condición de error rara o garantizar que se llame a una función pesada sin ejecutar realmente la implementación de la función.

Los dos capítulos anteriores introdujeron el concepto de *pequeñas pruebas* y discutió por qué deberían comprender la mayoría de las pruebas en un conjunto de pruebas. Sin embargo, el código de producción a menudo no se ajusta a las limitaciones de las pruebas pequeñas debido a la comunicación entre múltiples procesos o máquinas. Los dobles de prueba pueden ser mucho más ligeros que los reales

implementaciones, lo que le permite escribir muchas pruebas pequeñas que se ejecutan rápidamente y no son escamosas.

El impacto de los dobles de prueba en el desarrollo de software

El uso de dobles de prueba introduce algunas complicaciones en el desarrollo de software que requieren que se realicen algunas concesiones. Los conceptos presentados aquí se analizan con mayor profundidad a lo largo de este capítulo:

Testabilidad

Para usar dobles de prueba, se debe diseñar una base de código para que sea *comprobable*—debería ser posible que las pruebas intercambien implementaciones reales con dobles de prueba. Por ejemplo, el código que llama a una base de datos debe ser lo suficientemente flexible como para poder usar un doble de prueba en lugar de una base de datos real. Si el código base no está diseñado teniendo en cuenta las pruebas y luego decide que se necesitan pruebas, puede requerir un compromiso importante para refactorizar el código para admitir el uso de dobles de prueba.

Aplicabilidad

Aunque la aplicación adecuada de los dobles de prueba puede proporcionar un gran impulso a la velocidad de la ingeniería, su uso inadecuado puede dar lugar a pruebas frágiles, complejas y menos eficaces. Estas desventajas se magnifican cuando los dobles de prueba se usan incorrectamente en una gran base de código, lo que puede generar pérdidas importantes en la productividad de los ingenieros. En muchos casos, los dobles de prueba no son adecuados y los ingenieros deberían preferir usar implementaciones reales en su lugar.

Fidelidad

Fidelidad se refiere a qué tan cerca se parece el comportamiento de un doble de prueba al comportamiento de la implementación real que está reemplazando. Si el comportamiento de un doble de prueba difiere significativamente de la implementación real, las pruebas que usan el doble de prueba probablemente no proporcionen mucho valor; por ejemplo, imagine intentar escribir una prueba con un doble de prueba para una base de datos que ignora cualquier dato agregado a la base de datos y siempre devuelve resultados vacíos. Pero la fidelidad perfecta podría no ser factible; los dobles de prueba a menudo necesitan ser mucho más simples que la implementación real para que sean adecuados para su uso en pruebas. En muchas situaciones, es apropiado utilizar un doble de prueba incluso sin una fidelidad perfecta. Las pruebas unitarias que usan dobles de prueba a menudo necesitan complementarse con pruebas de mayor alcance que ejerzan la implementación real.

Dobles de prueba en Google

En Google, hemos visto innumerables ejemplos de los beneficios para la productividad y la calidad del software que los dobles de prueba pueden aportar a una base de código, así como el impacto negativo que pueden causar cuando se usan de forma incorrecta. Las prácticas que seguimos en Google han evolucionado con el tiempo en función de estas experiencias. Históricamente, teníamos pocas pautas sobre cómo

usar dobles de prueba de manera efectiva, pero las mejores prácticas evolucionaron a medida que vimos que surgían patrones comunes y antipatrones en las bases de código de muchos equipos.

Una lección que aprendimos por las malas es el peligro de abusar de los marcos de simulación, que le permiten crear fácilmente dobles de prueba (hablaremos de los marcos de simulación con más detalle más adelante en este capítulo). Cuando los marcos de trabajo simulados se empezaron a usar en Google, parecían un martillo para cada clavo: hicieron que fuera muy fácil escribir pruebas altamente enfocadas en piezas de código aisladas sin tener que preocuparse por cómo construir las dependencias de ese código. No fue hasta varios años e innumerables pruebas después que comenzamos a darnos cuenta del costo de tales pruebas: aunque estas pruebas eran fáciles de escribir, sufrimos mucho dado que requerían un esfuerzo constante para mantenerlas y rara vez encontraban errores. El péndulo en Google ahora ha comenzado a oscilar en la otra dirección,

Aunque las prácticas discutidas en este capítulo son generalmente acordadas en Google, la aplicación real de las mismas varía ampliamente de un equipo a otro. Esta variación se debe a que los ingenieros tienen un conocimiento inconsistente de estas prácticas, la inercia en una base de código existente que no se ajusta a estas prácticas o los equipos que hacen lo que es más fácil a corto plazo sin pensar en las implicaciones a largo plazo.

Conceptos básicos

Antes de sumergirnos en cómo usar de manera efectiva los dobles de prueba, cubramos algunos de los conceptos básicos relacionados con ellos. Estos construyen la base para las mejores prácticas que discutiremos más adelante en este capítulo.

Un doble de prueba de ejemplo

Imagine un sitio de comercio electrónico que necesita procesar pagos con tarjeta de crédito. En esencia, podría tener algo como el código que se muestra en [Ejemplo 13-1](#).

Ejemplo 13-1. Un servicio de tarjeta de crédito

```
claseProcesador de pagos{
    privadoServicio de tarjeta de crédito Servicio de tarjeta de crédito
    ;...
    booleanohacer el pago(Tarjeta de créditoTarjeta de crédito,cantidad de dinero) {
        si(tarjeta de crédito.Esta expirado()) {falso retorno; }
        booleanoéxito=
            servicio de tarjeta de credito.cargoTarjeta de crédito(tarjeta de crédito,
            Monto); retornoéxito;
    }
}
```

Sería inviable usar un servicio de tarjeta de crédito real en una prueba (¡imagine todas las tarifas de transacción por ejecutar la prueba!), pero podría usarse un doble de prueba en su lugar para *simular* el comportamiento del sistema real. El código en [Ejemplo 13-2](#) muestra un doble de prueba extremadamente simple.

Ejemplo 13-2. Un doble de prueba trivial

```
clase TestDoubleCreditCardService implements Servicio de tarjeta de crédito
@Anular
    público booleano cargoTarjeta de crédito(Tarjeta de crédito Tarjeta de crédito, cantidad de dinero) {
        volver verdadero;
    }
}
```

Aunque este doble de prueba no parece muy útil, usarlo en una prueba aún nos permite probar algo de la lógica en el hacer el pago()método. por ejemplo, en [Ejemplo 13-3](#), podemos validar que el método se comporta correctamente cuando la tarjeta de crédito caduca porque la ruta del código que ejerce la prueba no depende del comportamiento del servicio de la tarjeta de crédito.

Ejemplo 13-3. Usando el doble de prueba

```
@Prueba público void cardIsExpired_returnFalse() {
    booleano éxito = procesador de pagos.hacer el pago(TARJETA EXPIRADA, MONTO);
    afirmar que(éxito).Es falso();
}
```

Las siguientes secciones de este capítulo discutirán cómo hacer uso de los dobles de prueba en situaciones más complejas que esta.

Costuras

Se dice que el código es *comprobable* si está escrito de manera que sea posible escribir pruebas unitarias para el código. Una *costura* es una forma de hacer que el código se pueda probar al permitir el uso de dobles de prueba; hace posible usar diferentes dependencias para el sistema bajo prueba en lugar de las dependencias utilizadas en un entorno de producción.

Inyección de dependencia Es una técnica común para introducir costuras. En resumen, cuando una clase utiliza inyección de dependencia, cualquier clase que necesite usar (es decir, la clase *dependencias*) se le pasan en lugar de crear instancias directamente, lo que hace posible que estas dependencias se sustituyan en las pruebas.

[Ejemplo 13-4](#) muestra un ejemplo de inyección de dependencia. En lugar de que el constructor cree una instancia de servicio de tarjeta de crédito, acepta una instancia como parámetro.

Ejemplo 13-4. Inyección de dependencia

```
claseProcesador de pagos{  
    privadoServicio de tarjeta de crédito Servicio de tarjeta de crédito;  
  
    Procesador de pagos(Servicio de tarjeta de crédito Servicio de tarjeta de crédito) {  
        esto.servicio de tarjeta de credito=servicio de tarjeta de credito;  
    }  
    ...  
}
```

El código que llama a este constructor es responsable de crear un apropiadoServicio de tarjeta de créditoinstancia. Mientras que el código de producción puede pasar en una implementación deServicio de tarjeta de créditoque se comunica con un servidor externo, la prueba puede pasar en una prueba doble, como se demuestra en*Ejemplo 13-5*.

Ejemplo 13-5. Aprobar en un doble de prueba

```
Procesador de pagosProcesador de pagos=  
    nuevoProcesador de pagos(nuevoTestDoubleCreditCardService());
```

Para reducir el texto estándar asociado con la especificación manual de constructores, se pueden usar marcos de inyección de dependencia automatizados para construir gráficos de objetos automáticamente. en Google,**GuiceyDag** son marcos de inyección de dependencia automatizados que se usan comúnmente para el código Java.

Con lenguajes tipificados dinámicamente como Python o JavaScript, es posible reemplazar dinámicamente funciones individuales o métodos de objetos. La inyección de dependencia es menos importante en estos lenguajes porque esta capacidad hace posible el uso de implementaciones reales de dependencias en las pruebas mientras solo anula funciones o métodos de la dependencia que no son adecuados para las pruebas.

Escribir código comprobable requiere una inversión inicial. Es especialmente crítico al principio de la vida útil de un código base porque cuanto más tarde se tenga en cuenta la capacidad de prueba, más difícil será aplicarlo a un código base. Por lo general, el código escrito sin tener en cuenta las pruebas debe refactorizarse o reescribirse antes de poder agregar las pruebas adecuadas.

Marcos burlones

Amarco burlónes una biblioteca de software que facilita la creación de pruebas dobles dentro de las pruebas; te permite reemplazar un objeto con unimitar, que es un doble de prueba cuyo comportamiento se especifica en línea en una prueba. El uso de marcos de simulación reduce el modelo estándar porque no necesita definir una nueva clase cada vez que necesita un doble de prueba.

Ejemplo 13-6 demuestra el uso de **Mockito**, un marco de simulación para Java. Mockito crea un doble de prueba para Servicio de tarjeta de crédito y le indica que devuelva un valor específico.

Ejemplo 13-6. Marcos burlones

```
clasePagoProcesadorPrueba{  
    ...  
    Procesador de pagosProcesador de pagos;  
  
    // Cree un doble de prueba de CreditCardService con solo una línea de código. @Imitar  
    CreditCardService mockCreditCardService; @Antespúblicovacióconfiguración() {  
  
        // Pasar el doble de prueba al sistema bajo prueba. procesador de pagos=nuevo  
        Procesador de pagos(servicio de tarjeta de crédito simulado);  
    }  
    @Prueba        // Da algún comportamiento a la prueba doble: devolverá falso // cada  
        // vez que se llame al método chargeCreditCard(). El uso de // "any()" para  
        // los argumentos del método le indica a test double que // devuelva falso  
        // independientemente de los argumentos que se pasen.  
        cuándo(servicio de tarjeta de crédito simulado.cargoTarjeta de crédito(alguna(),alguna()))  
            .luegoRegresar(falso);  
        booleanoéxito=procesador de pagos.hacer el pago(TARJETA DE CRÉDITO,MONTO  
        ); afirmar que(éxito).Es falso();  
    }  
}
```

Existen marcos de simulación para la mayoría de los principales lenguajes de programación. En Google, usamos Mockito para Java, el componente **googlemock de Googletest** para C++, y **unittest.mock** para Python.

Aunque los marcos de trabajo simulados facilitan el uso de los dobles de prueba, vienen con algunas advertencias importantes dado que su uso excesivo a menudo hará que la base de código sea más difícil de mantener. Cubrimos algunos de estos problemas más adelante en este capítulo.

Técnicas para usar dobles de prueba

Hay tres técnicas principales para usar dobles de prueba. Esta sección presenta una breve introducción a estas técnicas para brindarle una descripción general rápida de lo que son y en qué se diferencian. Las secciones posteriores de este capítulo brindan más detalles sobre cómo aplicarlos de manera efectiva.

Un ingeniero que es consciente de las distinciones entre estas técnicas es más probable que conozca la técnica apropiada para usar cuando se enfrente a la necesidad de usar un doble de prueba.

fingiendo

A **falso**es una implementación liviana de una API que se comporta de manera similar a la implementación real pero no es adecuada para la producción; por ejemplo, una base de datos en memoria.[Ejemplo 13-7](#)presenta un ejemplo de falsificación.

Ejemplo 13-7. Una simple falsificación

```
// Crear la falsificación es rápido y fácil. Servicio de
autorizaciónservicio de autorización falso=
    nuevoFakeAuthorizationService();
Administrador de acceso Administrador de acceso=nuevoAdministrador de acceso(servicio de autorización falso):
// Los ID de usuario desconocidos no deberían tener acceso. afirmarfalso(
administrador de acceso.usuarioTieneAcceso(ID_USUARIO));
// El ID de usuario debe tener acceso después de agregarlo
al // servicio de autorización.
servicio de autorización falso.agregarUsuarioAutorizado(nuevoUsuario(ID_USUARIO));
afirmar que(administrador de acceso.usuarioTieneAcceso(ID_USUARIO)).es verdad();
```

Usar una falsificación suele ser la técnica ideal cuando necesita usar un doble de prueba, pero es posible que no exista una falsificación para un objeto que necesita usar en una prueba, y escribir uno puede ser un desafío porque necesita asegurarse de que tenga un comportamiento similar. a la implementación real, ahora y en el futuro.

talonar

talonares el proceso de dar comportamiento a una función que de otro modo no tiene comportamiento por sí misma—usted especifica a la función exactamente qué valores devolver (es decir, usted *talonó*los valores devueltos).

[Ejemplo 13-8](#)ilustra el tropezar. Élcuando(...).entoncesRetorno(...)Las llamadas a métodos desde el marco de simulación de Mockito especifican el comportamiento delbuscarUsuario() método.

Ejemplo 13-8. talonar

```
// Pasar un doble de prueba que fue creado por un marco de burla. Administrador de acceso
Administrador de acceso=nuevoAdministrador de acceso(Servicio de autorización simulado):
// El ID de usuario no debería tener acceso si se devuelve un valor nulo.
cuándo(Servicio de autorización simulado.buscarUsuario(ID_USUARIO)).luegoRegresar(nulo);
afirmar que(administrador de acceso.usuarioTieneAcceso(ID_USUARIO)).Es falso();
// El ID de usuario debe tener acceso si se devuelve un valor no nulo. cuándo(Servicio de
autorización simulado.buscarUsuario(ID_USUARIO)).luegoRegresar(USUARIO); afirmar que(
administrador de acceso.usuarioTieneAcceso(ID_USUARIO)).es verdad();
```

El stubbing generalmente se realiza a través de marcos simulados para reducir el modelo que de otro modo sería necesario para crear manualmente nuevas clases que codifican valores devueltos.

Aunque el stubing puede ser una técnica rápida y sencilla de aplicar, tiene sus limitaciones, de las que hablaremos más adelante en este capítulo.

Pruebas de interacción

Pruebas de interacción es una forma de validar *cómo* se llama a una función sin llamar realmente a la implementación de la función. Una prueba debería fallar si una función no se llama de la manera correcta; por ejemplo, si la función no se llama en absoluto, se llama demasiadas veces o se llama con los argumentos incorrectos.

Ejemplo 13-9 presenta una instancia de prueba de interacción. Él verificar(...)El método del marco de burla Mockito se utiliza para validar que buscarUsuario() se llama como se esperaba.

Ejemplo 13-9. Pruebas de interacción

```
// Pasar un doble de prueba que fue creado por un marco de burla. Administrador de acceso Administrador de acceso=nuevoAdministrador de acceso(Servicio de autorización simulado); administrador de acceso. usuarioTieneAcceso(ID_USUARIO);

// La prueba fallará si accessManager.userHasAccess(USER_ID) no llamó //
mockAuthorizationService.lookupUser(USER_ID),
verificar(Servicio de autorización simulado).buscarUsuario(ID_USUARIO);
```

Al igual que en el stubing, las pruebas de interacción generalmente se realizan a través de marcos de trabajo simulados. Esto reduce el estándar en comparación con la creación manual de nuevas clases que contienen código para realizar un seguimiento de la frecuencia con la que se llama a una función y qué argumentos se pasaron.

Las pruebas de interacción a veces se denominan *burlón*. Evitamos esta terminología en este capítulo porque puede confundirse con marcos de trabajo simulados, que se pueden usar tanto para creación de stub como para pruebas de interacción.

Como se analiza más adelante en este capítulo, las pruebas de interacción son útiles en ciertas situaciones, pero deben evitarse cuando sea posible porque el uso excesivo puede resultar fácilmente en pruebas frágiles.

Implementaciones reales

Aunque los dobles de prueba pueden ser herramientas de prueba invaluables, nuestra primera opción para las pruebas es usar las implementaciones reales del sistema bajo las dependencias de prueba; es decir, las mismas implementaciones que se utilizan en el código de producción. Las pruebas tienen mayor fidelidad cuando

ejecutan el código como se ejecutará en producción, y el uso de implementaciones reales ayuda a lograrlo.

En Google, la preferencia por las implementaciones reales se desarrolló con el tiempo, ya que vimos que el uso excesivo de marcos de trabajo simulados tendía a contaminar las pruebas con código repetitivo que no estaba sincronizado con la implementación real y dificultaba la refactorización. Veremos este tema con más detalle más adelante en este capítulo.

Preferir implementaciones reales en las pruebas se conoce como *prueba clásica*. También hay un estilo de prueba conocido como *prueba falsa*, en el que la preferencia es utilizar marcos de simulación en lugar de implementaciones reales. Aunque algunas personas en la industria del software practican pruebas simuladas (incluida [la creación de los primeros mocking frameworks](#)), en Google, hemos descubierto que este estilo de prueba es difícil de escalar. Requiere que los ingenieros sigan [pautas estrictas al diseñar el sistema bajo prueba](#), y el comportamiento predeterminado de la mayoría de los ingenieros de Google ha sido escribir código de una manera que sea más adecuada para el estilo de prueba clásico.

Prefiere el realismo al aislamiento

El uso de implementaciones reales para dependencias hace que el sistema bajo prueba sea más realista dado que todo el código en estas implementaciones reales se ejecutará en la prueba. Por el contrario, una prueba que utiliza dobles de prueba aísla el sistema bajo prueba de sus dependencias para que la prueba no ejecute código en las dependencias del sistema bajo prueba.

Preferimos las pruebas realistas porque dan más confianza de que el sistema bajo prueba funciona correctamente. Si las pruebas unitarias dependen demasiado de los dobles de prueba, es posible que un ingeniero deba ejecutar pruebas de integración o verificar manualmente que su función funcione como se espera para obtener el mismo nivel de confianza. Llevar a cabo estas tareas adicionales puede ralentizar el desarrollo e incluso puede permitir que se filtren errores si los ingenieros omiten estas tareas por completo cuando requieren demasiado tiempo para llevarlas a cabo en comparación con la ejecución de pruebas unitarias.

Reemplazar todas las dependencias de una clase con dobles de prueba arbitrariamente aísla el sistema bajo prueba a la implementación que el autor coloca directamente en la clase y excluye la implementación que se encuentra en diferentes clases. Sin embargo, una buena prueba debe ser independiente de la implementación: debe escribirse en términos de la API que se está probando en lugar de en términos de cómo está estructurada la implementación.

El uso de implementaciones reales puede hacer que su prueba falle si hay un error en la implementación real. ¡Esto es bueno! Tú *deseas* tus pruebas fallan en tales casos porque indica que su código no funcionará correctamente en producción. A veces, un error en una implementación real puede causar una cascada de fallas en las pruebas porque otras pruebas que usan la implementación real también pueden fallar. Pero con buenas herramientas de desarrollo, como un

Sistema de integración continua (CI), por lo general es fácil rastrear el cambio que causó la falla.

Estudio de caso: @DoNotMock

En Google, hemos visto suficientes pruebas que se basan demasiado en marcos de simulación para motivar la creación de @No se burlenanotación en Java, que está disponible como parte del Propenso a errores herramienta de análisis estático. Esta anotación es una forma para que los propietarios de API declaren, "no se debe burlar de este tipo porque existen mejores alternativas".

Si un ingeniero intenta usar un marco simulado para crear una instancia de una clase o interfaz que se ha anotado como @no se burlen, como se demuestra en [Ejemplo 13-10](#), verán un error que les indicará que utilicen una estrategia de prueba más adecuada, como una implementación real o una falsificación. Esta anotación se usa más comúnmente para objetos de valor que son lo suficientemente simples para usar tal cual, así como para API que tienen falsificaciones bien diseñadas disponibles.

Ejemplo 13-10. La anotación @DoNotMock

```
@DoNotMock("Use SimpleQuery.create() en lugar de burlarse.") clase  
pública abstractaConsulta{  
    resumen públicoCuerdaobtenerValorConsulta();  
}
```

¿Por qué le importaría al propietario de una API? En resumen, restringe severamente la capacidad del propietario de la API para realizar cambios en su implementación a lo largo del tiempo. Como exploraremos más adelante en el capítulo, cada vez que se utiliza un marco de simulación para pruebas de interacción o creación de apéndices, se duplica el comportamiento proporcionado por la API.

Cuando el propietario de la API desea cambiar su API, es posible que descubra que se ha burlado de ella miles o incluso decenas de miles de veces en la base de código de Google. Es muy probable que estos dobles de prueba muestren un comportamiento que viole el contrato de API del tipo que se burla, por ejemplo, devolver un valor nulo para un método que nunca puede devolver un valor nulo. Si las pruebas hubieran utilizado la implementación real o una falsa, el propietario de la API podría realizar cambios en su implementación sin corregir primero miles de pruebas defectuosas.

Cómo decidir cuándo usar una implementación real

Se prefiere una implementación real si es rápida, determinista y tiene dependencias simples. Por ejemplo, se debe usar una implementación real para un *objeto de valor*. Los ejemplos incluyen una cantidad de dinero, una fecha, una dirección geográfica o una clase de colección como una lista o un mapa.

Sin embargo, para un código más complejo, el uso de una implementación real a menudo no es factible. Es posible que no haya una respuesta exacta sobre cuándo usar una implementación real o un doble de prueba, dado que se deben realizar compensaciones, por lo que debe tener en cuenta las siguientes consideraciones.

Tiempo de ejecución

Una de las cualidades más importantes de las pruebas unitarias es que deben ser rápidas: desea poder ejecutarlas continuamente durante el desarrollo para poder obtener comentarios rápidos sobre si su código está funcionando (y también desea que finalicen rápidamente cuando ejecutar en un sistema CI). Como resultado, un doble de prueba puede ser muy útil cuando la implementación real es lenta.

¿Qué tan lento es demasiado lento para una prueba unitaria? Si una implementación real agregara un milisegundo al tiempo de ejecución de cada caso de prueba individual, pocas personas lo clasificarían como lento. Pero, ¿y si añadiera 10 milisegundos, 100 milisegundos, 1 segundo, etc.?

No hay una respuesta exacta aquí; puede depender de si los ingenieros sienten una pérdida de productividad y de cuántas pruebas están utilizando la implementación real (un segundo adicional por caso de prueba puede ser razonable si hay cinco casos de prueba, pero no si hay 500). Para situaciones límite, a menudo es más simple usar una implementación real hasta que se vuelve demasiado lento para usar, momento en el que las pruebas se pueden actualizar para usar una prueba doble en su lugar.

La paralelización de las pruebas también puede ayudar a reducir el tiempo de ejecución. En Google, nuestra infraestructura de prueba hace que sea trivial dividir las pruebas en un conjunto de pruebas para que se ejecuten en varios servidores. Esto aumenta el costo del tiempo de la CPU, pero puede proporcionar un gran ahorro en el tiempo del desarrollador. Discutimos esto más en [capítulo 18](#).

Otra compensación a tener en cuenta: el uso de una implementación real puede resultar en un aumento de los tiempos de compilación dado que las pruebas deben compilar la implementación real, así como todas sus dependencias. Usando un sistema de compilación altamente escalable como [bazel](#) puede ayudar porque almacena en caché los artefactos de compilación sin cambios.

Determinismo

una prueba es *determinista*, para una versión dada del sistema bajo prueba, ejecutar la prueba siempre da como resultado el mismo resultado; es decir, la prueba siempre pasa o siempre falla. Por el contrario, una prueba es *no determinista* si su resultado puede cambiar, incluso si el sistema bajo prueba permanece sin cambios.

No determinismo en las pruebas puede conducir a la descamación: las pruebas pueden fallar ocasionalmente incluso cuando no hay cambios en el sistema bajo prueba. Como se discutió en [Capítulo 11](#), la descamación daña la salud de un conjunto de pruebas si los desarrolladores comienzan a desconfiar de los resultados de la prueba e ignoran las fallas. Si el uso de una implementación real rara vez causa problemas, es posible que no justifique una respuesta, ya que hay poca interrupción para los ingenieros. Pero si ocurre descamación

bolígrafos a menudo, podría ser el momento de reemplazar una implementación real con un doble de prueba porque al hacerlo mejorará la fidelidad de la prueba.

Una implementación real puede ser mucho más compleja en comparación con un doble de prueba, lo que aumenta la probabilidad de que sea no determinista. Por ejemplo, una implementación real que utiliza subprocessos múltiples puede ocasionalmente causar que una prueba falle si la salida del sistema bajo prueba difiere según el orden en que se ejecutan los subprocessos.

Una causa común de no determinismo es el código que no es **hermético**; es decir, tiene dependencias de servicios externos que están fuera del control de una prueba. Por ejemplo, una prueba que intente leer el contenido de una página web desde un servidor HTTP podría fallar si el servidor está sobrecargado o si cambia el contenido de la página web. En su lugar, se debe utilizar un doble de prueba para evitar que la prueba dependa de un servidor externo. Si no es factible usar un doble de prueba, otra opción es usar una instancia hermética de un servidor, cuyo ciclo de vida está controlado por la prueba. Las instancias herméticas se analizan con más detalle en el próximo capítulo.

Otro ejemplo de no determinismo es el código que se basa en el reloj del sistema dado que la salida del sistema bajo prueba puede diferir según la hora actual. En lugar de confiar en el reloj del sistema, una prueba puede usar un doble de prueba que codifica una hora específica.

Construcción de dependencia

Al usar una implementación real, debe construir todas sus dependencias. Por ejemplo, un objeto necesita que se construya todo su árbol de dependencias: todos los objetos de los que depende, todos los objetos de los que dependen estos objetos dependientes, etc. Un doble de prueba a menudo no tiene dependencias, por lo que construir un doble de prueba puede ser mucho más simple en comparación con construir una implementación real.

Como ejemplo extremo, imagine intentar crear el objeto en el fragmento de código que sigue en una prueba. Llevaría mucho tiempo determinar cómo construir cada objeto individual. Las pruebas también requerirán un mantenimiento constante porque deben actualizarse cuando se modifica la firma de los constructores de estos objetos:

```
Foo foo = nuevo Foo(nuevo A(nuevo B(nuevo C()), nuevo D()), nuevo E(), ..., nuevo Z());
```

Puede ser tentador usar un doble de prueba porque construir uno puede ser trivial. Por ejemplo, esto es todo lo que se necesita para construir un doble de prueba cuando se usa el marco de simulación de Mockito:

```
@Mock Foo mockFoo;
```

Aunque la creación de este doble de prueba es mucho más simple, existen beneficios significativos al usar la implementación real, como se discutió anteriormente en esta sección. A menudo, también hay desventajas significativas en el uso excesivo de dobles de prueba de esta manera, que veremos más adelante en este capítulo. Por lo tanto, se debe hacer una compensación al considerar si usar una implementación real o un doble de prueba.

En lugar de construir manualmente el objeto en las pruebas, la solución ideal es usar el mismo código de construcción de objetos que se usa en el código de producción, como un método de fábrica o una inyección de dependencia automatizada. Para respaldar el caso de uso de las pruebas, el código de construcción del objeto debe ser lo suficientemente flexible para poder usar dobles de prueba en lugar de codificar las implementaciones que se usarán para la producción.

fingiendo

Si usar una implementación real no es factible dentro de una prueba, la mejor opción suele ser usar una falsa en su lugar. Se prefiere una falsificación sobre otras técnicas dobles de prueba porque se comporta de manera similar a la implementación real: el sistema bajo prueba ni siquiera debería saber si está interactuando con una implementación real o una falsificación.

Ejemplo 13-11 ilustra un sistema de archivos falso.

Ejemplo 13-11. Un sistema de archivos falso

```
// Esta falsificación implementa la interfaz FileSystem. Esta interfaz también es //  
utilizada por la implementación real.  
clase públicaSistema de archivos falsosimplementossistema de archivos{  
    // Almacena un mapa del nombre del archivo al contenido del archivo. Los archivos se almacenan en // la  
    memoria en lugar de en el disco, ya que las pruebas no deberían necesitar realizar operaciones de E/S en el  
    disco. privadoMapa<Cuerda,Cuerda>archivos=nuevoMapa hash<>(); @Anular  
  
    públicoCavocaciónescribir archivo(Cadena de nombre de archivo,Contenido de la cadena) {  
        // Agregue el nombre del archivo y el contenido al mapa. archivos.  
        agregar(nombre del archivo,contenido);  
    }  
    @Anular  
    públicoCuerdaLeer archivo(Cadena de nombre de archivo) {  
        Contenido de la cadena=archivos.conseguir(nombre del archivo);  
        // La implementación real lanzará esta excepción si no se encuentra // el  
        // archivo, por lo que la implementación falsa también debe lanzarla.  
        si(contenido==nulo) {tirar nuevoExcepción de archivo no encontrado(nombre del archivo); }  
        retornocontenido;  
    }  
}
```

¿Por qué son importantes las falsificaciones?

Las falsificaciones pueden ser una herramienta poderosa para las pruebas: se ejecutan rápidamente y le permiten probar su código de manera efectiva sin los inconvenientes de usar implementaciones reales.

Una sola falsificación tiene el poder de mejorar radicalmente la experiencia de prueba de una API. Si escala eso a una gran cantidad de falsificaciones para todo tipo de API, las falsificaciones pueden proporcionar un enorme impulso a la velocidad de ingeniería en una organización de software.

En el otro extremo del espectro, en una organización de software donde las falsificaciones son raras, la velocidad será más lenta porque los ingenieros pueden terminar luchando con el uso de implementaciones reales que conducen a pruebas lentas y escamosas. O bien, los ingenieros pueden recurrir a otras técnicas de pruebas dobles, como pruebas de interacción o de creación de apéndices, que, como veremos más adelante en este capítulo, pueden dar lugar a pruebas poco claras, frágiles y menos eficaces.

¿Cuándo se deben escribir las falsificaciones?

Una falsificación requiere más esfuerzo y más experiencia en el dominio para crear porque debe comportarse de manera similar a la implementación real. Una falsificación también requiere mantenimiento: cada vez que cambia el comportamiento de la implementación real, la falsificación también debe actualizarse para que coincida con este comportamiento. Debido a esto, el equipo propietario de la implementación real debe escribir y mantener una versión falsa.

Si un equipo está considerando escribir una falsificación, se debe hacer una compensación sobre si las mejoras de productividad que resultarán del uso de la falsificación superan los costos de escribirla y mantenerla. Si solo hay un puñado de usuarios, es posible que no valga la pena su tiempo, mientras que si hay cientos de usuarios, puede resultar en una mejora obvia de la productividad.

Para reducir la cantidad de falsificaciones que deben mantenerse, normalmente se debe crear una falsificación solo en la raíz del código que no es factible para su uso en las pruebas. Por ejemplo, si una base de datos no se puede usar en las pruebas, debe existir una falsificación para la propia API de la base de datos en lugar de para cada clase que llama a la API de la base de datos.

Mantener una falsificación puede ser una carga si su implementación debe duplicarse en los lenguajes de programación, como para un servicio que tiene bibliotecas de clientes que permiten invocar el servicio desde diferentes lenguajes. Una solución para este caso es crear una única implementación de servicio falso y hacer que las pruebas configuren las bibliotecas del cliente para enviar solicitudes a este servicio falso. Este enfoque es más pesado en comparación con tener el falso escrito completamente en la memoria porque requiere que la prueba se comunique entre procesos. Sin embargo, puede ser una compensación razonable, siempre que las pruebas aún puedan ejecutarse rápidamente.

La fidelidad de las falsificaciones

Quizás el concepto más importante que rodea la creación de falsificaciones es *fidelidad*; en otras palabras, en qué medida el comportamiento de una falsificación coincide con el comportamiento de la implementación real. Si el comportamiento de una falsificación no coincide con el comportamiento de la implementación real, una prueba que use esa falsificación no es útil; una prueba podría pasar cuando se usa la falsificación, pero esta misma ruta de código podría no funcionar correctamente en la implementación real.

La fidelidad perfecta no siempre es factible. Después de todo, la falsificación era necesaria porque la implementación real no era adecuada de una forma u otra. Por ejemplo, una base de datos falsa normalmente no sería fiel a una base de datos real en términos de almacenamiento en el disco duro porque la falsa almacenaría todo en la memoria.

Principalmente, sin embargo, una falsificación debe mantener la fidelidad a los contratos de API de la implementación real. Para cualquier entrada dada a una API, una falsificación debe devolver la misma salida y realizar los mismos cambios de estado de su implementación real correspondiente. Por ejemplo, para una implementación real debase de datos.guardar(itemId), si un elemento se guarda con éxito cuando aún no existe su ID pero se produce un error cuando ya existe el ID, la falsificación debe ajustarse a este mismo comportamiento.

Una forma de pensar en esto es que la falsificación debe tener perfecta fidelidad a la implementación real, pero *sólo desde la perspectiva de la prueba*. Por ejemplo, una falsificación de una API hash no necesita garantizar que el valor hash para una entrada determinada sea exactamente el mismo que el valor hash generado por la implementación real; es probable que a las pruebas no les importe el valor hash específico., solo que el valor hash es único para una entrada dada. Si el contrato de la API hash no garantiza qué valores de hash específicos se devolverán, la falsificación aún se ajusta al contrato, incluso si no tiene una fidelidad perfecta a la implementación real.

Otros ejemplos en los que la fidelidad perfecta normalmente podría no ser útil para las falsificaciones incluyen la latencia y el consumo de recursos. Sin embargo, no se puede usar una falsificación si necesita probar explícitamente estas restricciones (por ejemplo, una prueba de rendimiento que verifica la latencia de una llamada de función), por lo que deberá recurrir a otros mecanismos, como usar una implementación real en su lugar. de un falso

Es posible que una falsificación no necesite tener el 100% de la funcionalidad de su implementación real correspondiente, especialmente si la mayoría de las pruebas no necesitan dicho comportamiento (por ejemplo, código de manejo de errores para casos extremos raros). En este caso, lo mejor es que la falsificación falle rápidamente; por ejemplo, generar un error si se ejecuta una ruta de código no admitida. Esta falla le comunica al ingeniero que la falsificación no es apropiada en esta situación.

Las falsificaciones deben ser probadas

Una falsificación debe tener *propriopruuebas* para asegurar que se ajusta a la API de su correspondiente implementación real. Una falsificación sin pruebas inicialmente podría proporcionar un comportamiento realista, pero sin pruebas, este comportamiento puede divergir con el tiempo a medida que evoluciona la implementación real.

Un enfoque para escribir pruebas para falsificaciones implica escribir pruebas en la interfaz pública de la API y ejecutar esas pruebas tanto en la implementación real como en la falsa (esto se conoce como *pruebas de contrato*). Las pruebas que se ejecutan contra la implementación real probablemente serán más lentas, pero su desventaja se minimiza porque solo los propietarios de la falsificación deben ejecutarlas.

Qué hacer si una falsificación no está disponible

Si una falsificación no está disponible, primero solicite a los propietarios de la API que creen una. Es posible que los propietarios no estén familiarizados con el concepto de falsificación o que no se den cuenta del beneficio que brindan a los usuarios de una API.

Si los propietarios de una API no quieren o no pueden crear una falsificación, es posible que pueda escribir la suya propia. Una forma de hacer esto es envolver todas las llamadas a la API en una sola clase y luego crear una versión falsa de la clase que no se comunique con la API. Hacer esto también puede ser mucho más simple que crear una falsificación para toda la API porque, de todos modos, a menudo necesitará usar solo un subconjunto del comportamiento de la API. En Google, algunos equipos incluso han contribuido con su falsificación a los propietarios de la API, lo que ha permitido que otros equipos se beneficien de la falsificación.

Por último, podría decidir utilizar una implementación real (y lidiar con las ventajas y desventajas de las implementaciones reales que se mencionaron anteriormente en este capítulo), o recurrir a otras técnicas de pruebas dobles (y tratar con las ventajas y desventajas que mencionaremos). mencionar más adelante en este capítulo).

En algunos casos, puede pensar en una falsificación como una optimización: si las pruebas son demasiado lentas con una implementación real, puede crear una falsificación para que se ejecuten más rápido. Pero si la aceleración de una falsificación no supera el trabajo que se necesitaría para crear y mantener la falsificación, sería mejor seguir usando la implementación real.

talonar

Como se discutió anteriormente en este capítulo, el stubing es una forma de probar el comportamiento del código para una función que de otro modo no tiene comportamiento por sí misma. Suele ser una forma rápida y fácil de reemplazar una implementación real en una prueba. Por ejemplo, el código en [Ejemplo 13-12](#) utiliza stubing para simular la respuesta de un servidor de tarjetas de crédito.

Ejemplo 13-12. Usar stubbing para simular respuestas

```
@Prueba@pùblico@vacío@getTransactionCount() {  
    contador de transacciones=nuevoContador de transacciones(servidor de tarjeta de crédito  
simulado); // Usar stubing para devolver tres transacciones.  
    cuándo(servidor de tarjeta de crédito simulado).obtenerTransacciones().luegoRegresar(  
        lista nueva(TRANSACCIÓN_1,TRANSACCIÓN_2,TRANSACCIÓN_3)); afirmar  
        que(contador de transacciones.getTransactionCount()).es igual a(3);  
}
```

Los peligros del uso excesivo de stubbing

Debido a que el stubing es tan fácil de aplicar en las pruebas, puede ser tentador usar esta técnica siempre que no sea trivial usar una implementación real. Sin embargo, el uso excesivo de stubing puede resultar en grandes pérdidas de productividad para los ingenieros que necesitan mantener estas pruebas.

Las pruebas se vuelven poco claras

El stubing implica escribir código adicional para definir el comportamiento de las funciones que se están stubing. Tener este código adicional resta valor a la intención de la prueba, y este código puede ser difícil de entender si no está familiarizado con la implementación del sistema bajo prueba.

Una señal clave de que el stubing no es apropiado para una prueba es si se encuentra recorriendo mentalmente el sistema bajo prueba para comprender por qué ciertas funciones en la prueba están stubing.

Las pruebas se vuelven frágiles

El stubing filtra los detalles de implementación de su código en su prueba. Cuando cambien los detalles de implementación en su código de producción, deberá actualizar sus pruebas para reflejar estos cambios. Idealmente, una buena prueba debería cambiar solo si cambia el comportamiento de cara al usuario de una API; no debería verse afectado por los cambios en la implementación de la API.

Las pruebas se vuelven menos efectivas

Con el stubing, no hay forma de garantizar que la función que se está stubing se comporte como la implementación real, como en una declaración como la que se muestra en el siguiente fragmento que codifica parte del contrato de `agregar()` método (*“Si se pasa 1 y 2, se devuelve 3”*):

```
when(stubCalculator.add(1, 2)).thenReturn(3);
```

El stubing es una mala elección si el sistema bajo prueba depende del contrato de implementación real porque se verá obligado a duplicar los detalles del contrato y no hay forma de garantizar que el contrato sea correcto (es decir, que la función stub tiene fidelidad a la implementación real).

Además, con el stubing no hay forma de almacenar el estado, lo que puede dificultar la prueba de ciertos aspectos de su código. Por ejemplo, si llamanbase de datos.guardar(elemento)ya sea en una implementación real o falsa, es posible que pueda recuperar el elemento llamandobase de datos.get(elemento.id())dado que ambas llamadas acceden al estado interno, pero con stubing, no hay forma de hacerlo.

Un ejemplo de uso excesivo de stubing

Ejemplo 13-13ilustra una prueba que abusa de stubbing.

Ejemplo 13-13. Uso excesivo de stubing

```
@Prueba@pùblicovacióla tarjeta de crédito está cargada() {  
    // Pasar los dobles de prueba que fueron creados por un marco de simulación.  
    procesador de pagos=  
        nuevoProcesador de pagos(servidor de tarjeta de crédito simulado,Procesador de transacciones simuladas);  
        // Configure el stubing para estos dobles de prueba.  
        cuándo(servidor de tarjeta de crédito simulado.esServidorDisponible()).luegoRegresar(verdadero);  
        cuándo(Procesador de transacciones simuladas.comenzarTransacción()).luegoRegresar(transacción); cuándo(  
            servidor de tarjeta de crédito simulado.initTransaction(transacción)).luegoRegresar(verdadero); cuándo(  
            servidor de tarjeta de crédito simulado.pagar(transacción,tarjeta de crédito,500))  
                .luegoRegresar(falso);  
        cuándo(Procesador de transacciones simuladas.endTransaction()).luegoRegresar(  
            verdadero); // Llamar al sistema bajo prueba.  
        procesador de pagos.Procesar pago(tarjeta de crédito,Dinero.dolares(500)); //  
        // No hay forma de saber si el método pay() realmente llevó a cabo la //  
        // transacción, por lo que lo único que puede hacer la prueba es verificar que se  
        // llamó al método // pay().  
        verificar(servidor de tarjeta de crédito simulado).pagar(transacción,tarjeta de crédito,500);  
}
```

Ejemplo 13-14reescribe la misma prueba pero evita usar stubbing. Observe cómo la prueba es más corta y que los detalles de implementación (como la forma en que se usa el procesador de transacciones) no se exponen en la prueba. No se necesita una configuración especial porque el servidor de la tarjeta de crédito sabe cómo comportarse.

Ejemplo 13-14. Refactorización de una prueba para evitar stubbing

```
@Prueba@pùblicovacióla tarjeta de crédito está cargada() {  
    procesador de pagos=  
        nuevoProcesador de pagos(servidor de tarjetas de crédito,procesador de transacciones  
    ); // Llamar al sistema bajo prueba.  
    procesador de pagos.Procesar pago(tarjeta de crédito,Dinero.dolares(500));  
    // Consulta el estado del servidor de la tarjeta de crédito para ver si se realizó el pago. afirmar que(  
    servidor de tarjetas de crédito.obtener la carga más reciente(tarjeta de crédito))  
        .es igual a(500);  
}
```

Obviamente, no queremos que una prueba de este tipo se comunique con un servidor de tarjeta de crédito externo, por lo que un servidor de tarjeta de crédito falso sería más adecuado. Si no se dispone de un fake, otra opción es utilizar una implementación real que hable con un servidor hermético de tarjetas de crédito, aunque esto aumentará el tiempo de ejecución de las pruebas. (Exploramos los servidores herméticos en el próximo capítulo).

¿Cuándo es apropiado el stubbing?

En lugar de un reemplazo general para una implementación real, el stubbing es apropiado cuando necesita una función para devolver un valor específico para poner el sistema bajo prueba en un estado determinado, como [Ejemplo 13-12](#) que requiere que el sistema bajo prueba devuelva una lista de transacciones no vacía. Debido a que el comportamiento de una función se define en línea en la prueba, el stubbing puede simular una amplia variedad de valores devueltos o errores que podrían no ser posibles de desencadenar desde una implementación real o una falsificación.

Para garantizar que su propósito sea claro, cada función auxiliar debe tener una relación directa con las afirmaciones de la prueba. Como resultado, una prueba normalmente debe eliminar una pequeña cantidad de funciones porque eliminar muchas funciones puede llevar a que las pruebas sean menos claras. Una prueba que requiere que se apliquen stubs a muchas funciones puede ser una señal de que el stubbing se está utilizando en exceso, o que el sistema que se está probando es demasiado complejo y debe refactorizarse.

Tenga en cuenta que incluso cuando la creación de apéndices es adecuada, se prefieren las implementaciones reales o las falsificaciones porque no exponen los detalles de la implementación y le brindan más garantías sobre la corrección del código en comparación con la creación de apéndices. Pero el stubbing puede ser una técnica razonable de usar, siempre que su uso esté restringido para que las pruebas no se vuelvan demasiado complejas.

Pruebas de interacción

Como se discutió anteriormente en este capítulo, la prueba de interacción es una forma de validar cómo se llama a una función sin realmente llamar a la implementación de la función.

Los marcos de simulación facilitan la realización de pruebas de interacción. Sin embargo, para mantener las pruebas útiles, legibles y resistentes al cambio, es importante realizar pruebas de interacción solo cuando sea necesario.

Preferir las pruebas estatales sobre las pruebas de interacción

A diferencia de las pruebas de interacción, se prefiere probar el código a través de [prueba estatal](#).

Con la prueba de estado, llama al sistema bajo prueba y valida que se devolvió el valor correcto o que algún otro estado en el sistema bajo prueba se cambió correctamente. [Ejemplo 13-15](#) presenta un ejemplo de prueba estatal.

Ejemplo 13-15. Pruebas estatales

```
@Prueba@pùblicovacióordenarNÚmeros() {  
    clasificador de númerosclásificadord de nùmeros=nuevoclásificadord de nùmerosordenaciónd rápida.ordenamiento de  
    burbuja); // Llamar al sistema bajo prueba.  
    Listalista ordenada=clásificadord de nùmeros.ordenarNÚmeros(lista nueva(3,1,2));  
    // Validar que la lista devuelta esté ordenada. No importa qué // algoritmo de  
    // clasificación se use, siempre que se devuelva el resultado correcto. afirmar que(   
    lista ordenada).es igual a(lista nueva(1,2,3));  
}
```

Ejemplo 13-16 ilustra un escenario de prueba similar pero en su lugar utiliza pruebas de interacción. Tenga en cuenta que es imposible que esta prueba determine que los números están realmente ordenados, porque los dobles de prueba no saben cómo ordenar los números; todo lo que puede decirle es que el sistema bajo prueba intentó ordenar los números.

Ejemplo 13-16. Pruebas de interacción

```
@Prueba@pùblicovaciósorNumbers_quicksortIsUsed() {  
    // Pasar los dobles de prueba que fueron creados por un marco de simulación. clásificadord de  
    // nùmerosclásificadord de nùmeros=  
    nuevoclásificadord de nùmeros(simulacroQuicksort,simulacroBubbleSort);  
  
    // Llamar al sistema bajo prueba. clásificadord de nùmeros.  
    ordenarNÚmeros(lista nueva(3,1,2));  
  
    // Validar que numberSorter.sortNumbers() usó ordenación rápida. La  
    // prueba // fallará si nunca se llama a mockQuicksort.sort() (p. ej., si  
    // se usa mockBubbleSort) o si se llama con los argumentos incorrectos. verificar(   
    simulacroQuicksort).clásificard(lista nueva(3,1,2));  
}
```

En Google, descubrimos que enfatizar las pruebas estatales es más escalable; reduce la fragilidad de las pruebas, lo que facilita cambiar y mantener el código a lo largo del tiempo.

El problema principal con las pruebas de interacción es que no puede decirle que el sistema bajo prueba está funcionando correctamente; solo puede validar que ciertas funciones se llamen como se esperaba. Requiere que haga una suposición sobre el comportamiento del código; por ejemplo, "*Si base de datos.guardar(elemento) se llama, suponemos que el elemento se guardará en la base de datos.* Se prefiere la prueba de estado porque en realidad valida esta suposición (como guardar un elemento en una base de datos y luego consultar la base de datos para validar que el elemento existe).

Otra desventaja de las pruebas de interacción es que utiliza detalles de implementación del sistema bajo prueba: para validar que se llamó a una función, está exponiendo a la prueba que el sistema bajo prueba llama a esta función. Al igual que el stubbing, este código adicional hace que las pruebas sean frágiles porque filtra los detalles de implementación de su código de producción en las pruebas.

Algunas personas en Google se refieren en broma a las pruebas que abusan de la interacción.

probando como *pruebas de detectores de cambios* porque fallan en respuesta a cualquier cambio en el código de producción, incluso si el comportamiento del sistema bajo prueba permanece sin cambios.

¿Cuándo son apropiadas las pruebas de interacción?

Hay algunos casos en los que se justifican las pruebas de interacción:

- No puede realizar pruebas de estado porque no puede usar una implementación real o una falsificación (por ejemplo, si la implementación real es demasiado lenta y no existe una falsificación). Como alternativa, puede realizar pruebas de interacción para validar que se llame a determinadas funciones. Aunque no es lo ideal, esto proporciona un nivel básico de confianza de que el sistema bajo prueba funciona como se esperaba.
- Las diferencias en el número o el orden de las llamadas a una función provocarían un comportamiento no deseado. Las pruebas de interacción son útiles porque podría ser difícil validar este comportamiento con las pruebas de estado. Por ejemplo, si espera que una función de almacenamiento en caché reduzca la cantidad de llamadas a una base de datos, puede verificar que no se acceda al objeto de la base de datos más veces de lo esperado. Usando Mockito, el código podría verse similar a esto:

```
verificar(databaseReader, atMostOnce()).selectRecords();
```

Las pruebas de interacción no son un reemplazo completo de las pruebas estatales. Si no puede realizar pruebas de estado en una prueba unitaria, considere seriamente complementar su conjunto de pruebas con pruebas de mayor alcance que sí realicen pruebas de estado. Por ejemplo, si tiene una prueba de unidad que valida el uso de una base de datos a través de pruebas de interacción, considere agregar una prueba de integración que pueda realizar pruebas de estado en una base de datos real. Las pruebas de mayor alcance son una estrategia importante para la mitigación de riesgos y las analizamos en el siguiente capítulo.

Mejores prácticas para pruebas de interacción

Al realizar pruebas de interacción, seguir estas prácticas puede reducir parte del impacto de las desventajas antes mencionadas.

Prefiere realizar pruebas de interacción solo para funciones de cambio de estado

Cuando un sistema bajo prueba llama a una función en una dependencia, esa llamada cae en una de dos categorías:

cambio de estado

Funciones que tienen efectos secundarios en el mundo fuera del sistema bajo prueba.

Ejemplos: enviarEmail(), guardarRegistro(), acceder al registro().

No cambia de estado

Funciones que no tienen efectos secundarios; devuelven información sobre el mundo fuera del sistema bajo prueba y no modifican nada. Ejemplos:obtenerUsuario(), buscarResultados(), leerArchivo().

En general, debe realizar pruebas de interacción solo para funciones que cambian de estado. Realizar pruebas de interacción para funciones que no cambian de estado suele ser redundante dado que el sistema bajo prueba usará el valor de retorno de la función para hacer otro trabajo que usted pueda afirmar. La interacción en sí no es un detalle importante para la corrección, porque no tiene efectos secundarios.

Realizar pruebas de interacción para funciones que no cambian de estado hace que su prueba sea frágil porque necesitará actualizar la prueba cada vez que cambie el patrón de interacciones. También hace que la prueba sea menos legible dado que las afirmaciones adicionales hacen que sea más difícil determinar qué afirmaciones son importantes para garantizar la corrección del código. Por el contrario, las interacciones de cambio de estado representan algo útil que su código está haciendo para cambiar el estado en otro lugar.

Ejemplo 13-17 demuestra pruebas de interacción en funciones que cambian de estado y que no cambian de estado.

Ejemplo 13-17. Interacciones de cambio de estado y de no cambio de estado

```
@Prueba    Usuario Autorizador usuario Autorizador=  
        nuevoAutorizador de usuario(servicio de usuario simulado,base de datos de permisos simulados); cuándo(  
            simulacroPermisoServicio.obtener permiso(USUARIO_FALSO)).luegoRegresar(VACÍO);  
  
    // Llamar al sistema bajo prueba. usuario Autorizador.  
    conceder permiso(ACCESO DE USUARIO);  
  
    // addPermission() cambia de estado, por lo que es razonable realizar //  
    pruebas de interacción para validar que se llamó.  
    verificar(base de datos de permisos simulados).agregarPermiso(USUARIO_FALSO,ACCESO DE USUARIO);  
  
    // getPermission() no cambia de estado, por lo que esta línea de código no es //  
    necesaria. Una pista de que es posible que no se necesiten pruebas de  
    interacción: // getPermission() ya se aplicó anteriormente en esta prueba.  
    verificar(base de datos de permisos simulados).obtener permiso(USUARIO_FALSO);  
}
```

Evite la sobreespecificación

En [Capítulo 12](#), discutimos por qué es útil probar comportamientos en lugar de métodos. Esto significa que un método de prueba debe centrarse en verificar un comportamiento de un método o clase en lugar de tratar de verificar múltiples comportamientos en una sola prueba.

Al realizar pruebas de interacción, debemos intentar aplicar el mismo principio evitando especificar en exceso qué funciones y argumentos se validan. Esto conduce a pruebas que son más claras y concisas. También conduce a pruebas que son resistentes a los cambios realizados en los comportamientos que están fuera del alcance de cada prueba, por lo que fallarán menos pruebas si se realiza un cambio en la forma en que se llama a una función.

Ejemplo 13-18 ilustra las pruebas de interacción con sobreespecificación. La intención de la prueba es validar que el nombre del usuario esté incluido en el aviso de saludo, pero la prueba fallará si se cambia un comportamiento no relacionado.

Ejemplo 13-18. Pruebas de interacción sobreespecificadas

```
@Prueba    cuándo(servicio de usuario simulado.obtener nombre de usuario()).luegoRegresar("Usuario  
falso"); saludador de usuario.mostrarsaludo();//Llame al sistema bajo prueba.  
  
    // La prueba fallará si se cambia cualquiera de los argumentos de setText(). verificar(  
    solicitud de usuario).establecerTexto("Usuario falso","¡Buenos días!","Versión 2.1");  
  
    // La prueba fallará si no se llama a setIcon(), aunque este //  
    comportamiento es incidental a la prueba ya que no está relacionado  
    con // validar el nombre de usuario.  
    verificar(solicitud de usuario).conjuntoIcono(IMAGE_SUNSHINE);  
}
```

Ejemplo 13-19 ilustra las pruebas de interacción con más cuidado al especificar argumentos y funciones relevantes. Los comportamientos que se prueban se dividen en pruebas separadas, y cada prueba valida la cantidad mínima necesaria para garantizar que el comportamiento que está probando sea correcto.

Ejemplo 13-19. Pruebas de interacción bien especificadas

```
@Prueba    cuándo(servicio de usuario simulado.obtener nombre de usuario()).luegoRegresar("Usuario  
falso"); saludador de usuario.mostrarsaludo();//Llame al sistema bajo prueba. verificar(solicitud de  
usuario).establecerTexto(alguna(),equivalente("¡Buenos días!"),alguno()); verificar(solicitud de  
usuario).conjuntoIcono(IMAGE_SUNSHINE);  
}
```

Conclusión

Hemos aprendido que los dobles de prueba son cruciales para la velocidad de ingeniería porque pueden ayudar a probar su código de manera integral y garantizar que sus pruebas se ejecuten rápidamente. Por otro lado, su uso indebido puede suponer una gran pérdida de productividad porque pueden dar lugar a pruebas poco claras, frágiles y menos eficaces. Esta es la razón por la que es importante que los ingenieros comprendan las mejores prácticas sobre cómo aplicar de manera efectiva los dobles de prueba.

A menudo no hay una respuesta exacta sobre si usar una implementación real o un doble de prueba, o qué técnica de doble de prueba usar. Es posible que un ingeniero deba hacer algunas concesiones al decidir el enfoque adecuado para su caso de uso.

Aunque los dobles de prueba son excelentes para trabajar con dependencias que son difíciles de usar en las pruebas, si desea maximizar la confianza en su código, en algún momento aún querrá ejercer estas dependencias en las pruebas. El siguiente capítulo cubrirá las pruebas de mayor alcance, para las cuales se utilizan estas dependencias independientemente de su idoneidad para las pruebas unitarias; por ejemplo, incluso si son lentos o no deterministas.

TL; DR

- Se debe preferir una implementación real a un doble de prueba.
- Una falsificación suele ser la solución ideal si no se puede utilizar una implementación real en una prueba.
- El uso excesivo de stubing conduce a pruebas poco claras y quebradizas.
- Las pruebas de interacción deben evitarse cuando sea posible: conducen a pruebas que son frágiles porque exponen los detalles de implementación del sistema bajo prueba.

Pruebas más grandes

*Escrito por Joseph Graves
Editado por Tom Mansreck*

En capítulos anteriores, hemos contado cómo se estableció una cultura de prueba en Google y cómo las pruebas de unidades pequeñas se convirtieron en una parte fundamental del flujo de trabajo del desarrollador. Pero, ¿qué pasa con otros tipos de pruebas? Resulta que, de hecho, Google usa muchas pruebas más grandes, y estas comprenden una parte importante de la estrategia de mitigación de riesgos necesaria para una ingeniería de software saludable. Pero estas pruebas presentan desafíos adicionales para garantizar que sean activos valiosos y no sumideros de recursos. En este capítulo, discutiremos lo que queremos decir con "pruebas más grandes", cuándo las ejecutamos y las mejores prácticas para mantenerlas efectivas.

¿Qué son las pruebas más grandes?

Como se mencionó anteriormente, Google tiene nociones específicas del tamaño de la prueba. Las pruebas pequeñas están restringidas a un hilo, un proceso, una máquina. Las pruebas más grandes no tienen las mismas restricciones. Pero Google también tiene nociones de prueba *al alcance*. Una prueba unitaria necesariamente tiene un alcance menor que una prueba de integración. Y las pruebas de mayor alcance (a veces denominadas pruebas de sistema o de extremo a extremo) suelen implicar varias dependencias reales y menos pruebas dobles.

Las pruebas más grandes son muchas cosas que las pruebas pequeñas no son. No están sujetos a las mismas restricciones; por lo tanto, pueden exhibir las siguientes características:

- Pueden ser lentos. Nuestras pruebas grandes tienen un tiempo de espera predeterminado de 15 minutos o 1 hora, pero también tenemos pruebas que se ejecutan durante varias horas o incluso días.
- Pueden ser no herméticos. Las pruebas grandes pueden compartir recursos con otras pruebas y tráfico.

- Pueden ser no deterministas. Si una prueba grande no es hermética, es casi imposible garantizar el determinismo: otras pruebas o el estado del usuario pueden interferir con ella.

Entonces, ¿por qué tener pruebas más grandes? Reflexione sobre su proceso de codificación. ¿Cómo confirmas que los programas que escribes realmente funcionan? Es posible que esté escribiendo y ejecutando pruebas unitarias sobre la marcha, pero ¿se encuentra ejecutando el binario real y probándolo usted mismo? Y cuando comparte este código con otros, ¿cómo lo prueban? ¿Ejecutando sus pruebas unitarias o probándolas ellos mismos?

Además, ¿cómo sabe que su código continúa funcionando durante las actualizaciones? Suponga que tiene un sitio que utiliza la API de Google Maps y hay una nueva versión de la API. Es probable que sus pruebas unitarias no lo ayuden a saber si hay algún problema de compatibilidad. Probablemente lo ejecutaría y lo probaría para ver si algo se rompió.

Las pruebas unitarias pueden brindarle confianza sobre funciones, objetos y módulos individuales, pero las pruebas grandes brindan más confianza de que el sistema general funciona según lo previsto. Y tener escalas de pruebas automatizadas reales de formas que las pruebas manuales no lo hacen.

Fidelidad

La razón principal por la que existen pruebas más grandes es para abordar *fidelidad*. La fidelidad es la propiedad por la cual una prueba refleja el comportamiento real del sistema bajo prueba (SUT).

Una forma de visualizar la fidelidad es en términos del entorno. Como Figura 14-1 ilustra, las pruebas unitarias agrupan una prueba y una pequeña porción de código como una unidad ejecutable, lo que garantiza que el código se pruebe, pero es muy diferente de cómo se ejecuta el código de producción. La propia producción es, naturalmente, el entorno de mayor fidelidad en las pruebas. También hay un espectro de opciones provisionales. Una clave para las pruebas más grandes es encontrar el ajuste adecuado, porque el aumento de la fidelidad también conlleva un aumento de los costos y (en el caso de la producción) un mayor riesgo de falla.

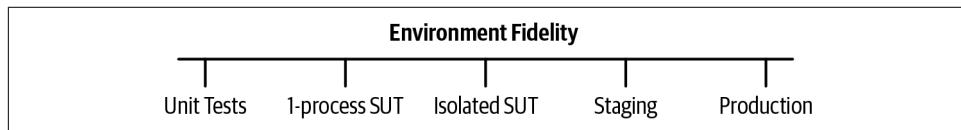


Figura 14-1. Escala de fidelidad creciente

Las pruebas también se pueden medir en términos de cuán fiel es el contenido de la prueba a la realidad. Los ingenieros descartan muchas pruebas grandes y hechas a mano si los datos de la prueba en sí parecen poco realistas. Los datos de prueba copiados de producción son mucho más fieles a la realidad (al haber sido capturados de esa manera), pero un gran desafío es cómo crear un tráfico de prueba realista. *antes de* lanzando el nuevo código. Este es particularmente un problema en la inteligencia artificial (IA), para la cual los datos de "semilla" a menudo sufren de sesgos intrínsecos. Y, debido a que la mayoría de los datos para las pruebas unitarias están elaborados a mano, cubre una gama limitada de casos y tiende a ajustarse a

los prejuicios del autor. Los escenarios descubiertos perdidos por los datos representan una brecha de fidelidad en las pruebas.

Brechas comunes en las pruebas unitarias

También pueden ser necesarias pruebas más grandes donde fallan las pruebas más pequeñas. Las subsecciones que siguen presentan algunas áreas particulares donde las pruebas unitarias no brindan una buena cobertura de mitigación de riesgos.

dobles infieles

Una prueba de una sola unidad generalmente cubre una clase o módulo. Dobles de prueba (como se discutió en [Capítulo 13](#)) se utilizan con frecuencia para eliminar dependencias pesadas o difíciles de probar. Pero cuando se reemplazan esas dependencias, es posible que el reemplazo y la cosa duplicada no concuerden.

Casi todas las pruebas unitarias en Google están escritas por el mismo ingeniero que está escribiendo la unidad bajo prueba. Cuando esas pruebas unitarias necesitan dobles y cuando los dobles utilizados son simulacros, es el ingeniero quien escribe la prueba unitaria que define el simulacro y su comportamiento previsto. Pero ese ingeniero usualmente lo hacía.¹ noescribir la cosa de la que se burlan y puede estar mal informado sobre su comportamiento real. La relación entre la unidad bajo prueba y un par determinado es un contrato de comportamiento, y si el ingeniero se equivoca sobre el comportamiento real, la comprensión del contrato no es válida.

Además, las burlas se vuelven obsoletas. Si esta prueba unitaria basada en simulacros no es visible para el autor de la implementación real y la implementación real cambia, no hay señal de que la prueba (y el código que se está probando) deba actualizarse para mantenerse al día con los cambios.

Tenga en cuenta que, como se menciona en [Capítulo 13](#), si los equipos proporcionan falsificaciones para sus propios servicios, esta preocupación se alivia en su mayor parte.

Problemas de configuración

Las pruebas unitarias cubren el código dentro de un binario dado. Pero ese binario normalmente no es completamente autosuficiente en términos de cómo se ejecuta. Por lo general, un binario tiene algún tipo de configuración de implementación o secuencia de comandos de inicio. Además, las instancias de producción reales que sirven al usuario final tienen sus propios archivos de configuración o bases de datos de configuración.

Si hay problemas con estos archivos o la compatibilidad entre el estado definido por estas tiendas y el binario en cuestión, esto puede generar problemas importantes para los usuarios. Las pruebas unitarias por sí solas no pueden verificar esta compatibilidad.¹ Por cierto, esta es una buena razón para asegurarse de que su configuración esté en el control de versiones, así como su código, porque entonces,

1 Ver “[Entrega continua](#)” en la página 483 y [capítulo 25](#) para más información.

los cambios en la configuración se pueden identificar como la fuente de errores en lugar de introducir errores externos aleatorios y se pueden incorporar a pruebas grandes.

En Google, los cambios de configuración son la razón número uno de nuestras principales interrupciones. Esta es un área en la que hemos tenido un rendimiento inferior y ha dado lugar a algunos de nuestros errores más vergonzosos. Por ejemplo, hubo una interrupción global de Google en 2013 debido a una mala configuración de red que nunca se probó. Las configuraciones tienden a escribirse en lenguajes de configuración, no en lenguajes de código de producción. También suelen tener ciclos de lanzamiento de producción más rápidos que los binarios y pueden ser más difíciles de probar. Todo esto conduce a una mayor probabilidad de fracaso. Pero al menos en este caso (y en otros), la configuración estaba controlada por la versión y pudimos identificar rápidamente al culpable y mitigar el problema.

Problemas que surgen bajo carga

En Google, las pruebas unitarias están diseñadas para ser pequeñas y rápidas porque deben encajar en nuestra infraestructura de ejecución de pruebas estándar y también ejecutarse muchas veces como parte de un flujo de trabajo de desarrollador sin fricciones. Pero las pruebas de rendimiento, carga y estrés a menudo requieren enviar grandes volúmenes de tráfico a un binario determinado. Estos volúmenes se vuelven difíciles de probar en el modelo de una prueba unitaria típica. Y nuestros grandes volúmenes son grandes, a menudo miles o millones de consultas por segundo (en el caso de los anuncios,[pujas en tiempo real](#))!

Comportamientos, entradas y efectos secundarios inesperados

Las pruebas unitarias están limitadas por la imaginación del ingeniero que las escribe. Es decir, solo pueden probar comportamientos y entradas anticipadas. Sin embargo, los problemas que los usuarios encuentran con un producto son en su mayoría imprevistos (de lo contrario, sería poco probable que lleguen a los usuarios finales como problemas). Este hecho sugiere que se necesitan diferentes técnicas de prueba para detectar comportamientos inesperados.

[Ley de Hyrum](#) es una consideración importante aquí: incluso si pudiéramos probar el 100% de la conformidad con un contrato estricto y específico, el contrato de usuario efectivo se aplica a todos los comportamientos visibles, no solo a un contrato establecido. Es poco probable que las pruebas unitarias por sí solas evalúen todos los comportamientos visibles que no están especificados en la API pública.

Comportamientos emergentes y el “efecto vacío”

Las pruebas unitarias están limitadas al ámbito que cubren (especialmente con el uso generalizado de dobles de prueba), por lo que si el comportamiento cambia en áreas fuera de este ámbito, no se puede detectar. Y debido a que las pruebas unitarias están diseñadas para ser rápidas y confiables, eliminan deliberadamente el caos de las dependencias reales, la red y los datos. Una prueba unitaria es como un problema de física teórica: instalada en el vacío, perfectamente escondida del desorden del mundo real, que es excelente para la velocidad y la confiabilidad pero no detecta ciertas categorías de defectos.

¿Por qué no tener pruebas más grandes?

En capítulos anteriores, discutimos muchas de las propiedades de una prueba fácil de usar para desarrolladores. En particular, debe ser de la siguiente manera:

De confianza

No debe ser escamoso y debe proporcionar una señal útil de aprobación/rechazo.

Rápido

Debe ser lo suficientemente rápido para no interrumpir el flujo de trabajo del desarrollador.

Escalable

Google necesita poder ejecutar todas las pruebas afectadas útiles de manera eficiente para los envíos previos y posteriores.

Las buenas pruebas unitarias exhiben todas estas propiedades. Las pruebas más grandes a menudo violan todas estas restricciones. Por ejemplo, las pruebas más grandes a menudo son más inestables porque usan más infraestructura que una prueba de unidad pequeña. También suelen ser mucho más lentos, tanto para configurar como para ejecutar. Y tienen problemas para escalar debido a los requisitos de recursos y tiempo, pero a menudo también porque no están aislados: estas pruebas pueden colisionar entre sí.

Además, las pruebas más grandes presentan otros dos desafíos. Primero, hay un desafío de propiedad. Una prueba de unidad es claramente propiedad del ingeniero (y el equipo) que posee la unidad. Una prueba más grande abarca varias unidades y, por lo tanto, puede abarcar varios propietarios. Esto presenta un desafío de propiedad a largo plazo: ¿quién es responsable de mantener la prueba y quién es responsable de diagnosticar problemas cuando la prueba falla? Sin una propiedad clara, una prueba se pudre.

El segundo desafío para las pruebas más grandes es el de la estandarización (o la falta de ella). A diferencia de las pruebas unitarias, las pruebas más grandes sufren una falta de estandarización en términos de la infraestructura y el proceso mediante el cual se escriben, ejecutan y depuran. El enfoque de las pruebas más grandes es un producto de las decisiones arquitectónicas de un sistema, lo que introduce una variación en el tipo de pruebas requeridas. Por ejemplo, la forma en que creamos y ejecutamos las pruebas de regresión AB diff en Google Ads es completamente diferente de la forma en que se crean y ejecutan dichas pruebas en los backends de búsqueda, que también es diferente de Drive. Usan diferentes plataformas, diferentes lenguajes, diferentes infraestructuras, diferentes bibliotecas y marcos de prueba competitivos.

Esta falta de estandarización tiene un impacto significativo. Debido a que las pruebas más grandes tienen tantas formas de ejecutarse, a menudo se omiten durante los cambios a gran escala. (Ver [capítulo 22](#).) La infraestructura no tiene una forma estándar de ejecutar esas pruebas, y pedirles a las personas que ejecutan LSC que conozcan los detalles locales para las pruebas en cada equipo no escala. Debido a que las pruebas más grandes difieren en la implementación de un equipo a otro, las pruebas que realmente prueban la integración entre esos equipos requieren la unificación de infraestructuras incompatibles. Y debido a esta falta de estandarización, no podemos enseñar a un

enfoque único para Nooglers (nuevos Googlers) o incluso ingenieros más experimentados, lo que perpetúa la situación y también conduce a una falta de comprensión sobre las motivaciones de tales pruebas.

Pruebas más grandes en Google

Cuando discutimos la historia de las pruebas en Google anteriormente (ver [Capítulo 11](#)), mencionamos cómo Google Web Server (GWS) ordenó pruebas automatizadas en 2003 y cómo este fue un momento decisivo. Sin embargo, en realidad teníamos pruebas automatizadas en uso antes de este punto, pero una práctica común era usar pruebas grandes y enormes automatizadas. Por ejemplo, AdWords creó una prueba integral en 2001 para validar escenarios de productos. De manera similar, en 2002, Search escribió una "prueba de regresión" similar para su código de indexación, y AdSense (que aún no se había lanzado públicamente) creó su variación en la prueba de AdWords.

También existían otros patrones de prueba "más grandes" alrededor de 2002. La interfaz de búsqueda de Google dependía en gran medida del control de calidad manual: versiones manuales de escenarios de prueba de extremo a extremo. Y Gmail obtuvo su versión de un entorno de "demostración local": una secuencia de comandos para abrir un entorno de Gmail de extremo a extremo localmente con algunos usuarios de prueba generados y datos de correo para pruebas manuales locales.

Cuando se lanzó C/J Build (nuestro primer marco de trabajo de construcción continua), no distinguía entre pruebas unitarias y otras pruebas, pero hubo dos desarrollos críticos que llevaron a una división. En primer lugar, Google se centró en las pruebas unitarias porque queríamos fomentar la pirámide de pruebas y garantizar que la gran mayoría de las pruebas escritas fueran pruebas unitarias. En segundo lugar, cuando TAP reemplazó a C/J Build como nuestro sistema formal de compilación continua, solo pudo hacerlo para las pruebas que cumplieron con los requisitos de elegibilidad de TAP: pruebas herméticas que se pueden compilar con un solo cambio y que podrían ejecutarse en nuestro clúster de compilación/prueba dentro de un límite de tiempo máximo. Aunque la mayoría de las pruebas unitarias cumplieron con este requisito, la mayoría de las pruebas más grandes no lo hicieron. Sin embargo, esto no detuvo la necesidad de otros tipos de pruebas, y han seguido llenando los vacíos de cobertura.

Pruebas más grandes y tiempo

A lo largo de este libro, hemos analizado la influencia del tiempo en la ingeniería de software, porque Google ha creado software que funciona desde hace más de 20 años. ¿Cómo se ven influenciadas las pruebas más grandes por la dimensión del tiempo? Sabemos que ciertas actividades tienen más sentido cuanto mayor sea la vida útil esperada del código, y la prueba de varias formas es una actividad que tiene sentido en todos los niveles, pero los tipos de prueba que son apropiados cambian durante la vida útil esperada del código.

Como señalamos antes, las pruebas unitarias comienzan a tener sentido para el software con una vida útil esperada de horas en adelante. A nivel de minutos (para guiones pequeños), manual

la prueba es más común, y el SUT generalmente se ejecuta localmente, pero es probable que la demostración locales producción, especialmente para guiones únicos, demostraciones o experimentos. En vidas más largas, las pruebas manuales continúan existiendo, pero los SUT generalmente divergen porque la instancia de producción a menudo está alojada en la nube en lugar de estar alojada localmente.

Todas las pruebas más grandes restantes brindan valor para el software de mayor duración, pero la principal preocupación se convierte en la capacidad de mantenimiento de tales pruebas a medida que aumenta el tiempo.

Por cierto, este impacto de tiempo podría ser una razón para el desarrollo del antipatrón de prueba de "cono de helado", como se menciona en el [Capítulo 11](#) y se muestra de nuevo en [Figura 14-2](#).

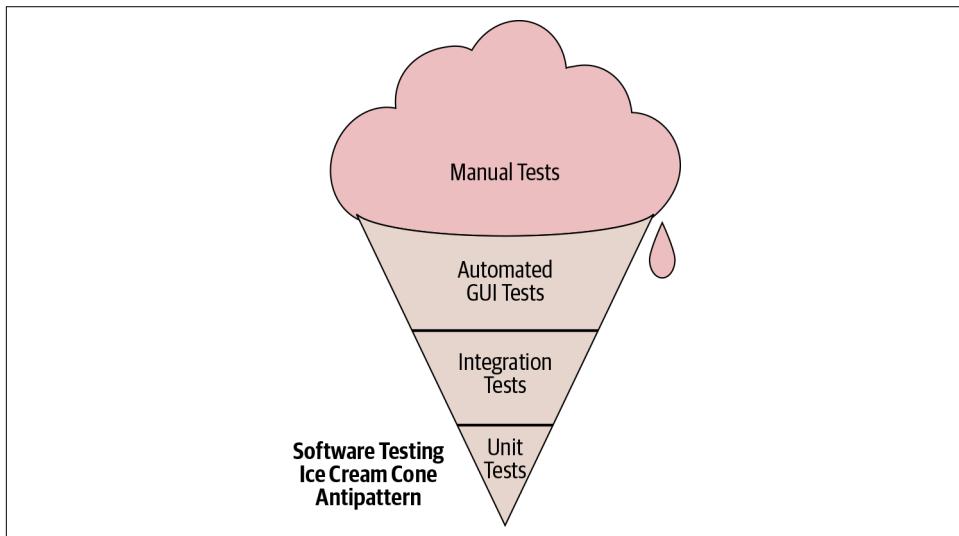


Figura 14-2. El antipatrón de prueba del cono de helado

Cuando el desarrollo comienza con pruebas manuales (cuando los ingenieros piensan que el código debe durar solo unos minutos), esas pruebas manuales se acumulan y dominan la cartera de pruebas general inicial. Por ejemplo, es bastante típico piratear un script o una aplicación y probarla ejecutándola, y luego continuar agregándole funciones pero continuar probándola ejecutándola manualmente. Este prototipo finalmente se vuelve funcional y se comparte con otros, pero en realidad no existen pruebas automatizadas para él.

Peor aún, si el código es difícil de probar unitariamente (debido a la forma en que se implementó en primer lugar), las únicas pruebas automatizadas que se pueden escribir son las de extremo a extremo, y sin darnos cuenta hemos creado un "código heredado". Dentro de días.

Está crítico para que la salud a largo plazo se mueva hacia la pirámide de prueba dentro de los primeros días de desarrollo mediante la creación de pruebas unitarias, y luego completarlo después de ese punto mediante la introducción de pruebas de integración automatizadas y alejarse de las pruebas manuales de extremo a extremo. Tuvimos éxito al hacer que las pruebas unitarias fueran un requisito para la presentación, pero

cubrir la brecha entre las pruebas unitarias y las pruebas manuales es necesario para la salud a largo plazo.

Pruebas más grandes a escala de Google

Parecería que las pruebas más grandes deberían ser más necesarias y más apropiadas a escalas más grandes de software, pero aunque esto sea así, la complejidad de crear, ejecutar, mantener y depurar estas pruebas aumenta con el crecimiento de la escala, incluso más. que con las pruebas unitarias.

En un sistema compuesto por microservicios o servidores separados, el patrón de interconexiones se parece a un gráfico: dejemos que el número de nodos en ese gráfico sea nuestro *norte*. Cada vez que se agrega un nuevo nodo a este gráfico, hay un efecto multiplicador en el número de rutas de ejecución distintas a través de él.

Figura 14-3 representa un SUT imaginario: este sistema consta de una red social con usuarios, un gráfico social, un flujo de publicaciones y algunos anuncios mezclados. Los anuncios son creados por los anunciantes y se muestran en el contexto del flujo social. Solo este SUT consta de dos grupos de usuarios, dos interfaces de usuario, tres bases de datos, una canalización de indexación y seis servidores. Hay 14 aristas enumeradas en el gráfico. Probar todas las posibilidades de extremo a extremo ya es difícil. Imagínese si agregamos más servicios, canalizaciones y bases de datos a esta combinación: fotos e imágenes, análisis de fotos de aprendizaje automático, etc.

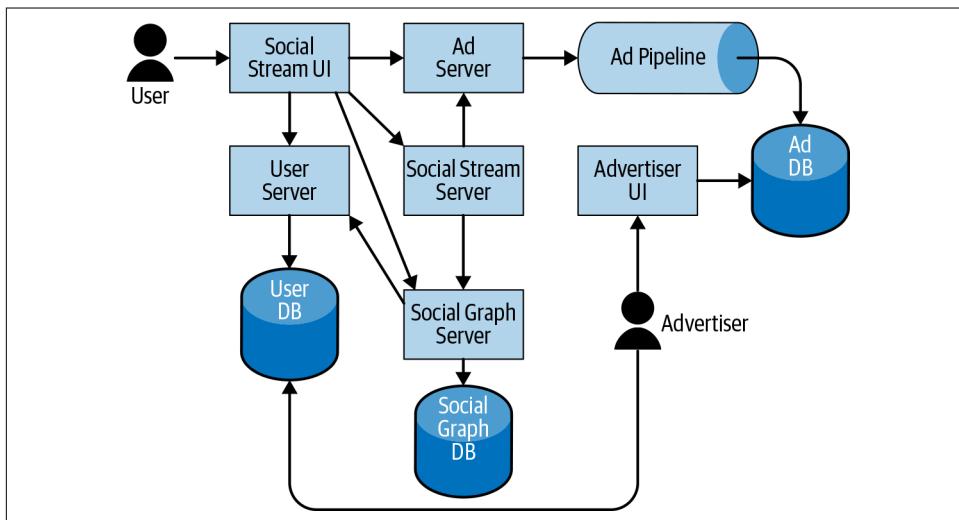


Figura 14-3. Ejemplo de un SUT bastante pequeño: una red social con publicidad

La tasa de distintos escenarios para probar de un extremo a otro puede crecer de manera exponencial o combinatoria dependiendo de la estructura del sistema bajo prueba, y ese crecimiento no se escala. Por lo tanto, a medida que el sistema crece, debemos encontrar estrategias de prueba alternativas más grandes para mantener las cosas manejables.

Sin embargo, el valor de tales pruebas también aumenta debido a las decisiones que fueron necesarias para lograr esta escala. Este es un impacto de la fidelidad: a medida que avanzamos hacia *norte* capas de software, si los dobles del servicio son de menor fidelidad ($1 - \epsilon$), la posibilidad de errores al ponerlo todo junto es exponencial en *norte*. Mirando de nuevo este SUT de ejemplo, si reemplazamos el servidor de usuario y el servidor de anuncios con dobles y esos dobles son de baja fidelidad (por ejemplo, 10 % de precisión), la probabilidad de un error es del 99 % ($1 - (0.1 * 0.1)$). Y eso es solo con dos dobles de baja fidelidad.

Por lo tanto, se vuelve crítico implementar pruebas más grandes de manera que funcionen bien a esta escala pero que mantengan una fidelidad razonablemente alta.

Consejo: "La prueba más pequeña posible"

Incluso para las pruebas de integración, más pequeño es mejor: un puñado de pruebas grandes es preferible a una enorme. Y, debido a que el alcance de una prueba a menudo está acoplado al alcance del SUT, encontrar formas de hacer que el SUT sea más pequeño ayuda a que la prueba sea más pequeña.

Una forma de lograr esta relación de prueba cuando se presenta un viaje de usuario que puede requerir contribuciones de muchos sistemas internos es "encadenar" pruebas, como se ilustra en [Figura 14-4](#), no específicamente en su ejecución, sino para crear múltiples pruebas de integración por pares más pequeñas que representen el escenario general. Esto se hace asegurándose de que la salida de una prueba se utilice como entrada para otra prueba conservando esta salida en un depósito de datos.

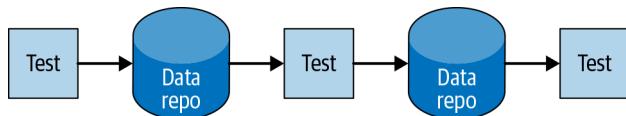


Figura 14-4. Pruebas encadenadas

Estructura de una prueba grande

Aunque las pruebas grandes no están sujetas a restricciones de pruebas pequeñas y posiblemente podrían consistir en cualquier cosa, la mayoría de las pruebas grandes exhiben patrones comunes. Las pruebas grandes generalmente consisten en un flujo de trabajo con las siguientes fases:

- Obtener un sistema bajo prueba
- Semilla de datos de prueba necesarios
- Realizar acciones utilizando el sistema bajo prueba
- Verificar comportamientos

El sistema bajo prueba

Un componente clave de las pruebas grandes es el SUT antes mencionado (ver Figura 14-5). Una prueba de unidad típica enfoca su atención en una clase o módulo. Además, el código de prueba se ejecuta en el mismo proceso (o Java Virtual Machine [JVM], en el caso de Java) que el código que se está probando. Para pruebas más grandes, el SUT suele ser muy diferente; uno o más procesos separados con código de prueba a menudo (pero no siempre) en su propio proceso.

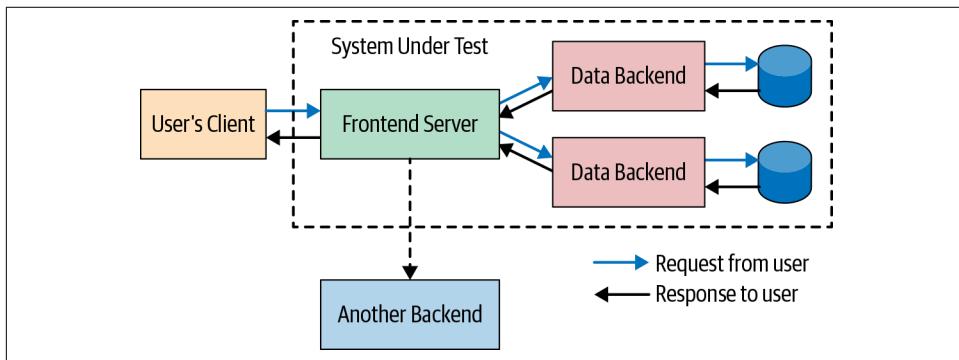


Figura 14-5. Un ejemplo de sistema bajo prueba (SUT)

En Google, usamos muchas formas diferentes de SUT, y el alcance del SUT es uno de los principales impulsores del alcance de la prueba grande en sí (cuanto más grande es el SUT, más grande es la prueba). Cada formulario de IVU se puede juzgar en función de dos factores principales:

hermeticidad

Este es el aislamiento del SUT de los usos e interacciones de otros componentes además de la prueba en cuestión. Un SUT con alta hermeticidad tendrá la menor exposición a fuentes de concurrencia y fallas en la infraestructura.

Fidelidad

La precisión del SUT para reflejar el sistema de producción que se está probando. Un SUT con alta fidelidad constará de archivos binarios que se asemejan a las versiones de producción (se basan en configuraciones similares, usan infraestructuras similares y tienen una topología general similar).

A menudo, estos dos factores están en conflicto directo.

Los siguientes son algunos ejemplos de IVU:

IVU de proceso único

Todo el sistema bajo prueba se empaqueta en un solo binario (incluso si en producción estos son múltiples binarios separados). Además, el código de prueba se puede empaquetar en el mismo binario que el SUT. Tal combinación de prueba-SUT puede ser una prueba "pequeña" si todo es de un solo subproceso, pero es la menos fiel a la topología y configuración de producción.

IVU máquina única

El sistema bajo prueba consta de uno o más binarios separados (igual que la producción) y la prueba es su propio binario. Pero todo se ejecuta en una máquina. Esto se utiliza para pruebas "medianas". Idealmente, usamos la configuración de lanzamiento de producción de cada binario cuando ejecutamos esos binarios localmente para aumentar la fidelidad.

TOU multimáquina

El sistema bajo prueba se distribuye en varias máquinas (muy parecido a una implementación de nube de producción). Esta es una fidelidad aún mayor que el SUT de una sola máquina, pero su uso hace que las pruebas sean de tamaño "grande" y la combinación es susceptible a una mayor descamación de la red y la máquina.

Entornos compartidos (puesta en escena y producción)

En lugar de ejecutar un SUT independiente, la prueba solo usa un entorno compartido. Esto tiene el costo más bajo porque estos entornos compartidos generalmente ya existen, pero la prueba puede entrar en conflicto con otros usos simultáneos y uno debe esperar a que el código se envíe a esos entornos. La producción también aumenta el riesgo de impacto en el usuario final.

Híbridos

Algunos SUT representan una combinación: podría ser posible ejecutar algunos de los SUT pero hacer que interactúe con un entorno compartido. Por lo general, lo que se prueba se ejecuta explícitamente, pero sus backends se comparten. Para una empresa tan expansiva como Google, es prácticamente imposible ejecutar varias copias de todos los servicios interconectados de Google, por lo que se requiere cierta hibridación.

Los beneficios de los IVU herméticos

El SUT en una prueba grande puede ser una fuente importante tanto de falta de confiabilidad como de tiempo de respuesta prolongado. Por ejemplo, una prueba en producción utiliza la implementación real del sistema de producción. Como se mencionó anteriormente, esto es popular porque no hay costos generales adicionales para el entorno, pero las pruebas de producción no se pueden ejecutar hasta que el código llegue a ese entorno, lo que significa que esas pruebas no pueden bloquear la publicación del código en ese entorno: el SUT es demasiado tarde, esencialmente.

La primera alternativa más común es crear un entorno de ensayo compartido gigante y ejecutar pruebas allí. Esto generalmente se hace como parte de algún proceso de promoción de lanzamiento, pero nuevamente limita la ejecución de la prueba solo cuando el código está disponible. Como alternativa, algunos equipos permitirán que los ingenieros "reserven" tiempo en el entorno de prueba y usen ese período de tiempo para implementar el código pendiente y ejecutar pruebas, pero esto no se escala con un número creciente de ingenieros o un número creciente de servicios., porque el entorno, su número de usuarios y la probabilidad de conflictos entre usuarios crecen rápidamente.

El siguiente paso es admitir SUT aislados en la nube o herméticos para máquinas. Dicho entorno mejora la situación al evitar los conflictos y los requisitos de reserva para la liberación del código.

Estudio de caso: Riesgos de las pruebas en producción y Webdriver Torso

Mencionamos que las pruebas en producción pueden ser riesgosas. Un episodio humorístico resultante de las pruebas en producción se conoció como el incidente del Webdriver Torso. Necesitábamos una forma de verificar que la reproducción de video en la producción de YouTube funcionaba correctamente, por lo que creamos secuencias de comandos automatizadas para generar videos de prueba, cargarlos y verificar la calidad de la carga. Esto se hizo en un canal de YouTube propiedad de Google llamado Webdriver Torso. Pero este canal era público, al igual que la mayoría de los videos.

Posteriormente, este canal fue publicitado en [un artículo en Wired](#), lo que propició su difusión en los medios y los posteriores esfuerzos por resolver el misterio. Finalmente, [un bloguero](#) vinculó todo a Google. Eventualmente, salimos limpios divirtiéndonos un poco, incluyendo un Rickroll y un Easter Egg, así que todo salió bien. Pero debemos pensar en la posibilidad de que el usuario final descubra cualquier dato de prueba que incluyamos en producción y estar preparados para ello.

Reducir el tamaño de su SUT en los límites del problema

Hay límites de prueba particularmente dolorosos que podría valer la pena evitar. Las pruebas que involucran tanto frontends como backends se vuelven dolorosas porque las pruebas de interfaz de usuario (UI) son notoriamente poco confiables y costosas:

- Las IU a menudo cambian en formas que hacen que las pruebas de IU sean frágiles pero que en realidad no afectan el comportamiento subyacente.
- Las interfaces de usuario suelen tener comportamientos asincrónicos que son difíciles de probar.

Si bien es útil tener pruebas de extremo a extremo de la interfaz de usuario de un servicio hasta su backend, estas pruebas tienen un costo de mantenimiento multiplicativo tanto para la interfaz de usuario como para los backends. En cambio, si el backend proporciona una API pública, a menudo es más fácil dividir las pruebas en pruebas conectadas en el límite de UI/API y usar la API pública para impulsar las pruebas de un extremo a otro. Esto es cierto ya sea que la interfaz de usuario sea un navegador, una interfaz de línea de comandos (CLI), una aplicación de escritorio o una aplicación móvil.

Otro límite especial es para las dependencias de terceros. Es posible que los sistemas de terceros no tengan un entorno público compartido para realizar pruebas y, en algunos casos, hay un costo por enviar tráfico a un tercero. Por lo tanto, no se recomienda que las pruebas automatizadas usen una API de terceros real, y esa dependencia es una unión importante en la que dividir las pruebas.

Para solucionar este problema de tamaño, hemos reducido este SUT reemplazando sus bases de datos con bases de datos en memoria y eliminando uno de los servidores fuera del alcance del SUT que realmente nos interesa, como se muestra en [Figura 14-6](#). Es más probable que este SUT encaje en una sola máquina.

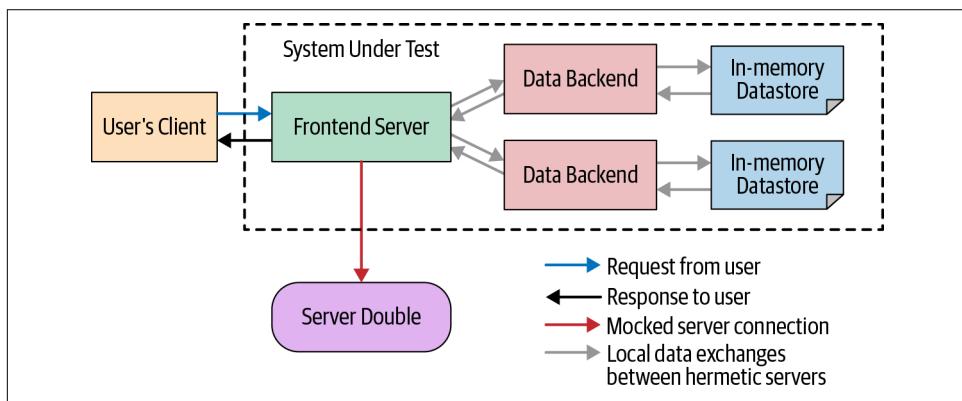


Figura 14-6. Un SUT de tamaño reducido

La clave es identificar las compensaciones entre fidelidad y costo/confiabilidad, e identificar límites razonables. Si podemos ejecutar un puñado de binarios y una prueba y empaquetar todo en las mismas máquinas que realizan nuestras compilaciones regulares, enlaces y ejecuciones de pruebas unitarias, tenemos las pruebas de "integración" más fáciles y estables para nuestros ingenieros.

Proxies de grabación/reproducción

En el capítulo anterior, discutimos los dobles de prueba y los enfoques que se pueden usar para desacoplar la clase bajo prueba de sus dependencias difíciles de probar. También podemos duplicar servidores y procesos completos mediante el uso de un servidor o proceso simulado, auxiliar o falso con la API equivalente. Sin embargo, no hay garantía de que el doble de prueba utilizado realmente se ajuste al contrato de la cosa real que está reemplazando.

Una forma de tratar con los servicios dependientes pero subsidiarios de un SUT es usar un doble de prueba, pero ¿cómo se sabe que el doble refleja el comportamiento real de la dependencia? Un enfoque cada vez mayor fuera de Google es utilizar un marco para [contrato impulsado por el consumidor](#) pruebas. Son pruebas que definen un contrato tanto para el cliente como para el proveedor del servicio, y este contrato puede impulsar pruebas automatizadas. Es decir, un cliente define un simulacro del servicio diciendo que, para estos argumentos de entrada, obtengo una salida particular. Luego, el servicio real usa este par de entrada/salida en una prueba real para garantizar que produce esa salida dadas esas entradas. Dos herramientas públicas para la prueba de contratos impulsada por el consumidor son [Pruebas de contrato de pacto](#) y [Contratos de Spring Cloud](#). La gran dependencia de Google de los búferes de protocolo significa que no los usamos internamente.

En Google, hacemos algo un poco diferente. **Nuestro enfoque más popular** (para el cual hay una API pública) es usar una prueba más grande para generar una más pequeña al registrar el tráfico a esos servicios externos cuando se ejecuta la prueba más grande y reproducirlo cuando se ejecutan pruebas más pequeñas. La prueba más grande, o "Modo de registro", se ejecuta continuamente después del envío, pero su propósito principal es generar estos registros de tráfico (sin embargo, debe pasar para que se generen los registros). La prueba más pequeña, o "Modo de reproducción", se usa durante el desarrollo y las pruebas previas al envío.

Uno de los aspectos interesantes de cómo funciona la grabación/reproducción es que, debido al no determinismo, las solicitudes deben coincidir a través de un comparador para determinar qué respuesta reproducir. Esto los hace muy similares a los stubs y simulacros en el sentido de que se utiliza la coincidencia de argumentos para determinar el comportamiento resultante.

¿Qué sucede con las pruebas nuevas o las pruebas en las que el comportamiento del cliente cambia significativamente? En estos casos, es posible que una solicitud ya no coincida con lo que está en el archivo de tráfico registrado, por lo que la prueba no puede pasarse en el modo de reproducción. En esa circunstancia, el ingeniero debe ejecutar la prueba en modo Record para generar nuevo tráfico, por lo que es importante que la ejecución de las pruebas Record sea fácil, rápida y estable.

Datos de prueba

Una prueba necesita datos, y una prueba grande necesita dos tipos diferentes de datos:

datos sembrados

Datos preinicializados en el sistema bajo prueba que reflejan el estado del SUT al inicio de la prueba

Trafic de prueba

Datos enviados al sistema bajo prueba por la propia prueba durante su ejecución

Debido a la noción de un SUT separado y más grande, el trabajo para generar el estado del SUT suele ser mucho más complejo que el trabajo de configuración realizado en una prueba unitaria. Por ejemplo:

datos de dominio

Algunas bases de datos contienen datos rellenos previamente en tablas y se utilizan como configuración para el entorno. Los archivos binarios de servicios reales que utilizan una base de datos de este tipo pueden fallar al iniciarse si no se proporcionan los datos del dominio.

Línea de base realista

Para que un SUT se perciba como realista, podría requerir un conjunto realista de datos básicos al inicio, tanto en términos de calidad como de cantidad. Por ejemplo, las pruebas grandes de una red social probablemente necesiten un gráfico social realista como estado base para las pruebas: deben existir suficientes usuarios de prueba con perfiles realistas, así como suficientes interconexiones entre esos usuarios para que se acepte la prueba.

Semillas de API

Las API mediante las cuales se generan los datos pueden ser complejas. Podría ser posible escribir directamente en un almacén de datos, pero hacerlo podría omitir los activadores y las comprobaciones realizadas por los archivos binarios reales que realizan las escrituras.

Los datos se pueden generar de diferentes maneras, como las siguientes:

Datos hechos a mano

Al igual que para las pruebas más pequeñas, podemos crear datos de prueba para pruebas más grandes a mano. Pero podría requerir más trabajo configurar datos para múltiples servicios en un SUT grande, y es posible que necesitemos crear una gran cantidad de datos para pruebas más grandes.

datos copiados

Podemos copiar datos, generalmente de producción. Por ejemplo, podríamos probar un mapa de la Tierra comenzando con una copia de los datos de nuestro mapa de producción para proporcionar una línea de base y luego probar nuestros cambios en él.

Datos muestreados

La copia de datos puede proporcionar demasiados datos para trabajar razonablemente. Los datos de muestreo pueden reducir el volumen, lo que reduce el tiempo de prueba y facilita el razonamiento. El “muestreo inteligente” consiste en técnicas para copiar los datos mínimos necesarios para lograr la máxima cobertura.

Verificación

Después de que se ejecuta un SUT y se le envía tráfico, aún debemos verificar el comportamiento. Hay algunas maneras diferentes de hacer esto:

Manual

Al igual que cuando prueba su binario localmente, la verificación manual utiliza humanos para interactuar con un SUT para determinar si funciona correctamente. Esta verificación puede consistir en probar las regresiones mediante la realización de acciones como se define en un plan de prueba consistente o puede ser exploratoria, trabajando a través de diferentes rutas de interacción para identificar posibles fallas nuevas.

Tenga en cuenta que las pruebas de regresión manual no se escalan de forma sublineal: cuanto más crece un sistema y más viajes hay a través de él, más tiempo humano se necesita para probar manualmente.

afirmaciones

Al igual que con las pruebas unitarias, estas son verificaciones explícitas sobre el comportamiento previsto del sistema. Por ejemplo, para una prueba de integración de la búsqueda de Google dexyzy, una afirmación podría ser la siguiente:

afirmar que (respuesta. Contiene ("Cueva colosal"))

Comparación A/B (diferencial)

En lugar de definir afirmaciones explícitas, las pruebas A/B implican ejecutar dos copias del SUT, enviar los mismos datos y comparar la salida. El comportamiento previsto no está definido explícitamente: un ser humano debe revisar manualmente las diferencias para asegurarse de que se pretende realizar cualquier cambio.

Tipos de pruebas más grandes

Ahora podemos combinar estos diferentes enfoques del SUT, los datos y las afirmaciones para crear diferentes tipos de pruebas grandes. Cada prueba tiene diferentes propiedades en cuanto a qué riesgos mitiga; cuánto trabajo se requiere para escribirlo, mantenerlo y depurarlo; y cuánto cuesta en términos de recursos para funcionar.

Lo que sigue es una lista de diferentes tipos de pruebas grandes que usamos en Google, cómo están compuestas, para qué sirven y cuáles son sus limitaciones:

- Pruebas funcionales de uno o más binarios
- Pruebas de navegadores y dispositivos
- Pruebas de rendimiento, carga y estrés
- Pruebas de configuración de implementación
- **Prueba exploratoria**
- Pruebas de diferencias A/B (regresión)
- Pruebas de aceptación del usuario (UAT)
- **Sondeos y análisis canary**
- Recuperación ante desastres e ingeniería del caos
- Evaluación del usuario

Dado un número tan amplio de combinaciones y, por lo tanto, una amplia gama de pruebas, ¿cómo gestionamos qué hacer y cuándo? Parte del diseño de software es redactar el plan de prueba, y una parte clave del plan de prueba es un esquema estratégico de qué tipos de prueba se necesitan y cuánto de cada uno. Esta estrategia de prueba identifica los principales vectores de riesgo y los enfoques de prueba necesarios para mitigar esos vectores de riesgo.

En Google, tenemos una función de ingeniería especializada de "Ingeniero de pruebas", y una de las cosas que buscamos en un buen ingeniero de pruebas es la capacidad de delinejar una estrategia de prueba para nuestros productos.

Pruebas funcionales de uno o más binarios que interactúan

Las pruebas de este tipo tienen las siguientes características:

- SUT: máquina única hermética o implementada en la nube aislada
- Datos: hecho a mano
- Verificación: afirmaciones

Como hemos visto hasta ahora, las pruebas unitarias no son capaces de probar un sistema complejo con verdadera fidelidad, simplemente porque se empaquetan de una manera diferente a como se empaqueta el código real. Muchos escenarios de pruebas funcionales interactúan con un binario dado de manera diferente que con las clases dentro de ese binario, y estas pruebas funcionales requieren SUT separados y, por lo tanto, son pruebas canónicas más grandes.

Probar las interacciones de múltiples binarios es, como era de esperar, incluso más complicado que probar un solo binario. Un caso de uso común es dentro de los entornos de microservicios cuando los servicios se implementan como muchos binarios separados. En este caso, una prueba funcional puede cubrir las interacciones reales entre los binarios mostrando un SUT compuesto por todos los binarios relevantes e interactuando con él a través de una API publicada.

Pruebas de navegadores y dispositivos

La prueba de interfaces de usuario web y aplicaciones móviles es un caso especial de prueba funcional de uno o más binarios que interactúan. Es posible realizar pruebas unitarias del código subyacente, pero para los usuarios finales, la API pública es la propia aplicación. Tener pruebas que interactúen con la aplicación como un tercero a través de su interfaz brinda una capa adicional de cobertura.

Pruebas de rendimiento, carga y estrés

Las pruebas de este tipo tienen las siguientes características:

- SUT: aislado implementado en la nube
- Datos: hechos a mano o multiplexados de producción
- Verificación: diff (métricas de rendimiento)

Aunque es posible probar una unidad pequeña en términos de rendimiento, carga y estrés, a menudo dichas pruebas requieren el envío simultáneo de tráfico a una API externa. Esta definición implica que tales pruebas son pruebas de subprocesos múltiples que generalmente prueban en el alcance de un binario bajo prueba. Sin embargo, estas pruebas son críticas para garantizar que no haya una degradación en el rendimiento entre versiones y que el sistema pueda manejar los picos esperados en el tráfico.

A medida que crece la escala de la prueba de carga, también crece el alcance de los datos de entrada y, finalmente, se vuelve difícil generar la escala de carga necesaria para desencadenar errores bajo carga. El manejo de cargas y tensiones son propiedades “altamente emergentes” de un sistema; es decir, estos comportamientos complejos pertenecen al sistema global pero no a los miembros individuales. Por lo tanto, es importante hacer que estas pruebas parezcan lo más cercanas posible a la producción. Cada SUT requiere recursos similares a los que requiere la producción, y se vuelve difícil mitigar el ruido de la topología de producción.

Un área de investigación para eliminar el ruido en las pruebas de rendimiento es modificar la topología de implementación: cómo se distribuyen los diversos binarios en una red de máquinas. La máquina que ejecuta un binario puede afectar las características de rendimiento; por lo tanto, si en una prueba de diferencia de rendimiento, la versión base se ejecuta en una máquina rápida (o una con una red rápida) y la nueva versión en una lenta, puede parecer una regresión de rendimiento. Esta característica implica que la implementación óptima es ejecutar ambas versiones en la misma máquina. Si una sola máquina no puede adaptarse a ambas versiones del binario, una alternativa es calibrar realizando varias ejecuciones y eliminando picos y valles.

Pruebas de configuración de implementación

Las pruebas de este tipo tienen las siguientes características:

- SUT: máquina única hermética o implementada en la nube aislada
- Datos: ninguno
- Verificación: aserciones (no falla)

Muchas veces, no es el código el origen de los defectos, sino la configuración: archivos de datos, bases de datos, definiciones de opciones, etc. Las pruebas más grandes pueden probar la integración del SUT con sus archivos de configuración porque estos archivos de configuración se leen durante el lanzamiento del binario dado.

Tal prueba es realmente una prueba de humo del SUT sin necesidad de muchos datos adicionales o verificación. Si el SUT se inicia con éxito, la prueba pasa. Si no, la prueba falla.

Prueba exploratoria

Las pruebas de este tipo tienen las siguientes características:

- SUT: producción o montaje compartido
- Datos: producción o un universo de prueba conocido
- Verificación: manual

Prueba exploratoria²es una forma de prueba manual que no se enfoca en buscar regresiones de comportamiento mediante la repetición de flujos de prueba conocidos, sino en buscar comportamientos cuestionables probando nuevos escenarios de usuarios. Los usuarios/evaluadores capacitados interactúan con un producto a través de sus API públicas, buscando nuevas rutas a través del sistema y cuyo comportamiento se desvía del comportamiento esperado o intuitivo, o si existen vulnerabilidades de seguridad.

Las pruebas exploratorias son útiles tanto para los sistemas nuevos como para los lanzados para descubrir comportamientos y efectos secundarios inesperados. Al hacer que los evaluadores sigan diferentes rutas accesibles a través del sistema, podemos aumentar la cobertura del sistema y, cuando estos evaluadores identifican errores, capturan nuevas pruebas funcionales automatizadas. En cierto sentido, esto es un poco como una versión manual de "prueba de fuzz" de la prueba de integración funcional.

Limitaciones

Las pruebas manuales no escalan de forma sublineal; es decir, requiere tiempo humano para realizar las pruebas manuales. Cualquier defecto encontrado por las pruebas exploratorias debe replicarse con una prueba automatizada que pueda ejecutarse con mucha más frecuencia.

Golpes de errores

Un enfoque común que usamos para las pruebas exploratorias manuales es elataque de errores. Un equipo de ingenieros y personal relacionado (gerentes, gerentes de producto, ingenieros de prueba, cualquiera que esté familiarizado con el producto) programa una "reunión", pero en esta sesión, todos los involucrados prueban manualmente el producto. Puede haber algunas pautas publicadas en cuanto a áreas de enfoque particulares para el bug bash y/o puntos de partida para usar el sistema, pero el objetivo es proporcionar suficiente variedad de interacción para documentar comportamientos cuestionables del producto y errores absolutos.

Pruebas de regresión de diferencias A/B

Las pruebas de este tipo tienen las siguientes características:

- SUT: dos entornos aislados implementados en la nube
- Datos: generalmente multiplexados de producción o muestreados
- Verificación: comparación de diferencias A/B

Las pruebas unitarias cubren las rutas de comportamiento esperadas para una pequeña sección de código. Pero es imposible predecir muchos de los posibles modos de falla para un producto dado de cara al público. Además, como establece la Ley de Hyrum, la API pública real no es la declarada sino

² James A. Whittaker, *Pruebas exploratorias de software: consejos, trucos, recorridos y técnicas para guiar el diseño de pruebas* (Nueva York: Addison-Wesley Professional, 2009).

todos los aspectos visibles para el usuario de un producto. Dadas esas dos propiedades, no sorprende que las pruebas de diferencias A/B sean posiblemente la forma más común de pruebas más grandes en Google. Este enfoque se remonta conceptualmente a 1998. En Google, hemos estado realizando pruebas basadas en este modelo desde 2001 para la mayoría de nuestros productos, comenzando con Anuncios, Búsqueda y Maps.

Las pruebas de diferencias A/B funcionan enviando tráfico a una API pública y comparando las respuestas entre versiones antiguas y nuevas (especialmente durante las migraciones). Cualquier desviación en el comportamiento debe reconciliarse como anticipada o no anticipada (regresiones). En este caso, el SUT se compone de dos conjuntos de binarios reales: uno que se ejecuta en la versión candidata y otro que se ejecuta en la versión base. Un tercer binario envía tráfico y compara los resultados.

Hay otras variantes. Usamos pruebas AA (comparando un sistema consigo mismo) para identificar el comportamiento no determinista, el ruido y la descamación, y para ayudar a eliminarlos de las diferencias AB. Ocasionalmente, también usamos pruebas ABC, comparando la última versión de producción, la compilación de referencia y un cambio pendiente, para que sea fácil ver de un vistazo no solo el impacto de un cambio inmediato, sino también los impactos acumulados de lo que sería el próxima versión de lanzamiento.

Las pruebas de diferencias A/B son una forma económica pero automatizable de detectar efectos secundarios imprevistos para cualquier sistema lanzado.

Limitaciones

Las pruebas de diferencia presentan algunos desafíos para resolver:

Aprobación

Alguien debe comprender los resultados lo suficiente como para saber si se esperan diferencias. A diferencia de una prueba típica, no está claro si las diferencias son buenas o malas (o si la versión de referencia es realmente válida), por lo que a menudo hay un paso manual en el proceso.

Ruido

Para una prueba diff, cualquier cosa que introduzca ruido inesperado en los resultados conduce a una investigación más manual de los resultados. Se vuelve necesario remediar el ruido, y esta es una gran fuente de complejidad en la construcción de una buena prueba de diferencias.

Cobertura

Generar suficiente tráfico útil para una prueba de diferencias puede ser un problema desafiante. Los datos de prueba deben cubrir suficientes escenarios para identificar las diferencias de caso de esquina, pero es difícil curar manualmente dichos datos.

Configuración

Configurar y mantener un SUT es bastante desafiante. Crear dos a la vez puede duplicar la complejidad, especialmente si comparten interdependencias.

UAT

Las pruebas de este tipo tienen las siguientes características:

- SUT: máquina hermética o implementada en la nube aislada
- Datos: hecho a mano
- Verificación: afirmaciones

Un aspecto clave de las pruebas unitarias es que las escribe el desarrollador que escribe el código bajo prueba. Pero eso hace que sea bastante probable que los malentendidos sobre el *destinado* comportamiento de un producto se reflejan no solo en el código, sino también en las pruebas unitarias. Dichas pruebas unitarias verifican que el código "Funciona según lo implementado" en lugar de "Funciona según lo previsto".

Para los casos en los que hay un cliente final específico o un proxy del cliente (un comité de clientes o incluso un gerente de producto), las UAT son pruebas automatizadas que ejercitan el producto a través de API públicas para garantizar el comportamiento general para determinados. [viajes de usuarios](#) como se pretendía. Existen múltiples marcos públicos (p. ej., Cucumber y RSpec) para hacer que tales pruebas se puedan escribir/leer en un lenguaje fácil de usar, a menudo en el contexto de "especificaciones ejecutables".

Google en realidad no hace mucho UAT automatizado y no usa mucho los lenguajes de especificación. Históricamente, muchos de los productos de Google han sido creados por los propios ingenieros de software. Ha habido poca necesidad de lenguajes de especificación ejecutables porque aquellos que definen el comportamiento previsto del producto a menudo dominan los lenguajes de codificación reales.

Probers y Canary Analysis

Las pruebas de este tipo tienen las siguientes características:

- IVU: producción
- Datos: producción
- Verificación: aserciones y diferencia A/B (de métricas)

Los sondeos y el análisis controlado son formas de garantizar que el entorno de producción en sí sea saludable. En estos aspectos, son una forma de seguimiento de la producción, pero son estructuralmente muy similares a otras pruebas grandes.

Los sondadores son pruebas funcionales que ejecutan afirmaciones codificadas contra el entorno de producción. Por lo general, estas pruebas realizan acciones conocidas y deterministas de solo lectura para que las afirmaciones se mantengan aunque los datos de producción cambien con el tiempo. Por ejemplo, un probador podría realizar una búsqueda en Google en [www.google.com](#) y verificar que se devuelva un resultado, pero en realidad no verificar el contenido del resultado. En ese sentido,

son "pruebas de humo" del sistema de producción, pero proporcionan una detección temprana de problemas importantes.

El análisis de Canary es similar, excepto que se centra en cuándo se envía una versión al entorno de producción. Si el lanzamiento se escalona a lo largo del tiempo, podemos ejecutar afirmaciones de sondeo dirigidas a los servicios actualizados (canary) y comparar las métricas de estado de las partes de producción canary y de referencia y asegurarnos de que no estén fuera de línea.

Las sondas deben usarse en cualquier sistema vivo. Si el proceso de implementación de producción incluye una fase en la que el binario se implementa en un subconjunto limitado de las máquinas de producción (una fase canary), se debe utilizar el análisis canary durante ese procedimiento.

Limitaciones

Cualquier problema detectado en este momento (en producción) ya está afectando a los usuarios finales.

Si un probador realiza una acción mutable (escribir), modificará el estado de producción. Esto podría conducir a uno de tres resultados: no determinismo y falla de las afirmaciones, falla de la capacidad de escribir en el futuro o efectos secundarios visibles para el usuario.

Recuperación ante desastres e ingeniería del caos

Las pruebas de este tipo tienen las siguientes características:

- IVU: producción
- Datos: producción y creados por el usuario (inyección de fallas)
- Verificación: manual y A/B diff (métricas)

Estos prueban qué tan bien reaccionarán sus sistemas ante cambios o fallas inesperados.

Durante años, Google ha realizado un juego de guerra anual llamado **Tierra** (Disaster Recovery Testing) durante el cual se inyectan fallas en nuestra infraestructura a una escala casi planetaria. Simulamos todo, desde incendios en centros de datos hasta ataques maliciosos. En un caso memorable, simulamos un terremoto que aisló completamente nuestra sede en Mountain View, California, del resto de la empresa. Hacerlo expuso no solo las deficiencias técnicas, sino que también reveló el desafío de administrar una empresa cuando todos los tomadores de decisiones clave eran inalcanzables.³

Los impactos de las pruebas DiRT requieren mucha coordinación en toda la empresa; por el contrario, la ingeniería del caos es más una "prueba continua" para su infraestructura técnica. **Popularizado por Netflix**, la ingeniería del caos implica escribir programas que

³ Durante esta prueba, casi nadie pudo hacer nada, por lo que muchas personas dejaron de trabajar y fueron a uno de nuestros muchos cafés, y al hacerlo, ¡terminamos creando un ataque DDoS en nuestros equipos de café!

introducir continuamente un nivel de fondo de fallas en sus sistemas y ver qué sucede. Algunas de las fallas pueden ser bastante grandes, pero en la mayoría de los casos, las herramientas de prueba de caos están diseñadas para restaurar la funcionalidad antes de que las cosas se salgan de control. El objetivo de la ingeniería del caos es ayudar a los equipos a romper las suposiciones de estabilidad y confiabilidad y ayudarlos a lidiar con los desafíos de desarrollar la resiliencia. Hoy en día, los equipos de Google realizan miles de pruebas de caos cada semana utilizando nuestro propio sistema llamado Catzilla..

Estos tipos de pruebas negativas y de fallas tienen sentido para los sistemas de producción en vivo que tienen suficiente tolerancia teórica a fallas para respaldarlos y para los cuales los costos y riesgos de las pruebas en sí son asequibles.

Limitaciones

Cualquier problema detectado en este momento (en producción) ya está afectando a los usuarios finales.

DiRT es bastante costoso de ejecutar y, por lo tanto, realizamos un ejercicio coordinado en una escala poco frecuente. Cuando creamos este nivel de interrupción, en realidad causamos dolor e impactamos negativamente en el desempeño de los empleados.

Si un probador realiza una acción mutable (escribir), modificará el estado de producción. Esto podría conducir al no determinismo y al fracaso de las afirmaciones, al fracaso de la capacidad de escribir en el futuro o a efectos secundarios visibles para el usuario.

Evaluación del usuario

Las pruebas de este tipo tienen las siguientes características:

- IVU: producción
- Datos: producción
- Verificación: manual y diferenciales A/B (de métricas)

Las pruebas basadas en producción permiten recopilar una gran cantidad de datos sobre el comportamiento del usuario. Tenemos algunas formas diferentes de recopilar métricas sobre la popularidad y los problemas con las próximas funciones, lo que nos brinda una alternativa a UAT:

dogfooding

Es posible usar implementaciones y experimentos limitados para hacer que las funciones en producción estén disponibles para un subconjunto de usuarios. Hacemos esto con nuestro propio personal a veces (comemos nuestro propio dogfood), y nos brindan comentarios valiosos en el entorno de implementación real.

Experimentación

Un nuevo comportamiento está disponible como experimento para un subconjunto de usuarios sin que lo sepan. Luego, el grupo experimental se compara con el grupo de control a nivel agregado en términos de alguna métrica deseada. Por ejemplo, en YouTube, nosotros

tuvo un experimento limitado que cambió la forma en que funcionaban los votos a favor de los videos (eliminando el voto a la baja), y solo una parte de la base de usuarios vio este cambio.

Esto es un enfoque enormemente importante para Google. Una de las primeras historias que escucha un usuario de Noogler al unirse a la empresa es sobre el momento en que Google lanzó un experimento que cambiaba el color de sombreado de fondo para los anuncios de AdWords en la Búsqueda de Google y notó un aumento significativo en los clics en los anuncios para los usuarios del grupo experimental en comparación con el grupo de control

Evaluación del evaluador

A los evaluadores humanos se les presentan los resultados de una operación determinada y eligen cuál es "mejor" y por qué. Esta retroalimentación se usa luego para determinar si un cambio dado es positivo, neutral o negativo. Por ejemplo, Google históricamente ha utilizado la evaluación de evaluadores para consultas de búsqueda (hemos publicado las pautas que brindamos a nuestros evaluadores). En algunos casos, la retroalimentación de estos datos de calificaciones puede ayudar a determinar el lanzamiento de cambios de algoritmo. La evaluación del calificador es fundamental para los sistemas no deterministas como los sistemas de aprendizaje automático para los que no hay una respuesta correcta clara, solo una noción de mejor o peor.

Pruebas grandes y el flujo de trabajo del desarrollador

Hemos hablado sobre qué son las pruebas grandes, por qué hacérselas, cuándo hacérselas y cuánto ha de hacerse, pero no hemos dicho mucho sobre quién. ¿Quién escribe las pruebas? ¿Quién ejecuta las pruebas e investiga las fallas? ¿Quién es el dueño de las pruebas? ¿Y cómo hacemos que esto sea tolerable?

Aunque es posible que no se aplique la infraestructura estándar de pruebas unitarias, aún es fundamental integrar pruebas más grandes en el flujo de trabajo del desarrollador. Una forma de hacerlo es garantizar que existan mecanismos automatizados para la ejecución previa y posterior al envío, incluso si estos son mecanismos diferentes a los de la prueba unitaria. En Google, muchas de estas pruebas grandes no pertenecen a TAP. No son herméticos, demasiado escamosos y/o requieren demasiados recursos. Pero aún debemos evitar que se rompan o, de lo contrario, no proporcionarán ninguna señal y se volverán demasiado difíciles de clasificar. Entonces, lo que hacemos es tener una compilación continua separada posterior al envío para estos. También animamos a ejecutar estas pruebas previas a la presentación, ya que proporciona retroalimentación directamente al autor.

Las pruebas de diferencias A/B que requieren la bendición manual de las diferencias también se pueden incorporar a dicho flujo de trabajo. Para el envío previo, puede ser un requisito de revisión de código aprobar cualquier diferencia en la interfaz de usuario antes de aprobar el cambio. Una de esas pruebas es que tenemos archivos que bloquean la publicación de errores automáticamente si el código se envía con diferencias no resueltas.

En algunos casos, las pruebas son tan grandes o dolorosas que la ejecución previa al envío agrega demasiada fricción al desarrollador. Estas pruebas aún se ejecutan después del envío y también se ejecutan como parte del proceso de lanzamiento. El inconveniente de no ejecutar estos presuntos es que la corrupción llega al monorepo y necesitamos identificar el cambio culpable para revertirlo. Pero nosotros

necesidad de hacer el equilibrio entre el dolor del desarrollador y la latencia de cambio incurrido y la confiabilidad de la compilación continua.

Creación de pruebas grandes

Aunque la estructura de las pruebas grandes es bastante estándar, todavía existe un desafío con la creación de una prueba de este tipo, especialmente si es la primera vez que alguien del equipo lo hace.

La mejor manera de hacer posible escribir tales pruebas es tener bibliotecas claras, documentación y ejemplos. Las pruebas unitarias son fáciles de escribir debido al soporte del idioma nativo (JUnit alguna vez fue esotérico pero ahora es la corriente principal). Reutilizamos estas bibliotecas de aserciones para pruebas de integración funcional, pero también hemos creado bibliotecas a lo largo del tiempo para interactuar con SUT, para ejecutar diferencias A/B, para sembrar datos de prueba y para orquestar flujos de trabajo de prueba.

Las pruebas más grandes son más costosas de mantener, tanto en recursos como en tiempo humano, pero no todas las pruebas grandes son iguales. Una de las razones por las que las pruebas de diferencias A/B son populares es que tienen un menor costo humano para mantener el paso de verificación. De igual forma, los COU de producción tienen un menor costo de mantenimiento que los COU herméticos aislados. Y debido a que toda esta infraestructura y código creados deben mantenerse, los ahorros de costos pueden agravarse.

Sin embargo, este costo debe ser visto de manera holística. Si el costo de reconciliar manualmente las diferencias o de respaldar y proteger las pruebas de producción supera los ahorros, se vuelve ineficaz.

Ejecución de pruebas grandes

Anteriormente mencionamos cómo nuestras pruebas más grandes no encajan en TAP y, por lo tanto, tenemos compilaciones continuas alternativas y preenvíos para ellas. Uno de los desafíos iniciales para nuestros ingenieros es cómo ejecutar pruebas no estándar y cómo iterar sobre ellas.

En la medida de lo posible, hemos tratado de hacer que nuestras pruebas más grandes se ejecuten de manera familiar para nuestros ingenieros. Nuestra infraestructura de envío previo pone una API común frente a la ejecución de estas pruebas y la ejecución de pruebas TAP, y nuestra infraestructura de revisión de código muestra ambos conjuntos de resultados. Pero muchas pruebas grandes están hechas a medida y, por lo tanto, necesitan documentación específica sobre cómo ejecutarlas bajo demanda. Esto puede ser una fuente de frustración para los ingenieros que no están familiarizados.

Acelerar las pruebas

Los ingenieros no esperan pruebas lentas. Cuanto más lenta sea una prueba, con menos frecuencia un ingeniero la ejecutará y más larga será la espera después de una falla hasta que vuelva a pasar.

La mejor manera de acelerar una prueba suele ser reducir su alcance o dividir una prueba grande en dos pruebas más pequeñas que se pueden ejecutar en paralelo. Pero hay algunos otros trucos que puede hacer para acelerar las pruebas más grandes.

Algunas pruebas ingenuas utilizarán la suspensión basada en el tiempo para esperar a que se produzca una acción no determinista, y esto es bastante común en las pruebas más grandes. Sin embargo, estas pruebas no tienen limitaciones de subprocessos y los usuarios de producción real desean esperar lo menos posible, por lo que es mejor que las pruebas reaccionen de la forma en que lo harían los usuarios de producción reales. Los enfoques incluyen lo siguiente:

- Sondeo de una transición de estado repetidamente durante una ventana de tiempo para que un evento se complete con una frecuencia más cercana a los microsegundos. Puede combinar esto con un valor de tiempo de espera en caso de que una prueba no alcance un estado estable.
- Implementación de un controlador de eventos.
- Suscripción a un sistema de notificación para la realización de un evento.

Tenga en cuenta que las pruebas que se basan en inactividad y tiempos de espera comenzarán a fallar cuando la flota que ejecuta esas pruebas se sobrecargue, lo que aumenta en espiral porque esas pruebas deben volver a ejecutarse con más frecuencia, lo que aumenta aún más la carga.

Menores tiempos de espera y retrasos del sistema interno

Un sistema de producción generalmente se configura asumiendo una topología de implementación distribuida, pero un SUT puede implementarse en una sola máquina (o al menos en un grupo de máquinas colocadas). Si hay tiempos de espera codificados o (especialmente) instrucciones de suspensión en el código de producción para tener en cuenta el retraso del sistema de producción, estos deben ajustarse y reducirse al ejecutar las pruebas.

Optimice el tiempo de compilación de la prueba

Una desventaja de nuestro monorepo es que todas las dependencias para una prueba grande se crean y proporcionan como entradas, pero esto puede no ser necesario para algunas pruebas más grandes. Si el SUT se compone de una parte central que es realmente el foco de la prueba y algunas otras dependencias binarias pares necesarias, podría ser posible usar versiones preconstruidas de esos otros binarios en una buena versión conocida. Nuestro sistema de compilación (basado en monorepo) no es compatible con este modelo fácilmente, pero el enfoque en realidad refleja más la producción en la que se lanzan diferentes servicios en diferentes versiones.

Eliminando la descamación

La descamación es lo suficientemente mala para las pruebas unitarias, pero para las pruebas más grandes, puede hacerlas inutilizables. Un equipo debe considerar la eliminación de la descamación de dichas pruebas como una alta prioridad. Pero, ¿cómo se puede eliminar la descamación de tales pruebas?

Minimizar la descamación comienza con la reducción del alcance de la prueba: un SUT hermético no correrá el riesgo de los tipos de descamación multiusuario y del mundo real de producción o un entorno de ensayo compartido, y un SUT hermético de una sola máquina no tendrá la red y Problemas de descamación del despliegue de un SUT distribuido. Pero puede mitigar otros problemas de descamación mediante el diseño y la implementación de pruebas y otras técnicas. En algunos casos, deberá equilibrarlos con la velocidad de prueba.

Así como hacer que las pruebas sean reactivas o basadas en eventos puede acelerarlas, también puede eliminar la descamación. Los tiempos de suspensión requieren mantenimiento de tiempo de espera, y estos tiempos de espera se pueden incrustar en el código de prueba. El aumento de los tiempos de espera internos del sistema puede reducir la irregularidad, mientras que la reducción de los tiempos de espera internos puede provocar irregularidades si el sistema se comporta de forma no determinista. La clave aquí es identificar una compensación que defina un comportamiento del sistema tolerable para los usuarios finales (por ejemplo, nuestro tiempo de espera máximo permitido es `nortesegundos`) pero maneja bien los comportamientos de ejecución de prueba inestables.

Un problema mayor con los tiempos de espera del sistema interno es que excederlos puede generar errores difíciles de clasificar. Un sistema de producción a menudo intentará limitar la exposición del usuario final a fallas catastróficas al manejar los posibles problemas internos del sistema con elegancia. Por ejemplo, si Google no puede publicar un anuncio en un límite de tiempo determinado, no devolvemos un 500, simplemente no publicamos un anuncio. Pero esto le parece a un corredor de prueba como si el código de publicación de anuncios pudiera romperse cuando solo hay un problema de tiempo de espera escamoso. Es importante hacer que el modo de falla sea obvio en este caso y facilitar el ajuste de dichos tiempos de espera internos para escenarios de prueba.

Hacer que las pruebas sean comprensibles

Un caso específico en el que puede ser difícil integrar pruebas en el flujo de trabajo del desarrollador es cuando esas pruebas producen resultados que son ininteligibles para el ingeniero que ejecuta las pruebas. Incluso las pruebas unitarias pueden producir cierta confusión: si mi cambio rompe su prueba, puede ser difícil entender por qué si generalmente no estoy familiarizado con su código, pero para pruebas más grandes, tal confusión puede ser insuperable. Las pruebas que son assertivas deben proporcionar una señal clara de aprobación/reprobación y deben proporcionar una salida de error significativa para ayudar a clasificar la fuente de la falla. Las pruebas que requieren investigación humana, como las pruebas de diferencias A/B, requieren un manejo especial para que sean significativas o corren el riesgo de omitirse durante el envío previo.

¿Cómo funciona esto en la práctica? Una buena prueba grande que falla debería hacer lo siguiente:

Tener un mensaje que identifique claramente cuál es la falla

El peor de los casos es tener un error que simplemente diga "Error en la afirmación" y un seguimiento de la pila. Un buen error anticipa la falta de familiaridad del corredor de prueba con el código y proporciona un mensaje que brinda contexto: "En `test>ReturnsOneFullPageOfSearchResultsForAPopularQuery`, esperaba 10 resultados de búsqueda pero obtuvo 1". Para una prueba de rendimiento o diferencia A/B que falla, debe haber una explicación clara en el resultado de lo que se mide y por qué el comportamiento se considera sospechoso.

Minimice el esfuerzo necesario para identificar la causa raíz de la discrepancia

Un seguimiento de pila no es útil para pruebas más grandes porque la cadena de llamadas puede abarcar múltiples límites de proceso. En cambio, es necesario producir un seguimiento a lo largo de la cadena de llamadas o invertir en automatización que pueda reducir al culpable. La prueba debería producir algún tipo de artefacto en este sentido. Por ejemplo, [Apuesto](#)es un marco utilizado por Google para asociar una sola ID de solicitud con todas las solicitudes en una cadena de llamadas RPC, y todos los registros asociados para esa solicitud se pueden correlacionar con esa ID para facilitar el seguimiento.

Proporcionar soporte e información de contacto.

Debería ser fácil para el corredor de la prueba obtener ayuda haciendo que los propietarios y los partidarios de la prueba sean fáciles de contactar.

Posesión de pruebas grandes

Las pruebas más grandes deben tener propietarios documentados: ingenieros que puedan revisar adecuadamente los cambios en la prueba y con quienes se pueda contar para brindar apoyo en caso de fallas en la prueba. Sin la propiedad adecuada, una prueba puede ser víctima de lo siguiente:

- Se vuelve más difícil para los contribuyentes modificar y actualizar la prueba
- Se tarda más en resolver los errores de prueba

Y la prueba se pudre.

Las pruebas de integración de componentes dentro de un proyecto en particular deben ser propiedad del líder del proyecto. Las pruebas centradas en funciones (pruebas que cubren una función comercial particular en un conjunto de servicios) deben ser propiedad de un "propietario de la función"; en algunos casos, este propietario puede ser un ingeniero de software responsable de la implementación de funciones de extremo a extremo; en otros casos, podría ser un gerente de producto o un "ingeniero de pruebas" que posee la descripción del escenario comercial. Quienquiera que sea el propietario de la prueba debe estar facultado para garantizar su salud general y debe tener tanto la capacidad de respaldar su mantenimiento como los incentivos para hacerlo.

Es posible crear una automatización en torno a los propietarios de las pruebas si esta información se registra de forma estructurada. Algunos enfoques que utilizamos incluyen los siguientes:

Propiedad de código regular

En muchos casos, una prueba más grande es un artefacto de código independiente que vive en una ubicación particular en nuestra base de código. En ese caso, podemos usar los PROPIETARIOS ([Capítulo 9](#)) información ya presente en el monorepo para insinuar a la automatización que los propietarios de una prueba en particular son los propietarios del código de prueba.

Anotaciones por prueba

En algunos casos, se pueden agregar varios métodos de prueba a una sola clase o módulo de prueba, y cada uno de estos métodos de prueba puede tener un propietario de función diferente. Usamos

anotaciones estructuradas por idioma para documentar al propietario de la prueba en cada uno de estos casos, de modo que si falla un método de prueba en particular, podamos identificar al propietario para contactarlo.

Conclusión

Un conjunto de pruebas completo requiere pruebas más grandes, tanto para garantizar que las pruebas coincidan con la fidelidad del sistema bajo prueba como para abordar problemas que las pruebas unitarias no pueden cubrir adecuadamente. Debido a que tales pruebas son necesariamente más complejas y más lentas de ejecutar, se debe tener cuidado para garantizar que las pruebas más grandes se posean correctamente, se mantengan bien y se ejecuten cuando sea necesario (como antes de las implementaciones en producción). En general, estas pruebas más grandes aún deben hacerse lo más pequeñas posible (sin dejar de mantener la fidelidad) para evitar la fricción del desarrollador. Para la mayoría de los proyectos de software es necesaria una estrategia de prueba integral que identifique los riesgos de un sistema y las pruebas más amplias que los aborden.

TL; DR

- Las pruebas más grandes cubren cosas que las pruebas unitarias no pueden.
- Las pruebas grandes se componen de un Sistema bajo prueba, Datos, Acción y Verificación.
- Un buen diseño incluye una estrategia de prueba que identifica los riesgos y pruebas más amplias que los mitigan.
- Se debe hacer un esfuerzo adicional con las pruebas más grandes para evitar que generen fricciones en el flujo de trabajo del desarrollador.

Deprecación

*Escrito por Hyrum Wright
Editado por Tom Mansreck*

Me encantan los plazos. Me gusta el sonido sibilante que hacen cuando pasan volando.

—Douglas Adams

Todos los sistemas envejecen. Aunque el software es un activo digital y los bits físicos en sí mismos no se degradan, las nuevas tecnologías, bibliotecas, técnicas, lenguajes y otros cambios ambientales a lo largo del tiempo vuelven obsoletos los sistemas existentes. Los sistemas antiguos requieren mantenimiento continuo, experiencia esotérica y, en general, más trabajo a medida que divergen del ecosistema circundante. A menudo, es mejor invertir esfuerzo en apagar los sistemas obsoletos, en lugar de dejar que sigan avanzando indefinidamente junto con los sistemas que los reemplazan. Pero la cantidad de sistemas obsoletos que siguen funcionando sugiere que, en la práctica, hacerlo no es trivial. Nos referimos al proceso de migración ordenada y eventual eliminación de sistemas obsoletos como *deprecación*.

La desaprobación es otro tema que pertenece con mayor precisión a la disciplina de la ingeniería de software que a la programación porque requiere pensar en cómo administrar un sistema a lo largo del tiempo. Para los ecosistemas de software de ejecución prolongada, la planificación y ejecución de la depreciación correctamente reduce los costos de los recursos y mejora la velocidad al eliminar la redundancia y la complejidad que se acumulan en un sistema con el tiempo. Por otro lado, los sistemas mal obsoletos pueden costar más que dejarlos solos. Si bien la desaprobación de los sistemas requiere un esfuerzo adicional, es posible planificar la desaprobación durante el diseño del sistema para que sea más fácil retirarlo y eliminarlo eventualmente. Las obsolescencias pueden afectar sistemas que van desde llamadas a funciones individuales hasta pilas de software completas. Para ser concretos, gran parte de lo que sigue se centra en las obsolescencias a nivel de código.

A diferencia de la mayoría de los otros temas que hemos discutido en este libro, Google todavía está aprendiendo cuál es la mejor manera de desaprobar y eliminar los sistemas de software. Este capítulo describe

las lecciones que aprendimos al desaprobar sistemas internos grandes y muy utilizados. A veces funciona como se espera y otras veces no, pero el problema general de eliminar los sistemas obsoletos sigue siendo una preocupación difícil y cambiante en la industria.

Este capítulo trata principalmente de la desaprobación de sistemas técnicos, no de productos de usuario final. La distinción es algo arbitraria dado que una API externa es solo otro tipo de producto, y una API interna puede tener consumidores que se consideran usuarios finales. Aunque muchos de los principios se aplican al rechazo de un producto público, aquí nos ocupamos de los aspectos técnicos y de política de desaprobar y eliminar sistemas obsoletos donde el propietario del sistema tiene visibilidad sobre su uso.

¿Por qué desaprobar?

Nuestra discusión sobre la desaprobación comienza con la premisa fundamental de que *el código es un pasivo, no un activo*. Después de todo, si el código fuera un activo, ¿por qué deberíamos siquiera molestarlos en perder tiempo tratando de rechazar y eliminar sistemas obsoletos? El código tiene costos, algunos de los cuales se soportan en el proceso de creación de un sistema, pero muchos otros costos se soportan a medida que se mantiene un sistema a lo largo de su vida útil. Estos costos continuos, como los recursos operativos necesarios para mantener un sistema en funcionamiento o el esfuerzo de actualizar continuamente su base de código a medida que evolucionan los ecosistemas circundantes, significan que vale la pena evaluar las compensaciones entre mantener en funcionamiento un sistema obsoleto o trabajar para convertirlo abajo.

La edad de un sistema por sí sola no justifica su desaprobación. Un sistema podría diseñarse finamente durante varios años para ser el epítome de la forma y función del software. Algunos sistemas de software, como el sistema de composición tipográfica LaTeX, se han mejorado a lo largo de décadas y, aunque todavía se producen cambios, son pocos y esporádicos. El hecho de que algo sea viejo no significa que esté obsoleto.

La desaprobación es más adecuada para los sistemas que son obsoletos demostrablemente y existe un reemplazo que proporciona una funcionalidad comparable. El nuevo sistema podría usar los recursos de manera más eficiente, tener mejores propiedades de seguridad, construirse de una manera más sostenible o simplemente corregir errores. Tener dos sistemas para lograr lo mismo puede no parecer un problema apremiante, pero con el tiempo, los costos de mantenimiento de ambos pueden aumentar sustancialmente. Los usuarios pueden necesitar usar el nuevo sistema, pero aún tienen dependencias que usan el obsoleto.

Es posible que los dos sistemas necesiten interactuar entre sí, lo que requiere un código de transformación complicado. A medida que ambos sistemas evolucionan, pueden llegar a depender el uno del otro, lo que dificulta la eventual eliminación de cualquiera de ellos. A la larga, hemos descubierto que tener varios sistemas que realizan la misma función también impide la evolución del sistema más nuevo porque todavía se espera que mantenga la compatibilidad.

con el viejo Gastar el esfuerzo de eliminar el sistema anterior puede dar sus frutos, ya que el sistema de reemplazo ahora puede evolucionar más rápidamente.

Anteriormente hicimos la afirmación de que "el código es un pasivo, no un activo". Si eso es cierto, ¿por qué hemos pasado la mayor parte de este libro discutiendo la forma más eficiente de construir sistemas de software que puedan vivir durante décadas? ¿Por qué poner todo ese esfuerzo en crear más código cuando simplemente terminará en el lado del pasivo del balance?

Códigos *sí mismo* aporta valor: es la *funcionalidad* que proporciona que aporta valor. Esta funcionalidad es un activo si satisface una necesidad del usuario: el código que implementa esta funcionalidad es simplemente un medio para ese fin. Si pudiéramos obtener la misma funcionalidad de una sola línea de código comprensible y mantenible que 10,000 líneas de código de espagueti intrincado, preferiríamos lo primero. El código en sí tiene un costo: cuanto más simple sea el código, manteniendo la misma cantidad de funcionalidad, mejor.

En lugar de centrarnos en cuánto código podemos producir o qué tan grande es nuestra base de código, deberíamos centrarnos en cuánta funcionalidad puede ofrecer por unidad de código y tratar de maximizar esa métrica. Una de las formas más sencillas de hacerlo es no escribir más código y esperar obtener más funcionalidad; está eliminando el exceso de código y los sistemas que ya no se necesitan. Las políticas y los procedimientos de desaprobación lo hacen posible.

Aunque la desaprobación es útil, hemos aprendido en Google que las organizaciones tienen un límite en la cantidad de trabajo de desaprobación que es razonable realizar simultáneamente, tanto desde el punto de vista de los equipos que realizan la desaprobación como de los clientes de esos equipos. Por ejemplo, aunque todo el mundo agradece tener carreteras recién pavimentadas, si el departamento de obras públicas decidiera cerrar *todos* camino para pavimentar simultáneamente, nadie iría a ninguna parte. Al concentrar sus esfuerzos, las cuadrillas de pavimentación pueden realizar trabajos específicos más rápido y al mismo tiempo permitir que el resto del tráfico progrese. Del mismo modo, es importante elegir los proyectos de obsolescencia con cuidado y luego comprometerse a seguir hasta terminarlos.

¿Por qué es tan difícil la desaprobación?

Hemos mencionado la Ley de Hyrum en otras partes de este libro, pero vale la pena repetir su aplicabilidad aquí: cuantos más usuarios de un sistema, mayor será la probabilidad de que los usuarios lo utilicen de formas inesperadas e imprevistas, y más difícil será desaprobarlo y eliminar tal sistema. Su uso simplemente "sucede que funciona" en lugar de estar "garantizado para funcionar". En este contexto, la eliminación de un sistema se puede considerar como el cambio definitivo: no solo estamos cambiando el comportamiento, ¡estamos eliminando ese comportamiento por completo! Este tipo de alteración radical liberará a una serie de dependientes inesperados.

Para complicar aún más las cosas, la desaprobación generalmente no es una opción hasta que esté disponible un sistema más nuevo que proporcione la misma (o mejor) funcionalidad. el nuevo sistema

podría ser mejor, pero también es diferente: después de todo, si fuera exactamente igual al sistema obsoleto, no proporcionaría ningún beneficio a los usuarios que migran a él (aunque sí podría beneficiar al equipo que lo opera). Esta diferencia funcional significa que una coincidencia uno a uno entre el sistema antiguo y el nuevo sistema es rara, y cada uso del sistema antiguo debe evaluarse en el contexto del nuevo.

Otra reticencia sorprendente a desaprobar es el apego emocional a los viejos sistemas, particularmente aquellos en los que el desaprobador ayudó a crear. Un ejemplo de esta aversión al cambio ocurre cuando eliminamos sistemáticamente el código antiguo en Google: ocasionalmente hemos encontrado resistencia del tipo "¡Me gusta este código!" Puede ser difícil convencer a los ingenieros para que derriben algo que han estado construyendo durante años. Esta es una respuesta comprensible, pero en última instancia contraproducente: si un sistema es obsoleto, tiene un costo neto para la organización y debe eliminarse. Una de las formas en que hemos abordado las preocupaciones sobre mantener el código antiguo en Google es asegurarnos de que el repositorio del código fuente no solo se pueda buscar en el baúl, sino también históricamente. Incluso el código que se ha eliminado se puede volver a encontrar (ver[capítulo 17](#)).

Hay un viejo chiste dentro de Google que dice que hay dos formas de hacer las cosas: la que está en desuso y la que aún no está lista. Por lo general, esto es el resultado de una nueva solución que está "casi" terminada y es la desafortunada realidad de trabajar en un entorno tecnológico que es complejo y acelerado.

Los ingenieros de Google se han acostumbrado a trabajar en este entorno, pero todavía puede resultar desconcertante. La buena documentación, muchas señales y los equipos de expertos que ayudan con el proceso de desuso y migración hacen que sea más fácil saber si debe usar lo antiguo, con todas sus verrugas, o lo nuevo, con todas sus incertidumbres.

Finalmente, financiar y ejecutar esfuerzos de desaprobación puede ser políticamente difícil; dotar de personal a un equipo y dedicar tiempo a eliminar sistemas obsoletos cuesta dinero real, mientras que los costos de no hacer nada y dejar que el sistema funcione sin supervisión no son fácilmente observables. Puede ser difícil convencer a las partes interesadas relevantes de que los esfuerzos de desaprobación valen la pena, especialmente si tienen un impacto negativo en el desarrollo de nuevas funciones. Las técnicas de investigación, como las descritas en[Capítulo 7](#), puede proporcionar evidencia concreta de que vale la pena desaprobar.

Dada la dificultad de desaprobar y eliminar sistemas de software obsoletos, a menudo es más fácil para los usuarios desarrollar un sistema *en el lugar*, en lugar de reemplazarlo por completo. La incrementalidad no evita el proceso de desaprobación por completo, pero lo divide en partes más pequeñas y manejables que pueden generar beneficios incrementales. Dentro de Google, hemos observado que migrar a sistemas completamente nuevos es *extremadamente* costoso, y los costos son frecuentemente subestimados. Esfuerzos de desaprobación incrementales

logrado mediante la refactorización en el lugar puede mantener los sistemas existentes en funcionamiento al mismo tiempo que facilita la entrega de valor a los usuarios.

Desaprobación durante el diseño

Al igual que muchas actividades de ingeniería, la desaprobación de un sistema de software se puede planificar a medida que se construyen esos sistemas por primera vez. Las opciones de lenguaje de programación, arquitectura de software, composición del equipo e incluso la política y la cultura de la empresa tienen un impacto en la facilidad con la que eventualmente se eliminará un sistema una vez que haya llegado al final de su vida útil.

El concepto de diseñar sistemas para que eventualmente puedan quedar obsoletos puede ser radical en la ingeniería de software, pero es común en otras disciplinas de la ingeniería. Considere el ejemplo de una planta de energía nuclear, que es una pieza de ingeniería extremadamente compleja. Como parte del diseño de una central nuclear, debe tenerse en cuenta su eventual desmantelamiento después de una vida de servicio productivo, llegando incluso a destinar fondos para este propósito.¹ Muchas de las opciones de diseño en la construcción de una planta de energía nuclear se ven afectadas cuando los ingenieros saben que eventualmente será necesario desmantelarla.

Desafortunadamente, los sistemas de software rara vez están tan cuidadosamente diseñados. Muchos ingenieros de software se sienten atraídos por la tarea de construir y lanzar nuevos sistemas, no por mantener los existentes. La cultura corporativa de muchas empresas, incluida Google, enfatiza la creación y el envío rápido de nuevos productos, lo que a menudo desincentiva el diseño teniendo en cuenta la obsolescencia desde el principio. Y a pesar de la noción popular de los ingenieros de software como autómatas impulsados por datos, puede ser psicológicamente difícil planificar la eventual desaparición de las creaciones en las que estamos trabajando tan duro para construir.

Entonces, ¿en qué tipo de consideraciones deberíamos pensar al diseñar sistemas que podamos desechar más fácilmente en el futuro? Estas son algunas de las preguntas que animamos a los equipos de ingeniería de Google a hacer:

- ¿Qué tan fácil será para mis consumidores migrar de mi producto a un reemplazo potencial?
- ¿Cómo puedo reemplazar partes de mi sistema de forma incremental?

Muchas de estas preguntas se relacionan con la forma en que un sistema proporciona y consume dependencias. Para obtener una discusión más detallada sobre cómo administramos estas dependencias, consulte [capítulo 16](#).

¹"Diseño y Construcción de Centrales Nucleares para Facilitar el Desmantelamiento", Serie de Informes Técnicos No. 382, OIEA, Viena (1997).

Finalmente, debemos señalar que la decisión de apoyar un proyecto a largo plazo se toma cuando una organización decide por primera vez construir el proyecto. Una vez que existe un sistema de software, las únicas opciones que quedan son admitirlo, desaprobarlo cuidadosamente o dejar que deje de funcionar cuando algún evento externo provoque que se rompa. Todas estas son opciones válidas, y las compensaciones entre ellas serán específicas de la organización. Una nueva startup con un solo proyecto la matará sin contemplaciones cuando la empresa quiebre, pero una gran empresa deberá pensar más detenidamente sobre el impacto en su cartera y reputación al considerar la eliminación de proyectos antiguos. Como se mencionó anteriormente, Google todavía está aprendiendo cuál es la mejor manera de hacer estas concesiones con nuestros propios productos internos y externos.

En resumen, no inicie proyectos que su organización no se comprometa a apoyar durante la vida útil esperada de la organización. Incluso si la organización decide desaprobar y eliminar el proyecto, aún habrá costos, pero se pueden mitigar mediante la planificación y las inversiones en herramientas y políticas.

Tipos de depreciación

El desuso no es un solo tipo de proceso, sino un continuo de ellos, que van desde "lo apagaremos algún día, esperamos" hasta "este sistema desaparecerá mañana, es mejor que los clientes estén preparados para eso". En términos generales, dividimos este continuo en dos áreas separadas: consultiva y obligatoria.

Depreciación de aviso

Consultivas bajas son aquellas que no tienen una fecha límite y no son de alta prioridad para la organización (y para las cuales la empresa no está dispuesta a dedicar recursos). Estos también podrían etiquetarse como *aspiracional/obsolescencia*: el equipo sabe que el sistema ha sido reemplazado y, aunque esperan que los clientes eventualmente migren al nuevo sistema, no tienen planes inminentes para brindar soporte para ayudar a mover clientes o eliminar el sistema anterior. Este tipo de desaprobación a menudo carece de cumplimiento: esperamos que los clientes se muden, pero no podemos obligarlos a hacerlo. Como le dirán nuestros amigos de la SRE: "La esperanza no es una estrategia".

Las obsolescencias de avisos son una buena herramienta para anunciar la existencia de un nuevo sistema y animar a los primeros usuarios a empezar a probarlo. Este nuevo sistema debería *no ser* considerado en un período beta: debe estar listo para usos y cargas de producción y debe estar preparado para soportar nuevos usuarios indefinidamente. Por supuesto, cualquier sistema nuevo experimentará dolores de crecimiento, pero después de que el antiguo sistema haya quedado obsoleto de alguna manera, el nuevo sistema se convertirá en una pieza fundamental de la infraestructura de la organización.

Un escenario que hemos visto en Google en el que las obsolescencias de avisos tienen fuertes beneficios es cuando el nuevo sistema ofrece beneficios convincentes a sus usuarios. En estos casos,

simplemente notificar a los usuarios de este nuevo sistema y proporcionarles herramientas de autoservicio para migrar a él, a menudo fomenta la adopción. Sin embargo, los beneficios no pueden ser simplemente incrementales: deben ser transformadores. Los usuarios dudarán en migrar por su cuenta para obtener beneficios marginales, e incluso los sistemas nuevos con grandes mejoras no obtendrán una adopción completa utilizando solo esfuerzos de desaprobación de asesoramiento.

La desaprobación del aviso permite a los autores del sistema empujar a los usuarios en la dirección deseada, pero no se debe contar con ellos para realizar la mayor parte del trabajo de migración. A menudo es tentador simplemente poner una advertencia de obsolescencia en un sistema antiguo y marcharse sin ningún esfuerzo adicional. Nuestra experiencia en Google ha sido que esto puede conducir a (ligeramente) menos usos nuevos de un sistema obsoleto, pero rara vez lleva a que los equipos migren activamente fuera de él. Los usos existentes del antiguo sistema ejercen una especie de atracción conceptual (o técnica): comparativamente, muchos usos del antiguo sistema tenderán a recoger una gran parte de los nuevos usos, sin importar cuánto digamos: "Por favor, utilice el nuevo sistema." El antiguo sistema seguirá requiriendo mantenimiento y otros recursos a menos que se aliente más activamente a sus usuarios a migrar.

Desaprobación obligatoria

Este estímulo activo viene en forma de *obligatorio* deprecación. Este tipo de desaprobación generalmente viene con una fecha límite para la eliminación del sistema obsoleto: si los usuarios continúan dependiendo de él más allá de esa fecha, encontrarán que sus propios sistemas ya no funcionan.

Contrariamente a la intuición, la mejor manera de escalar los esfuerzos de desuso obligatorio es ubicar la experiencia de los usuarios que migran dentro de un solo equipo de expertos, generalmente el equipo responsable de eliminar el sistema anterior por completo. Este equipo tiene incentivos para ayudar a otros a migrar del sistema obsoleto y puede desarrollar experiencia y herramientas que luego se pueden usar en toda la organización. Muchas de estas migraciones se pueden efectuar usando las mismas herramientas discutidas en [capítulo 22](#).

Para que la desaprobación obligatoria realmente funcione, su cronograma debe tener un mecanismo de cumplimiento. Esto no implica que el cronograma no pueda cambiar, sino que facilita al equipo que ejecuta el proceso de desaprobación para interrumpir a los usuarios que no cumplen después de que hayan sido suficientemente advertidos a través de los esfuerzos para migrarlos. Sin este poder, es fácil que los equipos de clientes ignoren el trabajo de desaprobación a favor de características u otro trabajo más apremiante.

Al mismo tiempo, las desactivaciones obligatorias sin personal para hacer el trabajo pueden parecer mezquinas para los equipos de los clientes, lo que generalmente impide completar la desactivación. Los clientes simplemente ven ese trabajo de desaprobación como un mandato sin fondos, que les obliga a dejar de lado sus propias prioridades para trabajar solo para mantener sus servicios en funcionamiento. Esto se parece mucho al fenómeno de "correr para permanecer en el lugar" y crea fricciones entre los encargados del mantenimiento de la infraestructura y sus clientes. es por esto

razón por la que recomendamos encarecidamente que las depreciaciones obligatorias sean atendidas activamente por un equipo especializado hasta su finalización.

También vale la pena señalar que incluso con la fuerza de la política detrás de ellos, las depreciaciones obligatorias aún pueden enfrentar obstáculos políticos. Imagínese intentar hacer cumplir un esfuerzo de desaprobación obligatorio cuando el último usuario que queda del sistema anterior es una pieza crítica de la infraestructura de la que depende toda su organización. ¿Qué tan dispuesto estaría a romper esa infraestructura, y, transitivamente, a todos los que dependen de ella, solo por el hecho de establecer una fecha límite arbitraria? Es difícil creer que la desaprobación sea realmente obligatoria si ese equipo puede vetar su progreso.

El repositorio monolítico y el gráfico de dependencia de Google nos brindan una gran perspectiva de cómo se utilizan los sistemas en nuestro ecosistema. Aun así, es posible que algunos equipos ni siquiera sepan que dependen de un sistema obsoleto, y puede ser difícil descubrir estas dependencias de forma analítica. También es posible encontrarlos dinámicamente a través de pruebas de frecuencia y duración crecientes durante las cuales el antiguo sistema se apaga temporalmente. Estos cambios intencionales brindan un mecanismo para descubrir dependencias no deseadas al ver qué falla, lo que alerta a los equipos sobre la necesidad de prepararse para la próxima fecha límite. Dentro de Google, ocasionalmente cambiamos el nombre de los símbolos de solo implementación para ver qué usuarios dependen de ellos sin darse cuenta.

Con frecuencia en Google, cuando un sistema está programado para su desuso y eliminación, el equipo anuncia interrupciones planificadas de mayor duración en los meses y semanas anteriores al cierre. De manera similar a los ejercicios de prueba de recuperación ante desastres (DiRT) de Google, estos eventos a menudo descubren dependencias desconocidas entre los sistemas en ejecución. Este enfoque incremental permite que esos equipos dependientes descubran y luego planifiquen la eventual eliminación del sistema, o incluso trabajen con el equipo obsoleto para ajustar su cronograma. (Los mismos principios también se aplican a las dependencias de código estático, pero la información semántica proporcionada por las herramientas de análisis estático suele ser suficiente para detectar todas las dependencias del sistema obsoleto).

Advertencias de obsolescencia

Tanto para las desaprobaciones obligatorias como las de asesoramiento, a menudo es útil tener una forma programática de marcar los sistemas como obsoletos para que los usuarios sean advertidos sobre su uso y se animen a retirarse. A menudo es tentador marcar algo como obsoleto y esperar que sus usos eventualmente desaparezcan, pero recuerda: "la esperanza no es una estrategia". Las advertencias de obsolescencia pueden ayudar a evitar nuevos usos, pero rara vez conducen a la migración de sistemas existentes.

Lo que suele ocurrir en la práctica es que estos avisos se acumulan con el tiempo. Si se utilizan en un contexto transitivo (por ejemplo, la biblioteca A depende de la biblioteca B, que depende de la biblioteca C, y C emite una advertencia, que aparece cuando se construye A), estas advertencias pronto pueden abrumar a los usuarios de un sistema para el punto en que los ignoran por completo. En el cuidado de la salud, este fenómeno se conoce como "**Alerta de fatiga**".

Cualquier advertencia de obsolescencia emitida a un usuario debe tener dos propiedades: capacidad de acción y relevancia. Una advertencia es procesable si el usuario puede usar la advertencia para realizar alguna acción relevante, no solo en teoría, sino en términos prácticos, dada la experiencia en esa área problemática que esperamos para un ingeniero promedio. Por ejemplo, una herramienta puede advertir que una llamada a una función determinada debe reemplazarse con una llamada a su contraparte actualizada, o un correo electrónico puede describir los pasos necesarios para mover datos de un sistema antiguo a uno nuevo. En cada caso, la advertencia proporcionó los siguientes pasos que un ingeniero puede realizar para dejar de depender del sistema obsoleto.²

Una advertencia puede ser procesable, pero aun así ser molesta. Para ser útil, una advertencia de desaprobación también debe ser *importante*. Una advertencia es relevante si aparece en un momento en que un usuario realmente realiza la acción indicada. Es mejor advertir sobre el uso de una función obsoleta mientras el ingeniero está escribiendo el código que usa esa función, no después de que se haya registrado en el repositorio durante varias semanas. Del mismo modo, es mejor enviar un correo electrónico para la migración de datos varios meses antes de que se elimine el sistema antiguo en lugar de una ocurrencia tardía un fin de semana antes de que se produzca la eliminación.

Es importante resistir la tentación de poner advertencias de desaprobación en todo lo posible. Las advertencias en sí mismas no son malas, pero las herramientas ingenuas a menudo producen una cantidad de mensajes de advertencia que pueden abrumar al ingeniero desprevenido. Dentro de Google, somos muy liberales al marcar funciones antiguas como obsoletas, pero aprovechamos herramientas como [Propenso a errores](#) clang-tidy para garantizar que las advertencias se muestren de manera específica. Como se discutió en [capítulo 20](#), limitamos estas advertencias a las líneas modificadas recientemente como una forma de advertir a las personas sobre los nuevos usos del símbolo obsoleto. Se agregan advertencias mucho más intrusivas, como los objetivos obsoletos en el gráfico de dependencia, solo para las obsolescencias obligatorias, y el equipo está alejando activamente a los usuarios. En cualquier caso, las herramientas desempeñan un papel importante en la presentación de la información adecuada a las personas adecuadas en el momento adecuado, lo que permite agregar más advertencias sin fatigar al usuario.

Gestión del proceso de desaprobación

Aunque pueden sentirse como diferentes tipos de proyectos porque estamos deconstruyendo un sistema en lugar de construirlo, los proyectos de desaprobación son similares a otros proyectos de ingeniería de software en la forma en que se administran y ejecutan. No dedicaremos demasiado esfuerzo a analizar las similitudes entre esos esfuerzos de gestión, pero vale la pena señalar las formas en que difieren.

² Ver <https://abseil.io/docs/cpp/tools/api-upgrades> para un ejemplo.

Propietarios de procesos

Hemos aprendido en Google que, sin propietarios explícitos, es poco probable que un proceso de desaprobación logre un progreso significativo, sin importar cuántas advertencias y alertas pueda generar un sistema. Tener propietarios de proyectos explícitos que tengan la tarea de administrar y ejecutar el proceso de desuso puede parecer un mal uso de los recursos, pero las alternativas son aún peores: nunca desactive nada ni delegue los esfuerzos de desuso a los usuarios del sistema. El segundo caso se convierte simplemente en una desaprobación consultiva, que nunca terminará orgánicamente, y el primero es un compromiso de mantener todo sistema antiguo hasta el infinito. Centralizar los esfuerzos de desaprobación ayuda a asegurar mejor que la experiencia realmente *reducecostes* haciéndolos más transparentes.

Los proyectos abandonados a menudo presentan un problema al establecer la propiedad y alinear los incentivos. Toda organización de tamaño razonable tiene proyectos que todavía se utilizan activamente pero que nadie posee ni mantiene claramente, y Google no es una excepción. Los proyectos a veces entran en este estado porque están obsoletos: los propietarios originales se han trasladado a un proyecto sucesor, dejando el obsoleto en el sótano, todavía una dependencia de un proyecto crítico, y con la esperanza de que se desvanezca con el tiempo.

Es poco probable que tales proyectos desaparezcan por sí solos. A pesar de nuestras mejores esperanzas, hemos descubierto que estos proyectos aún requieren expertos en desaprobación para eliminarlos y evitar que fallen en momentos inoportunos. Estos equipos deben tener la eliminación como su objetivo principal, no solo como un proyecto secundario de algún otro trabajo. En el caso de prioridades contrapuestas, el trabajo de desaprobación casi siempre se percibirá como de menor prioridad y rara vez recibirá la atención que necesita. Este tipo de tareas de limpieza importantes, no urgentes, son un gran uso del 20% del tiempo y brindan a los ingenieros exposición a otras partes de la base de código.

Hitos

Al construir un nuevo sistema, los hitos del proyecto generalmente son bastante claros: "Lanzar las funciones de frobnazzer para el próximo trimestre". Siguiendo las prácticas de desarrollo incremental, los equipos construyen y entregan la funcionalidad de forma incremental a los usuarios, quienes obtienen una victoria cada vez que aprovechan una nueva característica. El objetivo final puede ser lanzar todo el sistema, pero los hitos incrementales ayudan a dar al equipo una sensación de progreso y garantizan que no tengan que esperar hasta el final del proceso para generar valor para la organización.

Por el contrario, a menudo se puede sentir que el único hito de un proceso de desaprobación es eliminar por completo el sistema obsoleto. El equipo puede sentir que no ha hecho ningún progreso hasta que haya apagado las luces y se haya ido a casa. Aunque este puede ser el paso más significativo para el equipo, si ha hecho su trabajo correctamente, a menudo es el que menos se nota por parte de cualquier persona externa al equipo, porque en ese momento, el sistema obsoleto ya no tiene usuarios. Los gerentes de proyecto de desaprobación deben resistir la tentación de

haga de este el único hito medible, particularmente dado que es posible que ni siquiera suceda en todos los proyectos de desaprobación.

De manera similar a la creación de un nuevo sistema, la gestión de un equipo que trabaja en el desuso debe implicar hitos incrementales concretos, que sean medibles y brinden valor a los usuarios. Las métricas utilizadas para evaluar el progreso de la depreciación serán diferentes, pero sigue siendo bueno para la moral celebrar los logros incrementales en el proceso de depreciación. Hemos encontrado útil reconocer los hitos incrementales apropiados, como eliminar un subcomponente clave, tal como reconoceríamos los logros en la construcción de un nuevo producto.

Herramientas de desaprobación

Gran parte de las herramientas utilizadas para administrar el proceso de desaprobación se analiza en profundidad en otras partes de este libro, como el proceso de cambio a gran escala (LSC) ([capítulo 22](#)) o nuestras herramientas de revisión de código ([capítulo 19](#)). En lugar de hablar sobre los detalles de las herramientas, describiremos brevemente cómo esas herramientas son útiles al administrar la desaprobación de un sistema obsoleto. Estas herramientas se pueden categorizar como herramientas de detección, migración y prevención de reincidencia.

Descubrimiento

Durante las primeras etapas de un proceso de desaprobación, y de hecho durante todo el proceso, es útil saber *cómo* y *por quién* se está utilizando un sistema obsoleto. Gran parte del trabajo inicial de desaprobación consiste en determinar quién está usando el sistema anterior y de qué maneras imprevistas. Dependiendo de los tipos de uso, este proceso puede requerir revisar la decisión de desaprobación una vez que se obtenga nueva información. También usamos estas herramientas a lo largo del proceso de desaprobación para comprender cómo progresa el esfuerzo.

Dentro de Google, usamos herramientas como Code Search (ver [capítulo 17](#)) y Kythe (ver [Capítulo 23](#)) para determinar estáticamente qué clientes usan una biblioteca determinada y, a menudo, para probar el uso existente para ver de qué tipo de comportamientos dependen inesperadamente los clientes. Debido a que las dependencias de tiempo de ejecución generalmente requieren el uso de una biblioteca estática o un cliente ligero, esta técnica proporciona gran parte de la información necesaria para iniciar y ejecutar un proceso de desaprobación. El registro y el muestreo del tiempo de ejecución en producción ayudan a descubrir problemas con las dependencias dinámicas.

Finalmente, tratamos nuestro conjunto de pruebas globales como un oráculo para determinar si se han eliminado todas las referencias a un símbolo antiguo. Como se discutió en [Capítulo 11](#), las pruebas son un mecanismo para prevenir cambios de comportamiento no deseados en un sistema a medida que el ecosistema evoluciona. La obsolescencia es una gran parte de esa evolución, y los clientes son responsables de tener suficientes pruebas para garantizar que la eliminación de un sistema obsoleto no los perjudique.

Migración

Gran parte del trabajo de los esfuerzos de desaprobación en Google se logra mediante el uso del mismo conjunto de herramientas de generación y revisión de código que mencionamos anteriormente. El proceso y las herramientas de LSC son particularmente útiles para administrar el gran esfuerzo de actualizar la base de código para hacer referencia a nuevas bibliotecas o servicios de tiempo de ejecución.

Prevención de la reincidencia

Finalmente, una parte de la infraestructura de desaprobación que a menudo se pasa por alto son las herramientas para evitar la adición de nuevos usos de la misma cosa que se está eliminando activamente. Incluso para desaprobaciones de asesoramiento, es útil advertir a los usuarios que eviten un sistema obsoleto en favor de uno nuevo cuando estén escribiendo código nuevo. Sin la prevención de reincidencias, la desactivación puede convertirse en un juego de golpear un topo en el que los usuarios agregan constantemente nuevos usos de un sistema con el que están familiarizados (o encuentran ejemplos de otros lugares en el código base), y el equipo de desactivación migra constantemente estos nuevos usos. . Este proceso es contraproducente y desmoralizador.

Para evitar que la obsolescencia retroceda a nivel micro, utilizamos el marco de análisis estático Tricorder para notificar a los usuarios que están agregando llamadas a un sistema obsoleto y brindarles comentarios sobre el reemplazo apropiado. Los propietarios de sistemas obsoletos pueden agregar anotaciones del compilador a los símbolos obsoletos (como el @obsoletoanotación de Java), y Tricorder muestra nuevos usos de estos símbolos en el momento de la revisión. Estas anotaciones otorgan control sobre los mensajes a los equipos que poseen el sistema obsoleto y, al mismo tiempo, alertan automáticamente al autor del cambio. En casos limitados, las herramientas también sugieren una corrección de botón para migrar al reemplazo sugerido.

En un nivel macro, usamos listas blancas de visibilidad en nuestro sistema de compilación para garantizar que no se introduzcan nuevas dependencias en el sistema obsoleto. Las herramientas automatizadas examinan periódicamente estas listas blancas y las eliminan a medida que los sistemas dependientes se migran fuera del sistema obsoleto.

Conclusión

La desaprobación puede parecer el trabajo sucio de limpiar la calle después de que el desfile del circo acaba de pasar por la ciudad, sin embargo, estos esfuerzos mejoran el ecosistema de software en general al reducir los gastos generales de mantenimiento y la carga cognitiva de los ingenieros. El mantenimiento escalable de sistemas de software complejos a lo largo del tiempo es más que solo crear y ejecutar software: también debemos poder eliminar sistemas que están obsoletos o que no se usan.

Un proceso completo de desaprobación implica gestionar con éxito los desafíos sociales y técnicos a través de políticas y herramientas. La desaprobación de manera organizada y bien administrada a menudo se pasa por alto como una fuente de beneficios para una organización, pero es esencial para su sostenibilidad a largo plazo.

TL; DR

- Los sistemas de software tienen costos continuos de mantenimiento que deben sopesarse frente a los costos de eliminarlos.
- Quitar cosas suele ser más difícil que construirlas porque los usuarios existentes a menudo usan el sistema más allá de su diseño original.
- La evolución de un sistema en el lugar suele ser más barato que reemplazarlo por uno nuevo, cuando se incluyen los costos de desconexión.
- Es difícil evaluar honestamente los costos involucrados en la decisión de desaprobar: además de los costos directos de mantenimiento involucrados en mantener el sistema anterior, hay costos de ecosistema involucrados en tener múltiples sistemas similares para elegir y que podrían necesitar interoperar. El antiguo sistema podría ser implícitamente un lastre para el desarrollo de funciones para el nuevo. Estos costos del ecosistema son difusos y difíciles de medir. Los costos de depreciación y eliminación a menudo son igualmente difusos.

PARTE IV

Herramientas

Control de Versiones y Gestión de Sucursales

*Escrito por Titus Winters
Editado por Lisa Carey*

Quizás ninguna herramienta de ingeniería de software se adopte tan universalmente en la industria como el control de versiones. Difícilmente se puede imaginar una organización de software más grande que unas pocas personas que no dependa de un Sistema de control de versiones (VCS) formal para administrar su código fuente y coordinar actividades entre ingenieros.

En este capítulo, veremos por qué el uso del control de versiones se ha convertido en una norma tan inequívoca en la ingeniería de software, y describiremos los diversos enfoques posibles para el control de versiones y la administración de sucursales, incluido cómo lo hacemos a escala en todos de Google. También examinaremos los pros y los contras de varios enfoques; aunque creemos que todo el mundo debería utilizar el control de versiones, algunas políticas y procesos de control de versiones pueden funcionar mejor para su organización (o en general) que otros. En particular, encontramos el "desarrollo basado en troncales" popularizado por DevOps¹(un repositorio, sin ramas de desarrollo) para ser un enfoque de política particularmente escalable, y proporcionaremos algunas sugerencias sobre por qué.

¿Qué es el control de versiones?



Esta sección puede resultar un poco básica para muchos lectores: el uso del control de versiones es, después de todo, bastante omnipresente. Si desea saltar adelante, le sugerimos saltar a la sección "[Fuente de la verdad](#)" en la página 334.

1 The DevOps Research Association, que fue adquirida por Google entre el primer borrador de este capítulo y ha publicado extensamente sobre esto en el informe anual "State of DevOps Report" y en el libro *Acelerar*. Por lo que sabemos, popularizó la terminología *desarrollo basado en troncos*.

Un VCS es un sistema que realiza un seguimiento de las revisiones (versiones) de los archivos a lo largo del tiempo. Un VCS mantiene algunos metadatos sobre el conjunto de archivos que se administran y, en conjunto, una copia de los archivos y los metadatos se denomina repositorio.²(repo para abreviar). Un VCS ayuda a coordinar las actividades de los equipos al permitir que varios desarrolladores trabajen en el mismo conjunto de archivos simultáneamente. Los primeros VCS hicieron esto al otorgar a una persona a la vez el derecho de editar un archivo; ese estilo de bloqueo es suficiente para establecer la secuencia (un acuerdo "que es más nuevo", una característica importante de VCS). Los sistemas más avanzados aseguran que los cambios en un *recopilación*de los archivos enviados a la vez se tratan como una sola unidad (*atomicidad* cuando un cambio lógico toca varios archivos). Los sistemas como CVS (un VCS popular de los años 90) que no tenían esta atomicidad para una confirmación estaban sujetos a corrupción y cambios perdidos. Garantizar la atomicidad elimina la posibilidad de que los cambios anteriores se sobrescriban involuntariamente, pero requiere rastrear con qué versión se sincronizó por última vez: en el momento de la confirmación, la confirmación se rechaza si algún archivo en la confirmación se ha modificado en el encabezado desde la última vez que la desarrollador local sincronizado. Especialmente en un VCS de seguimiento de cambios de este tipo, la copia de trabajo de un desarrollador de los archivos administrados necesitará sus propios metadatos. Según el diseño del VCS, esta copia del repositorio puede ser un repositorio en sí mismo o puede contener una cantidad reducida de metadatos; dicha copia reducida suele ser un "cliente" o un "área de trabajo".

Esto parece mucha complejidad: ¿por qué es necesario un VCS? ¿Qué tiene este tipo de herramienta que le ha permitido convertirse en una de las pocas herramientas casi universales para el desarrollo de software y la ingeniería de software?

Imagine por un momento trabajar sin un VCS. Para un grupo (muy) pequeño de desarrolladores distribuidos que trabajan en un proyecto de alcance limitado sin ninguna comprensión del control de versiones, la solución más simple y con la infraestructura más baja es simplemente pasar copias del proyecto de un lado a otro. Esto funciona mejor cuando las ediciones no son simultáneas (las personas trabajan en diferentes zonas horarias, o al menos con diferentes horas de trabajo). Si existe la posibilidad de que las personas no sepan qué versión es la más actualizada, inmediatamente tenemos un problema molesto: rastrear qué versión es la más actualizada. Cualquiera que haya intentado colaborar en un entorno fuera de la red probablemente recordará los horrores de copiar archivos de ida y vuelta llamados *Presentación v5 - final*

- *líneas rojas - versión v2 de Josh*. Y como veremos, cuando no hay una sola fuente de verdad acordada, la colaboración se vuelve muy conflictiva y propensa a errores.

La introducción del almacenamiento compartido requiere un poco más de infraestructura (obtener acceso al almacenamiento compartido), pero proporciona una solución fácil y obvia. La coordinación del trabajo en una unidad compartida puede ser suficiente por un tiempo con un número lo suficientemente pequeño de personas, pero aún requiere una colaboración fuera de banda para evitar sobrescribir el trabajo de los demás. Además, trabajar directamente en ese almacenamiento compartido significa que cualquier tarea de desarrollo que no

2 Aunque la idea formal de lo que es y no es un repositorio cambia un poco dependiendo de su elección de VCS, y la terminología variará.

mantener la compilación funcionando continuamente comenzará a obstaculizar a todos en el equipo: si estoy haciendo un cambio en alguna parte de este sistema al mismo tiempo que inicia una compilación, su compilación no funcionará. Obviamente, esto no escala bien.

En la práctica, la falta de bloqueo de archivos y la falta de seguimiento de combinación conducirán inevitablemente a colisiones y a la sobrescritura del trabajo. Es muy probable que dicho sistema introduzca una coordinación fuera de banda para decidir quién está trabajando en un archivo determinado. Si ese bloqueo de archivos está codificado en software, hemos comenzado a reinventar un control de versiones de primera generación como RCS (entre otros). Después de darse cuenta de que otorgar permisos de escritura de un archivo a la vez es demasiado toso y comienza a desechar un seguimiento a nivel de línea, definitivamente estamos reinventando el control de versiones. Parece casi inevitable que querremos algún mecanismo estructurado para gobernar estas colaboraciones. Debido a que parece que solo estamos reinventando la rueda en esta hipótesis, también podríamos usar una herramienta lista para usar.

¿Por qué es importante el control de versiones?

Si bien el control de versiones es prácticamente omnipresente ahora, no siempre fue así. Los primeros VCS se remontan a la década de 1970 (SCCS) y la década de 1980 (RCS), muchos años después de las primeras referencias a la ingeniería de software como una disciplina distinta. Los equipos participaron en “[la desarrollo multipersona de software multiversion](#)” antes de que la industria tuviera una noción formal de control de versiones. El control de versiones evolucionó como respuesta a los nuevos desafíos de la colaboración digital. Se necesitaron décadas de evolución y difusión para que el uso confiable y consistente del control de versiones evolucionara hasta convertirse en la norma que es hoy.³ Entonces, ¿cómo se volvió tan importante y, dado que parece una solución evidente, por qué alguien podría resistirse a la idea de VCS?

Recuerde que la ingeniería de software es una programación integrada en el tiempo; estamos trazando una distinción (en dimensionalidad) entre la producción instantánea de código fuente y el acto de mantener ese producto a lo largo del tiempo. Esa distinción básica explica en gran medida la importancia y la facilidad de VCS: en el nivel más fundamental, el control de versiones es la herramienta principal del ingeniero para administrar la interacción entre la fuente sin procesar y el tiempo. Podemos conceptualizar VCS como una forma de extender un sistema de archivos estándar. Un sistema de archivos es una asignación del nombre del archivo al contenido. Un VCS amplía eso para proporcionar una asignación desde (nombre de archivo, hora) hasta el contenido, junto con los metadatos necesarios para rastrear los últimos puntos de sincronización y el historial de auditoría. El control de versiones hace que la consideración del tiempo sea una parte explícita de la operación: innecesario en un programa.

3 De hecho, he dado varias charlas públicas que utilizan la "adopción del control de versiones" como el ejemplo canónico de cómo las normas de la ingeniería de software pueden *hacer* revolucionar con el tiempo. En mi experiencia, en la década de 1990, el control de versiones se entendía bastante bien como una mejor práctica, pero no se seguía universalmente. A principios de la década de 2000, todavía era común encontrarse con grupos profesionales que no lo usaban. Hoy en día, el uso de herramientas como Git parece omnipresente incluso entre los estudiantes universitarios que trabajan en proyectos personales. Es probable que parte de este aumento en la adopción se deba a una mejor experiencia del usuario en las herramientas (nadie quiere volver a RCS), pero el papel de la experiencia y las normas cambiantes es importante.

tarea de ming, crítica en una tarea de ingeniería de software. En la mayoría de los casos, un VCS también permite una entrada adicional a esa asignación (un nombre de rama) para permitir asignaciones paralelas; por lo tanto:

VCS (nombre de archivo, hora, rama) => contenido del archivo

En el uso predeterminado, esa entrada de rama tendrá un valor predeterminado comúnmente entendido: lo llamamos "cabeza", "predeterminado" o "troncal" para indicar la rama principal.

La vacilación (menor) restante hacia el uso consistente del control de versiones proviene casi directamente de combinar programación e ingeniería de software: enseñamos programación, capacitamos a programadores, entrevistamos para trabajos basados en problemas y técnicas de programación. Es perfectamente razonable que un nuevo empleado, incluso en un lugar como Google, tenga poca o ninguna experiencia con el código en el que trabaja más de una persona o durante más de un par de semanas. Dada esa experiencia y comprensión del problema, el control de versiones parece una solución extraña. El control de versiones está resolviendo un problema que nuestro nuevo empleado no necesariamente ha experimentado: un "deshacer", no para un solo archivo sino para todo un proyecto, agregando mucha complejidad para beneficios a veces no obvios.

En algunos grupos de software, se produce el mismo resultado cuando la gerencia ve el trabajo de los técnicos como "desarrollo de software" (sentarse y escribir código) en lugar de "ingeniería de software" (producir código, mantenerlo funcionando y útil durante algún tiempo). período). Con un modelo mental de programación como tarea principal y poca comprensión de la interacción entre el código y el paso del tiempo, es fácil ver algo descrito como "volver a una versión anterior para deshacer un error" como algo extraño., lujo de techo alto.

Además de permitir el almacenamiento separado y la referencia a las versiones a lo largo del tiempo, el control de versiones nos ayuda a cerrar la brecha entre los procesos de un solo desarrollador y de múltiples desarrolladores. En términos prácticos, esta es la razón por la cual el control de versiones es tan crítico para la ingeniería de software, porque nos permite escalar equipos y organizaciones, aunque lo usamos con poca frecuencia como un botón de "deshacer". El desarrollo es inherentemente un proceso de bifurcación y fusión, tanto cuando se coordina entre múltiples desarrolladores o un solo desarrollador en diferentes puntos en el tiempo. Un VCS elimina la pregunta de "¿cuál es más reciente?" El uso del control de versiones moderno automatiza las operaciones propensas a errores, como el seguimiento del conjunto de cambios que se han aplicado. El control de versiones es cómo nos coordinamos entre múltiples desarrolladores y/o múltiples puntos en el tiempo.

Debido a que VCS se ha integrado tan completamente en el proceso de ingeniería de software, incluso las prácticas legales y reglamentarias se han puesto al día. VCS permite un registro formal de cada cambio en cada línea de código, lo que es cada vez más necesario para satisfacer los requisitos de auditoría. Al combinar el desarrollo interno y el uso adecuado de fuentes de terceros, VCS ayuda a rastrear la procedencia y el origen de cada línea de código.

Además de los aspectos técnicos y regulatorios del seguimiento de la fuente a lo largo del tiempo y el manejo de las operaciones de sincronización/ramificación/fusión, el control de versiones desencadena algunos cambios no técnicos en el comportamiento. El ritual de comprometerse con el control de versiones y generar un registro de confirmación desencadena un momento de reflexión: ¿qué ha logrado desde su última confirmación? ¿Está la fuente en un estado con el que está satisfecho? El momento de introspección asociado con el compromiso, la redacción de un resumen y la finalización de una tarea puede tener valor por sí solo para muchas personas. El inicio del proceso de confirmación es un momento perfecto para ejecutar una lista de verificación, ejecutar análisis estáticos ([ver capítulo 20](#)), verificar la cobertura de prueba, ejecutar pruebas y análisis dinámico, etc.

Como cualquier proceso, el control de versiones conlleva algunos gastos generales: alguien debe configurar y administrar su sistema de control de versiones, y los desarrolladores individuales deben usarlo. Pero no se equivoque al respecto: estos casi siempre pueden ser bastante baratos. Como anécdota, los ingenieros de software más experimentados usarán instintivamente el control de versiones para cualquier proyecto que dure más de uno o dos días, incluso para un proyecto de un solo desarrollador. La coherencia de ese resultado argumenta que la compensación en términos de valor (incluida la reducción de riesgos) frente a los gastos generales debe ser bastante sencilla. Pero hemos prometido reconocer que el contexto es importante y alentar a los líderes de ingeniería a pensar por sí mismos. Siempre vale la pena considerar alternativas, incluso en algo tan fundamental como el control de versiones.

En verdad, es difícil imaginar una tarea que pueda considerarse ingeniería de software moderna que no adopte inmediatamente un VCS. Dado que comprende el valor y la necesidad del control de versiones, es probable que ahora se pregunte qué tipo de control de versiones necesita.

VCS centralizado frente a VCS distribuido

En el nivel más simple, todos los VCS modernos son equivalentes entre sí: siempre que su sistema tenga la noción de realizar cambios atómicamente en un lote de archivos, todo lo demás es solo IU. Podría construir la misma semántica general (no el flujo de trabajo) de cualquier VCS moderno a partir de otro y una pila de scripts de shell simples. Por lo tanto, discutir sobre qué VCS es "mejor" es principalmente una cuestión de experiencia del usuario: la funcionalidad principal es la misma, las diferencias se presentan en la experiencia del usuario, la denominación, las características de casos extremos y el rendimiento. Elegir un VCS es como elegir un formato de sistema de archivos: al elegir entre un formato lo suficientemente moderno, las diferencias son bastante menores, y la pregunta más importante es el contenido con el que llena ese sistema y la forma en que lo hace.*utilizareso*. Sin embargo, las principales diferencias arquitectónicas en los VCS pueden facilitar o dificultar las decisiones de configuración, política y escalado, por lo que es importante ser consciente de las grandes diferencias arquitectónicas, principalmente la decisión entre centralizado o descentralizado.

SVC centralizado

En las implementaciones centralizadas de VCS, el modelo es uno de un repositorio central único (probablemente almacenado en algún recurso informático compartido para su organización). Aunque un desarrollador puede tener archivos desprotegidos y accesibles en su estación de trabajo local, las operaciones que interactúan en el estado de control de versión de esos archivos deben comunicarse al servidor central (agregar archivos, sincronizar, actualizar archivos existentes, etc.). Cualquier código enviado por un desarrollador se envía a ese repositorio central. Las primeras implementaciones de VCS fueron todos VCS centralizados.

Volviendo a la década de 1970 y principios de la de 1980, vemos que los primeros de estos VCS, como RCS, se centraron en bloquear y evitar múltiples ediciones simultáneas. Puede copiar el contenido de un repositorio, pero si desea editar un archivo, es posible que deba adquirir un bloqueo, aplicado por el VCS, para garantizar que solo usted realice las modificaciones. Cuando haya completado una edición, libera el bloqueo. El modelo funcionaba bien cuando cualquier cambio dado era algo rápido, o si rara vez había más de una persona que deseaba el bloqueo de un archivo en un momento dado. Las pequeñas ediciones, como ajustar los archivos de configuración, funcionaron bien, al igual que trabajar en un equipo pequeño que mantuvo horas de trabajo inconexas o que rara vez trabajó en archivos superpuestos durante períodos prolongados. Este tipo de bloqueo simplista tiene problemas inherentes con la escala: puede funcionar bien para algunas personas,⁴

Como respuesta a este problema de escalamiento, los VCS que fueron populares durante los años 90 y principios de los 2000 operaron a un nivel superior. Estos VCS centralizados más modernos evitan el bloqueo exclusivo pero rastrean qué cambios ha sincronizado, lo que requiere que su edición se base en la versión más reciente de cada archivo en su confirmación. CVS envolvió y perfeccionó RCS (principalmente) operando en lotes de archivos a la vez y permitiendo que varios desarrolladores desprotegieran un archivo al mismo tiempo: siempre que su versión base contuviera todos los cambios en el repositorio, se le permite cometer. Subversion avanzó aún más al proporcionar una verdadera atomicidad para confirmaciones, seguimiento de versiones y un mejor seguimiento de operaciones inusuales (cambios de nombre, uso de enlaces simbólicos, etc.).

VCS distribuido

A partir de mediados de la década de 2000, muchos VCS populares siguieron el paradigma del Sistema de control de versiones distribuidas (DVCS), visto en sistemas como Git y Mercurial. Él

⁴ Anécdota: para ilustrar esto, busqué información sobre las ediciones pendientes o no enviadas que Googlers había realizado. Representando un archivo semipopular en mi proyecto más reciente. En el momento de escribir este artículo, hay 27 cambios pendientes, 12 de personas de mi equipo, 5 de personas de equipos relacionados y 10 de ingenieros que nunca he conocido. Esto está funcionando básicamente como se esperaba. Los sistemas técnicos o las políticas que requieren coordinación fuera de banda ciertamente no se adaptan a la ingeniería de software las 24 horas del día, los 7 días de la semana en ubicaciones distribuidas.

La principal diferencia conceptual entre DVCS y VCS centralizado más tradicional (Subversion, CVS) es la pregunta: "¿Dónde puede comprometerse?" o tal vez, "¿Qué copias de estos archivos cuentan como repositorio?"

Un mundo DVCS no impone la restricción de un repositorio central: si tiene una copia (clon, bifurcación) del repositorio, tiene un repositorio con el que puede comprometerse, así como todos los metadatos necesarios para consultar información sobre cosas, como el historial de revisiones. Un flujo de trabajo estándar es clonar algún repositorio existente, hacer algunas modificaciones, enviarlas localmente y luego enviar un conjunto de confirmaciones a otro repositorio, que puede ser o no la fuente original de la clonación. Cualquier noción de centralidad es puramente conceptual, una cuestión de política, no fundamental para la tecnología o los protocolos subyacentes.

El modelo DVCS permite una mejor colaboración y operación fuera de línea sin declarar inherentemente que un repositorio en particular es la fuente de la verdad. Un repositorio no es necesario "adelante" o "detrás" porque los cambios no se proyectan inherentemente en una línea de tiempo lineal. Sin embargo, considerando común uso, tanto el modelo centralizado como el DVCS son en gran medida intercambiables: mientras que un VCS centralizado proporciona un repositorio central claramente definido a través de la tecnología, la mayoría de los ecosistemas DVCS definen un repositorio central para un proyecto como una cuestión de política. Es decir, la mayoría de los proyectos de DVCS se construyen alrededor de una fuente conceptual de verdad (un repositorio particular en GitHub, por ejemplo). Los modelos DVCS tienden a asumir un caso de uso más distribuido y han encontrado una adopción particularmente sólida en el mundo del código abierto.

En términos generales, el sistema de control de fuente dominante en la actualidad es Git, que implementa DVCS.⁵ En caso de duda, utilícelo: hay algo de valor en hacer lo que hacen los demás. Si se espera que sus casos de uso sean inusuales, recopile algunos datos y evalúe las compensaciones.

Google tiene una relación compleja con DVCS: nuestro repositorio principal se basa en un VCS centralizado interno personalizado (masivo). Hay intentos periódicos de integrar más opciones externas estándar y de igualar el flujo de trabajo que nuestros ingenieros (especialmente los Nooglers) esperan del desarrollo externo. Desafortunadamente, esos intentos de avanzar hacia herramientas más comunes como Git se han visto obstaculizados por el gran tamaño de la base de código y la base de usuarios, por no hablar de los efectos de la Ley de Hyrum que nos vinculan a un VCS en particular y la interfaz para ese VCS.⁶ Quizás esto no sea sorprendente: la mayoría de las herramientas existentes no escalan bien con 50,000 ingenieros y decenas de millones de

⁵ Desbordamiento de pilaResultados de la encuesta de desarrolladores, 2018.

⁶ Los números de versión que aumentan monótonamente, en lugar de los hashes de confirmación, son particularmente problemáticos. Muchos han crecido sistemas y secuencias de comandos en el ecosistema de desarrolladores de Google que asumen que el orden numérico de las confirmaciones es el mismo que el orden temporal; deshacer esas dependencias ocultas es difícil.

se compromete⁷ El modelo DVCS, que a menudo (pero no siempre) incluye la transmisión de historial y metadatos, requiere una gran cantidad de datos para poner en funcionamiento un repositorio.

En nuestro flujo de trabajo, la centralidad y el almacenamiento en la nube para el código base parecen ser fundamentales para escalar. El modelo DVCS se basa en la idea de descargar el código base completo y tener acceso a él localmente. En la práctica, con el tiempo y a medida que su organización crece, cualquier desarrollador operará en un porcentaje relativamente menor de los archivos en un repositorio y en una pequeña fracción de las versiones de esos archivos. A medida que crecemos (en número de archivos y número de ingenieros), esa transmisión se convierte casi en su totalidad en un desperdicio. La única necesidad de localidad para la mayoría de los archivos ocurre durante la construcción, pero los sistemas de construcción distribuidos (y reproducibles) también parecen escalar mejor para esa tarea (ver [capítulo 18](#)).

Fuente de la verdad

Los VCS centralizados (Subversion, CVS, Perforce, etc.) integran la noción de la fuente de la verdad en el diseño mismo del sistema: lo que se haya confirmado más recientemente en el troncal es la versión actual. Cuando un desarrollador va a verificar el proyecto, de manera predeterminada, esa versión troncal es la que se le presentará. Sus cambios están "hechos" cuando se han vuelto a confirmar sobre esa versión.

Sin embargo, a diferencia del VCS centralizado, no hay *inherent* noción de qué copia del repositorio distribuido es la única fuente de verdad en los sistemas DVCS. En teoría, es posible pasar etiquetas de compromiso y PR sin centralización ni coordinación, lo que permite que ramas dispares de desarrollo se propaguen sin control y, por lo tanto, arriesgan un retorno conceptual al mundo de *Presentación v5 - final - líneas rojas - versión de Josh v2*. Debido a esto, DVCS requiere políticas y normas más explícitas que un VCS centralizado.

Los proyectos bien administrados que usan DVCS declaran que una rama específica en un repositorio específico es la fuente de la verdad y así evitan las posibilidades más caóticas. Vemos esto en la práctica con la difusión de soluciones DVCS alojadas como GitHub o GitLabusers pueden clonar y bifurcar el repositorio para un proyecto, pero todavía hay un único repositorio principal: las cosas están "hechas" cuando están en la rama troncal de ese repositorio. .

No es un accidente que la centralización y la Fuente de la verdad se hayan vuelto a utilizar incluso en un mundo DVCS. Para ayudar a ilustrar cuán importante es esta idea de la Fuente de la Verdad, imaginemos lo que sucede cuando no tenemos una fuente clara de la verdad.

7 Para el caso, a partir de la publicación del artículo de Monorepo, el repositorio en sí tenía algo así como 86 TB de datos y metadatos, ignorando las ramas de lanzamiento. Instalar eso en una estación de trabajo de desarrollador directamente sería... un desafío.

Escenario: ninguna fuente clara de la verdad

Imagine que su equipo se adhiere a la filosofía de DVCS lo suficiente como para evitar definir una sucursal+repositorio específico como la fuente definitiva de la verdad.

En algunos aspectos, esto recuerda a la *Presentación v5 - final - líneas rojas - versión de Josh v2* modelo: después de extraer del repositorio de un compañero de equipo, no está necesariamente claro qué cambios están presentes y cuáles no. En algunos aspectos, es mejor que eso porque el modelo DVCS rastrea la combinación de parches individuales con una granularidad mucho más fina que esos esquemas de nombres ad hoc, pero hay una diferencia entre que el DVCS *sepa* incorporar los cambios y cada ingeniero asegurándose de tener *todas* los cambios pasados/relevantes representados.

Considere lo que se necesita para asegurarse de que una compilación de lanzamiento incluya todas las características que ha desarrollado cada desarrollador durante las últimas semanas. ¿Qué mecanismos (no centralizados, escalables) existen para hacerlo? ¿Podemos diseñar políticas que sean fundamentalmente mejores que hacer que todos firmen? ¿Hay alguno que requiera solo un esfuerzo humano sublineal a medida que el equipo crece? ¿Seguirá funcionando a medida que aumente el número de desarrolladores en el equipo? Por lo que podemos ver: probablemente no. Sin una fuente central de la verdad, alguien va a mantener una lista de las funciones que están potencialmente listas para incluirse en la próxima versión. Eventualmente, esa contabilidad está reproduciendo el modelo de tener una Fuente de la Verdad centralizada.

Además, imagine: cuando un nuevo desarrollador se une al equipo, ¿dónde obtienen una copia nueva y en buen estado del código?

DVCS permite una gran cantidad de excelentes flujos de trabajo y modelos de uso interesantes. Pero si le preocupa encontrar un sistema que requiera un esfuerzo humano sublineal para administrarlo a medida que el equipo crece, es muy importante tener un repositorio (y una rama) realmente definido para ser la fuente definitiva de la verdad.

Hay algo de relatividad en esa Fuente de la Verdad. Es decir, para un proyecto dado, esa Fuente de la Verdad podría ser diferente para una organización diferente. Esta advertencia es importante: es razonable que los ingenieros de Google o RedHat tengan diferentes fuentes de la verdad para los parches del kernel de Linux, aún diferentes de las que tendría Linus (el mantenedor del kernel de Linux). DVCS funciona bien cuando las organizaciones y sus Fuentes de la verdad son jerárquicas (e invisibles para quienes están fuera de la organización); ese es quizás el efecto más útil en la práctica del modelo DVCS. Un ingeniero de RedHat puede comprometerse con el repositorio local de la Fuente de la Verdad, y los cambios pueden impulsarse periódicamente desde allí, mientras que Linus tiene una noción completamente diferente de lo que es la Fuente de la Verdad. Siempre que no haya elección o incertidumbre sobre dónde se debe impulsar un cambio,

En todo este pensamiento, estamos asignando un significado especial a la rama troncal. Pero, por supuesto, "troncal" en su VCS es solo la tecnología predeterminada, y una organización puede

elegir diferentes políticas además de eso. Tal vez se abandonó la rama predeterminada y todo el trabajo realmente se realiza en alguna rama de desarrollo personalizada, aparte de la necesidad de proporcionar un nombre de rama en más operaciones, no hay nada intrínsecamente roto en ese enfoque; es simplemente no estándar. Hay una verdad (a menudo tácita) cuando se habla del control de versiones: la tecnología es solo una parte de ella para cualquier organización determinada; casi siempre hay una cantidad igual de política y convención de uso además de eso.

Ningún tema en el control de versiones tiene más política y convención que la discusión sobre cómo usar y administrar las ramas. Analizamos la gestión de sucursales con más detalle en la siguiente sección.

Control de versiones frente a gestión de dependencias

Hay mucha similitud conceptual entre las discusiones sobre las políticas de control de versiones y la gestión de dependencias ([ver capítulo 21](#)). Las diferencias se encuentran principalmente en dos formas: las políticas de VCS se refieren en gran medida a cómo administra su propio código y, por lo general, son mucho más detalladas. La administración de dependencias es más desafiante porque nos enfocamos principalmente en proyectos administrados y controlados por otras organizaciones, con una mayor granularidad, y estas situaciones significan que no tiene un control perfecto. Hablaremos mucho más de estos temas de alto nivel más adelante en el libro.

Gestión de Sucursales

Ser capaz de rastrear diferentes revisiones en el control de versiones abre una variedad de enfoques diferentes sobre cómo administrar esas diferentes versiones. Colectivamente, estos diferentes enfoques caen bajo el término *gestión de sucursales*, en contraste con un solo "baúl".

El trabajo en progreso es similar a una sucursal

Cualquier discusión que tenga una organización sobre las políticas de administración de sucursales debe reconocer al menos que cada pieza de trabajo en progreso en la organización es equivalente a una sucursal. Esto es más explícito en el caso de un DVCS en el que es más probable que los desarrolladores realicen numerosas confirmaciones de pruebas locales antes de retroceder a la Fuente de la verdad aguas arriba. Esto sigue siendo cierto para los VCS centralizados: los cambios locales no confirmados no son conceptualmente diferentes de los cambios confirmados en una rama, además de ser potencialmente más difíciles de encontrar y comparar. Algunos sistemas centralizados incluso lo hacen explícito. Por ejemplo, cuando se usa Perforce, cada cambio recibe dos números de revisión: uno que indica el punto de ramificación implícito donde se creó el cambio y otro que indica dónde se volvió a confirmar, como se ilustra en [Figura 16-1](#). Los usuarios de Perforce pueden consultar para ver quién tiene cambios pendientes en un archivo determinado, inspeccionar los cambios pendientes en los cambios no confirmados de otros usuarios y más.

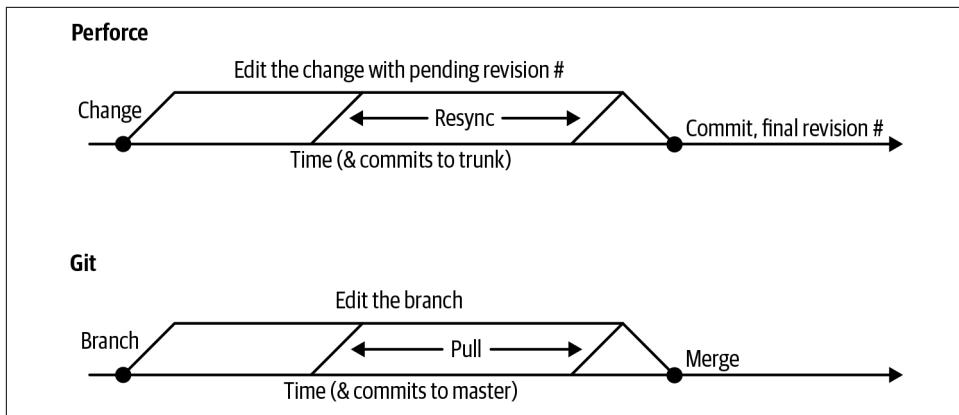


Figura 16-1. Dos números de revisión en Perforce

Esta idea de "el trabajo no comprometido es similar a una rama" es particularmente relevante cuando se piensa en tareas de refactorización. Imagine que a un desarrollador se le dice: "Cambio el nombre de Widget a OldWidget". Según las políticas de administración de sucursales de una organización y la comprensión, lo que cuenta como una sucursal y qué sucursales importan, esto podría tener varias interpretaciones:

- Renombrar Widget en la rama troncal en el repositorio de Source of Truth
- Renombrar Widget en todas las ramas en el repositorio de Source of Truth
- Cambie el nombre de Widget en todas las ramas en el repositorio de Source of Truth y encuentre todos los desarrolladores con cambios destacados en los archivos que hacen referencia a Widget

Si tuviéramos que especular, intentar respaldar el caso de uso de "cambiar el nombre de esto en todas partes, incluso en cambios sobresalientes" es parte de por qué los VCS comerciales centralizados tienden a rastrear cosas como "¿qué ingenieros tienen este archivo abierto para editar?" (No creemos que esta sea una forma escalable de llevar a cabo una tarea de refactorización, pero entendemos el punto de vista).

Ramas de desarrollo

En la era anterior a las pruebas unitarias consistentes (ver [Capítulo 11](#)), cuando la introducción de cualquier cambio dado tenía un alto riesgo de hacer retroceder la funcionalidad en otras partes del sistema, tenía sentido tratar *maleroe* especialmente. "No nos comprometemos con el troncal", podría decir su Tech Lead, "hasta que los nuevos cambios hayan pasado por una ronda completa de pruebas. En su lugar, nuestro equipo utiliza ramas de desarrollo específicas de funciones".

Una rama de desarrollo (generalmente "rama de desarrollo") es un punto medio entre "esto se hace pero no se compromete" y "esto es en lo que se basa el nuevo trabajo". El problema que estos están tratando de resolver (inestabilidad del producto) es legítimo, pero uno que

hemos encontrado que se resuelve mucho mejor con un uso más extenso de pruebas, Integración Continua (CI) ([ver capítulo 23](#)) y prácticas de aplicación de la calidad, como la revisión exhaustiva del código.

Creemos que una política de control de versiones que hace un uso extensivo de las ramas de desarrollo como un medio para lograr la estabilidad del producto es intrínsecamente equivocada. El mismo conjunto de confirmaciones se fusionará con el tronco eventualmente. Las fusiones pequeñas son más fáciles que las grandes. Las fusiones realizadas por el ingeniero que creó esos cambios son más fáciles que agrupar los cambios no relacionados y fusionarlos más tarde (lo que sucederá eventualmente si un equipo comparte una rama de desarrollo). Si las pruebas previas al envío en la fusión revelan nuevos problemas, se aplica el mismo argumento: es más fácil determinar de quién son los cambios responsables de una regresión si solo hay un ingeniero involucrado. La fusión de una rama de desarrollo grande implica que se están produciendo más cambios en esa ejecución de prueba, lo que hace que las fallas sean más difíciles de aislar. La clasificación y la causa raíz del problema es difícil; arreglarlo es aún peor.

Más allá de la falta de experiencia y los problemas inherentes a la fusión de una sola sucursal, existen importantes riesgos de escalamiento cuando se depende de las sucursales de desarrollo. Esta es una pérdida de productividad muy común para una organización de software. Cuando hay varias ramas que se desarrollan de forma aislada durante períodos prolongados, la coordinación de las operaciones de fusión se vuelve significativamente más costosa (y posiblemente más riesgosa) de lo que sería con el desarrollo basado en troncales.

¿Cómo nos volvimos adictos a las ramas de desarrollo?

Es fácil ver cómo las organizaciones caen en esta trampa: ven: "La fusión de esta rama de desarrollo de larga duración redujo la estabilidad" y concluyen: "Las fusiones de ramas son riesgosas". En lugar de resolver eso con "mejores pruebas" y "no usar estrategias de desarrollo basadas en sucursales", se enfocan en desacelerar y coordinar el síntoma: la sucursal se fusiona. Los equipos comienzan a desarrollar nuevas sucursales basadas en otras sucursales en vuelo. Los equipos que trabajan en una rama de desarrollo de larga duración pueden o no sincronizar regularmente esa rama con la rama de desarrollo principal. A medida que la organización crece, la cantidad de sucursales de desarrollo también crece y se pone más esfuerzo en coordinar la estrategia de fusión de esa sucursal. Se dedica un esfuerzo cada vez mayor a la coordinación de fusiones de sucursales, una tarea que inherentemente no escala. Algun ingeniero desafortunado se convierte en el maestro de compilación/coordinador de fusión/ingeniero de administración de contenido, enfocado en actuar como el coordinador único para fusionar todas las ramas dispares en la organización. Las reuniones programadas regularmente intentan garantizar que la organización haya "desarrollado la estrategia de fusión para la semana". Los equipos que no se eligen para fusionarse a menudo necesitan volver a sincronizarse y volver a realizar la prueba después de cada una de estas grandes fusiones.

⁸ Una encuesta informal reciente en Twitter sugiere que alrededor del 25 % de los ingenieros de software han estado sujetos a reuniones de estrategia de fusión programadas.

Todo ese esfuerzo de fusionarse y volver a probar *esgastos generales puros*. La alternativa requiere un paradigma diferente: desarrollo basado en troncos, depender en gran medida de las pruebas y la CI, mantener la construcción verde y deshabilitar las funciones incompletas/no probadas en el tiempo de ejecución. Todos son responsables de sincronizar con el troncal y confirmar; sin reuniones de "estrategia de fusión", sin fusiones grandes/caras. Y, sin discusiones acaloradas sobre qué versión de una biblioteca se debe usar

- Sólo puede haber uno. Debe haber una sola Fuente de la Verdad. Al final, se usará una sola revisión para un lanzamiento: reducir a una sola fuente de verdad es solo el enfoque de "desplazamiento a la izquierda" para identificar qué se incluye y qué no.

Ramas de lanzamiento

Si el período entre versiones (o la vida útil de la versión) de un producto es superior a unas pocas horas, puede ser sensato crear una rama de versión que represente el código exacto que se incluyó en la compilación de la versión de su producto. Si se descubren fallas críticas entre el lanzamiento real de ese producto y el próximo ciclo de lanzamiento, las correcciones se pueden seleccionar (una combinación mínima y dirigida) desde el tronco hasta su rama de lanzamiento.

En comparación con las ramas de desarrollo, las ramas de lanzamiento son generalmente benignas: no es la tecnología de las ramas lo que es problemático, es el uso. La principal diferencia entre una rama de desarrollo y una rama de lanzamiento es el estado final esperado: se espera que una rama de desarrollo se fusione de nuevo con el tronco, e incluso podría ser ramificada por otro equipo. Se espera que eventualmente se abandone una rama de lanzamiento.

En las organizaciones técnicas de más alto funcionamiento que ha identificado la organización DevOps Research and Assessment (DORA) de Google, las sucursales de lanzamiento son prácticamente inexistentes. Es probable que las organizaciones que han logrado la implementación continua (CD), la capacidad de liberar desde el enlace troncal muchas veces al día, tiendan a omitir las ramas de liberación: es mucho más fácil simplemente agregar la corrección y volver a implementar. Por lo tanto, los picos de cereza y las ramas parecen una sobrecarga innecesaria. Obviamente, esto es más aplicable a las organizaciones que implementan digitalmente (como servicios web y aplicaciones) que a aquellas que impulsan cualquier forma de lanzamiento tangible a los clientes; por lo general, es valioso saber exactamente lo que se ha enviado a los clientes.

Esa misma investigación de DORA también sugiere una fuerte correlación positiva entre el "desarrollo basado en troncales", "sin ramas de desarrollo de larga duración" y buenos resultados técnicos. La idea subyacente en ambas ideas parece clara: las sucursales son un lastre para la productividad. En muchos casos, pensamos que las estrategias complejas de bifurcación y combinación son una muleta de seguridad percibida, un intento de mantener estable el tronco. Como vemos a lo largo de este libro, hay otras formas de lograr ese resultado.

Control de versiones en Google

En Google, la gran mayoría de nuestra fuente se administra en un único repositorio (mono-repo) compartido entre aproximadamente 50,000 ingenieros. Casi todos los proyectos que son propiedad de Google viven allí, excepto los grandes proyectos de código abierto como Chromium y Android. Esto incluye productos de cara al público como la Búsqueda, Gmail, nuestros productos publicitarios, nuestras ofertas de Google Cloud Platform, así como la infraestructura interna necesaria para respaldar y desarrollar todos esos productos.

Confiamos en un VCS centralizado desarrollado internamente llamado Piper, creado para ejecutarse como un microservicio distribuido en nuestro entorno de producción. Esto nos ha permitido utilizar el almacenamiento, la comunicación y la tecnología informática como servicio estándar de Google para proporcionar un VCS disponible a nivel mundial que almacena más de 80 TB de contenido y metadatos. El monorepo de Piper es editado y comprometido simultáneamente por miles de ingenieros todos los días. Entre humanos y procesos semiautomáticos que utilizan el control de versiones (o mejoran las cosas registradas en VCS), manejaremos regularmente entre 60 000 y 70 000 confirmaciones en el repositorio por día laboral. Los artefactos binarios son bastante comunes porque no se transmite el repositorio completo y, por lo tanto, los costos normales de los artefactos binarios no se aplican realmente. Debido al enfoque en la escala de Google desde la primera concepción, las operaciones en este ecosistema de VCS siguen siendo baratas a escala humana: se tarda quizás 15 segundos en total en crear un nuevo cliente en el enlace troncal, agregar un archivo y confirmar un cambio (no revisado) en Piper. Esta interacción de baja latencia y escalamiento bien entendido/bien diseñado simplifica mucho la experiencia del desarrollador.

En virtud de que Piper es un producto interno, tenemos la capacidad de personalizarlo y aplicar las políticas de control de fuente que elijamos. Por ejemplo, tenemos una noción de propiedad granular en el monorepo: en cada nivel de la jerarquía de archivos, podemos encontrar archivos OWNERS que enumeran los nombres de usuario de los ingenieros que pueden aprobar confirmaciones dentro de ese subárbol del repositorio (además de a los PROPIETARIOS que figuran en los niveles superiores del árbol). En un entorno con muchos repositorios, esto podría haberse logrado al tener repositorios separados con la aplicación de permisos del sistema de archivos controlando el acceso de confirmación o a través de un "gancho de confirmación" de Git (acción desencadenada en el momento de la confirmación) para realizar una verificación de permisos por separado. Al controlar el VCS, podemos hacer que el concepto de propiedad y aprobación sea más explícito y aplicado por el VCS durante un intento de operación de confirmación. El modelo también es flexible: la propiedad es solo un archivo de texto, no vinculado a una separación física de repositorios, por lo que es trivial actualizarlo como resultado de una transferencia de equipo o una reestructuración de la organización.

Una versión

Los increíbles poderes de escalamiento de Piper por sí solos no permitirían el tipo de colaboración en la que confiamos. Como dijimos anteriormente: el control de versiones también tiene que ver con la política. Además de nuestro VCS, una característica clave de la política de control de versiones de Google es lo que llamamos "Versión única". Esto amplía el concepto de "Fuente Única de la Verdad" que

visto anteriormente, asegurándose de que un desarrollador sepa qué rama y repositorio es su fuente de verdad, a algo así como "Para cada dependencia en nuestro repositorio, debe haber solo una versión de esa dependencia para elegir".⁹ Para paquetes de terceros, esto significa que solo puede haber una única versión de ese paquete registrado en nuestro repositorio, en estado estable.¹⁰ Para paquetes internos, esto significa que no hay bifurcación sin volver a empaquetar/cambiar el nombre: debe ser tecnológicamente seguro mezclar tanto el original como la bifurcación en el mismo proyecto sin un esfuerzo especial. Esta es una característica poderosa para nuestro ecosistema: hay muy pocos paquetes con restricciones como "Si incluye este paquete (A), no puede incluir otro paquete (B)".

Esta noción de tener una sola copia en una sola rama en un solo repositorio como nuestra Fuente de la Verdad es intuitiva pero también tiene una aplicación sutilmente profunda. Investiguemos un escenario en el que tenemos un monorepo (y, por lo tanto, podría decirse que hemos cumplido con la letra de la ley sobre la Fuente Única de la Verdad), pero hemos permitido que las bifurcaciones de nuestras bibliotecas se propaguen en el tronco.

Escenario: múltiples versiones disponibles

Imagine el siguiente escenario: algún equipo descubre un error en el código de infraestructura común (en nuestro caso, Abseil o Guava o similar). En lugar de arreglarlo en su lugar, el equipo decide bifurcar esa infraestructura y modificarla para solucionar el error, sin cambiar el nombre de la biblioteca o los símbolos. Informa a otros equipos cercanos a ellos: "Oye, tenemos una versión mejorada de Abseil registrada aquí: échale un vistazo". Algunos otros equipos crean bibliotecas que se basan en esta nueva bifurcación.

Como veremos en [capítulo 21](#), ahora estamos en una situación peligrosa. Si algún proyecto en el código base llega a depender tanto de la versión original como de la versión bifurcada de Abseil simultáneamente, en el mejor de los casos, la compilación falla. En el peor de los casos, estaremos sujetos a errores de tiempo de ejecución difíciles de entender derivados de la vinculación de dos versiones no coincidentes de la misma biblioteca. La "bifurcación" ha agregado efectivamente una propiedad de coloreado/partición al código base: el conjunto de dependencias transitivas para cualquier destino dado debe incluir exactamente una copia de esta biblioteca. Cualquier enlace agregado desde la partición de "sabor original" de la base de código a la partición de "nueva bifurcación" probablemente romperá las cosas. Esto significa que, al final, algo tan simple como "agregar una nueva dependencia" se convierte en una operación que podría requerir ejecutar todas las pruebas para todo el código base, para asegurarnos de que no hemos violado uno de estos requisitos de partición. Eso es costoso, desafortunado y no escala bien.

9 Por ejemplo, durante una operación de actualización, puede haber dos versiones registradas, pero si un desarrollador está agregando una nueva dependencia en un paquete existente, no debería haber *elegido* qué versión depender.

10 Dicho esto, fallamos en esto en muchos casos porque los paquetes externos a veces tienen copias fijadas de sus propios dependencias incluidas en su versión de origen. Puede leer más sobre cómo todo esto sale mal en [capítulo 21](#).

En algunos casos, es posible que podamos piratear cosas para permitir que un ejecutable resultante funcione correctamente. Java, por ejemplo, tiene una práctica relativamente estándar llamada *sombreado*, que modifica los nombres de las dependencias internas de una biblioteca para ocultar esas dependencias del resto de la aplicación. Cuando se trata de funciones, esto es técnicamente sólido, incluso si teóricamente es un truco. Cuando se trata de tipos que se pueden pasar de un paquete a otro, las soluciones de sombreado no funcionan ni en la teoría ni en la práctica. Por lo que sabemos, cualquier truco tecnológico que permita que múltiples versiones aisladas de una biblioteca funcionen en el mismo binario comparte esta limitación: ese enfoque funcionará para las funciones, pero no existe una solución buena (eficiente) para los tipos de sombreado: múltiples las versiones de cualquier biblioteca que proporcione un tipo de vocabulario (o cualquier construcción de nivel superior) fallarán. El sombreado y los enfoques relacionados solucionan el problema subyacente: se necesitan varias versiones de la misma dependencia.[capítulo 21.](#)

Cualquier sistema de políticas que permita múltiples versiones en el mismo código base está permitiendo la posibilidad de estas costosas incompatibilidades. Es posible que se salga con la suya durante un tiempo (ciertamente tenemos una serie de pequeñas violaciones de esta política), pero en general, cualquier situación de múltiples versiones tiene una posibilidad muy real de conducir a grandes problemas.

La regla de la “versión única”

Con ese ejemplo en mente, además del modelo de fuente única de la verdad, podemos comprender la profundidad de esta regla aparentemente simple para el control de fuentes y la gestión de sucursales:

Los desarrolladores nunca deben tener la opción de "¿De qué versión de este componente debo depender?"

Coloquialmente, esto se convierte en algo así como una "regla de una versión". En la práctica, "One-Version" no es difícil y rápido,¹¹ pero expresando esto alrededor de limitar las versiones que pueden ser elegidas al agregar una nueva dependencia transmite una comprensión muy poderosa.

Para un desarrollador individual, la falta de elección puede parecer un impedimento arbitrario. Sin embargo, vemos una y otra vez que para una organización, es un componente crítico en el escalamiento eficiente. La consistencia tiene una profunda importancia en todos los niveles de una organización. Desde una perspectiva, este es un efecto secundario directo de las discusiones sobre la consistencia y garantizar la capacidad de aprovechar los "puntos de estrangulamiento" consistentes.

11 Por ejemplo, si hay bibliotecas externas/de terceros que se actualizan periódicamente, podría ser inviable actualice esa biblioteca y actualice todo el uso de ella en un solo cambio atómico. Como tal, a menudo es necesario agregar una nueva versión de esa biblioteca, evitar que los nuevos usuarios agreguen dependencias a la anterior y cambiar gradualmente el uso de la antigua a la nueva.

(Casi) Sin ramas de larga duración

Hay varias ideas y políticas más profundas implícitas en nuestra regla de una versión; la más importante de ellas: las ramas de desarrollo deben ser mínimas o, en el mejor de los casos, de muy corta duración. Esto se deriva de una gran cantidad de trabajos publicados en los últimos 20 años, desde procesos ágiles hasta resultados de investigación de DORA sobre desarrollo basado en troncales e incluso el Proyecto Phoenix.¹² lecciones sobre "reducir el trabajo en curso". Cuando incluimos la idea de trabajo pendiente como similar a una rama de desarrollo, esto refuerza aún más que el trabajo debe realizarse en pequeños incrementos contra el tronco, comprometido regularmente.

Como contrapunto: en una comunidad de desarrollo que depende en gran medida de las ramas de desarrollo de larga duración, no es difícil imaginar que la oportunidad de elegir vuelva a aparecer.

Imagine este escenario: un equipo de infraestructura está trabajando en un nuevo Widget, mejor que el anterior. La emoción crece. Otros proyectos recién iniciados preguntan: "¿Podemos depender de su nuevo widget?" Obviamente, esto se puede manejar si ha invertido en políticas de visibilidad de base de código, pero el problema profundo ocurre cuando el nuevo Widget está "permitido" pero solo existe en una rama paralela. Recuerde: el nuevo desarrollo no debe tener una opción al agregar una dependencia. Ese nuevo Widget debe estar comprometido con el tronco, deshabilitado desde el tiempo de ejecución hasta que esté listo y oculto para otros desarrolladores por visibilidad si es posible, o las dos opciones de Widget deben diseñarse de manera que puedan coexistir, vinculadas a los mismos programas.

Curiosamente, ya hay evidencia de que esto es importante en la industria. En Accelerate y en los informes State of DevOps más recientes, DORA señala que existe una relación predictiva entre el desarrollo basado en enlaces troncales y las organizaciones de software de alto rendimiento. Google no es la única organización que descubrió esto ni necesariamente teníamos en mente los resultados esperados cuando evolucionaron estas políticas.

– Parecía que nada más funcionaba. El resultado de DORA ciertamente coincide con nuestra experiencia.

Nuestras políticas y herramientas para cambios a gran escala (LSC; ver [capítulo 22](#)) ponen un peso adicional en la importancia del desarrollo basado en troncales: los cambios amplios/superficiales que se aplican en la base de código ya son una tarea enorme (a menudo tediosa) al modificar todo lo registrado en la rama troncal. Tener un número ilimitado de ramas de desarrollo adicionales que podrían necesitar ser refactorizadas al mismo tiempo sería un impuesto terriblemente alto para ejecutar ese tipo de cambios, encontrando un conjunto cada vez mayor de ramas ocultas. En un modelo DVCS, es posible que ni siquiera sea posible identificar todas esas ramas.

12 Kevin Behr, Gene Kim y George Spafford, *El Proyecto Fénix* (Portland: IT Revolution Press, 2018).

Por supuesto, nuestra experiencia no es universal. Es posible que se encuentre en situaciones inusuales que requieran ramas de desarrollo de mayor duración en paralelo (y fusionadas regularmente con) troncal.

Esos escenarios deben ser raros y debe entenderse que son costosos. De los aproximadamente 1000 equipos que trabajan en el monorepo de Google, solo hay un par que tiene una rama de desarrollo de este tipo.¹³ Por lo general, estos existen por una razón muy específica (y muy inusual). La mayoría de esas razones se reducen a alguna variación de "Tenemos un requisito inusual de compatibilidad con el tiempo". A menudo, se trata de garantizar la compatibilidad de los datos en reposo entre versiones: los lectores y escritores de algún formato de archivo deben ponerse de acuerdo sobre ese formato a lo largo del tiempo, incluso si se modifican las implementaciones del lector o escritor. Otras veces, las ramas de desarrollo de larga duración pueden provenir de la promesa de compatibilidad de API a lo largo del tiempo, cuando One Version no es suficiente y debemos prometer que una versión anterior de un cliente de microservicio aún funciona con un servidor más nuevo (o viceversa). Ese puede ser un requisito muy desafiante, algo que no debe prometer a la ligera para una API en constante evolución. y algo que debe tratar con cuidado para asegurarse de que el período de tiempo no comience a crecer accidentalmente. La dependencia a lo largo del tiempo en cualquier forma es mucho más costosa y complicada que el código que no varía en el tiempo. Internamente, los servicios de producción de Google hacen relativamente pocas promesas de esa forma.¹⁴

También nos beneficiamos enormemente de un tope en el posible sesgo de versión impuesto por nuestro "horizonte de construcción": cada trabajo en producción necesita ser reconstruido y redistribuido cada seis meses, como máximo. (Por lo general, es mucho más frecuente que eso).

Estamos seguros de que hay otras situaciones que podrían necesitar ramas de desarrollo de larga duración. Solo asegúrate de mantenerlos raros. Si adopta otras herramientas y prácticas discutidas en este libro, muchas tenderán a ejercer presión contra ramas de desarrollo de larga duración. La automatización y las herramientas que funcionan muy bien en el tronco y fallan (o requieren más esfuerzo) para una rama de desarrollo pueden ayudar a alentar a los desarrolladores a mantenerse actualizados.

¿Qué pasa con las ramas de lanzamiento?

Muchos equipos de Google usan ramas de lanzamiento, con selecciones de cereza limitadas. Si va a publicar un lanzamiento mensual y continuar trabajando hacia el próximo lanzamiento, es perfectamente razonable crear una rama de lanzamiento. De manera similar, si va a enviar dispositivos a los clientes, es valioso saber exactamente qué versión está disponible "en el campo". Sea cauteloso y sensato, mantenga los picos de cereza al mínimo y no planee volver a fusionarse con el baúl. Nuestros diversos equipos tienen todo tipo de políticas sobre las ramas de publicación dado que relativamente pocos equipos han llegado al tipo de cadencia de publicación rápida prometida por CD (ver [Capítulo 24](#)) que obvia la necesidad o el deseo de una rama de liberación. Generalmente hablando,

¹³ Es difícil obtener un recuento preciso, pero la cantidad de estos equipos es casi seguro menos de 10. ¹⁴ Las interfaces en la nube son una historia diferente.

Las sucursales de lanzamiento no causan ningún costo generalizado en nuestra experiencia. O, al menos, ningún costo perceptible por encima y más allá del costo inherente adicional al VCS.

Monorepos

En 2016, publicamos un artículo (muy citado, muy discutido) sobre el enfoque monorepo de Google.¹⁵ El enfoque monorepo tiene algunos beneficios inherentes, y el principal de ellos es que adherirse a One Version es trivial: generalmente es más difícil violar One Version que hacer lo correcto. No hay un proceso para decidir qué versiones de algo son oficiales, o descubrir qué repositorios son importantes. Herramientas de construcción para entender el estado de la construcción (ver [capítulo 23](#)) tampoco requiere descubrir dónde existen repositorios importantes. La coherencia ayuda a aumentar el impacto de la introducción de nuevas herramientas y optimizaciones. En general, los ingenieros pueden ver lo que hacen los demás y usarlo para informar sus propias elecciones en el código y el diseño del sistema. Todas estas son cosas muy buenas.

Dado todo eso y nuestra creencia en los méritos de la regla de una versión, es razonable preguntar si un monorepo es el único camino verdadero. En comparación, la comunidad de código abierto parece funcionar bien con un enfoque de "muchos repositorios" basado en un número aparentemente infinito de repositorios de proyectos no coordinados y no sincronizados.

En resumen: no, no creemos que el enfoque monorepo tal como lo hemos descrito sea la respuesta perfecta para todos. Continuando con el paralelo entre el formato del sistema de archivos y VCS, es fácil imaginarse decidir entre usar 10 unidades para proporcionar un sistema de archivos lógico muy grande o 10 sistemas de archivos más pequeños a los que se accede por separado. En un mundo de sistemas de archivos, hay pros y contras para ambos. Los problemas técnicos al evaluar la elección del sistema de archivos varían desde la resistencia a interrupciones, las restricciones de tamaño, las características de rendimiento, etc. Los problemas de usabilidad probablemente se centren más en la capacidad de hacer referencia a archivos a través de los límites del sistema de archivos, agregar enlaces simbólicos y sincronizar archivos.

Un conjunto de cuestiones muy similar rige si se prefiere un monorepo o una colección de repositorios de grano más fino. Las decisiones específicas de cómo almacenar su código fuente (o almacenar sus archivos, para el caso) son fácilmente discutibles y, en algunos casos, los detalles de su organización y su flujo de trabajo serán más importantes que otros. Estas son decisiones que tendrá que tomar usted mismo.

Lo importante no es si nos centramos en monorepo; es adherirse al principio de One-Version en la mayor medida posible: los desarrolladores no deben tener una *elección* al agregar una dependencia a alguna biblioteca que ya está en uso en la organización. Las violaciones de elección de la regla de una versión conducen a discusiones de estrategia de fusión, dependencias de diamantes, trabajo perdido y esfuerzo desperdiciado.

15 Rachel Potvin y Josh Levenberg, "¿Por qué Google almacena miles de millones de líneas de código en un solo repositorio?" *Comunicaciones de la ACM*, 59 núm. 7 (2016): 78-87.

Las herramientas de ingeniería de software, que incluyen VCS y sistemas de compilación, brindan cada vez más mecanismos para combinar de manera inteligente entre repositorios detallados y monorepos para brindar una experiencia similar a monorepo: un orden acordado de confirmaciones y comprensión del gráfico de dependencia. Los submódulos de Git, Bazel con dependencias externas y los subproyectos de CMake permiten a los desarrolladores modernos sintetizar algo que se aproxime débilmente al comportamiento de monorepo sin los costos y las desventajas de un monorepo.¹⁶ Por ejemplo, los repositorios de grano fino son más fáciles de manejar en términos de escala (Git a menudo tiene problemas de rendimiento después de unos pocos millones de confirmaciones y tiende a ser lento para clonar cuando los repositorios incluyen artefactos binarios grandes) y almacenamiento (los metadatos de VCS pueden sumarse, especialmente si tiene artefactos binarios en su sistema de control de versiones). Los repositorios de granularidad fina en un repositorio de estilo federado/virtual-monorepo (VMR) pueden facilitar el aislamiento de proyectos experimentales o de alto secreto sin dejar de mantener una versión y permitir el acceso a utilidades comunes.

En otras palabras: si todos los proyectos de su organización tienen los mismos requisitos de confidencialidad, legales, de privacidad y de seguridad,¹⁷ un verdadero monorepo es una buena manera de hacerlo. De lo contrario, apuntar para la funcionalidad de un monorepo, pero permítase la flexibilidad de implementar esa experiencia de una manera diferente. Si puede manejar repositorios separados y adherirse a One Version o su carga de trabajo está lo suficientemente desconectada como para permitir repositorios realmente separados, genial. De lo contrario, sintetizar algo como un VMR de alguna manera puede representar lo mejor de ambos mundos.

Después de todo, su elección de formato de sistema de archivos realmente no importa tanto como lo que escriba en él.

Futuro del control de versiones

Google no es la única organización que discute públicamente los beneficios de un enfoque monorepo. Microsoft, Facebook, Netflix y Uber también han mencionado públicamente su confianza en el enfoque. DORA ha publicado al respecto extensamente. Es vagamente posible que todas estas empresas exitosas y longevas estén equivocadas, o al menos que sus situaciones sean lo suficientemente diferentes como para ser inaplicables a la organización más pequeña promedio. Aunque es posible, creemos que es poco probable.

La mayoría de los argumentos en contra de los monorepos se centran en las limitaciones técnicas de tener un solo repositorio grande. Si clonar un repositorio desde el origen es rápido y económico, es más probable que los desarrolladores mantengan los cambios pequeños y aislados (y que eviten hacer

16 No creemos que hayamos visto nada que haga esto particularmente bien, pero las dependencias entre repositorios/virtuales La idea de monorepo está claramente en el aire.

17 O tiene la voluntad y la capacidad de personalizar su VCS y mantener esa personalización para el vida útil de su base de código/organización. Por otra parte, tal vez no planees eso como una opción; eso es un montón de gastos generales.

errores al comprometerse con la rama incorrecta de trabajo en curso). Si clonar un repositorio (o realizar alguna otra operación común de VCS) requiere horas de tiempo de desarrollador desperdiciado, puede ver fácilmente por qué una organización evitaría confiar en un repositorio/operación tan grande. Afortunadamente, evitamos este escollo al centrarnos en proporcionar un VCS que escala de forma masiva.

Mirando los últimos años de importantes mejoras en Git, claramente se está haciendo mucho trabajo para admitir repositorios más grandes: clones superficiales, ramas dispersas, mejor optimización y más. Esperamos que esto continúe y que disminuya la importancia de “pero debemos mantener el depósito pequeño”.

El otro argumento importante en contra de monorepos es que no coincide con la forma en que ocurre el desarrollo en el mundo del software de código abierto (OSS). Aunque es cierto, muchas de las prácticas en el mundo OSS provienen (con razón) de priorizar la libertad, la falta de coordinación y la falta de recursos informáticos. Los proyectos separados en el mundo OSS son organizaciones efectivamente separadas que resultan ser capaces de ver el código de los demás. Dentro de los límites de una organización, podemos hacer más suposiciones: podemos asumir la disponibilidad de los recursos informáticos, podemos asumir la coordinación y podemos suponer que existe cierta cantidad de autoridad centralizada.

Una preocupación menos común pero quizás más legítima con el enfoque monorepo es que a medida que su organización crece, es cada vez menos probable que cada pieza de código esté sujeta exactamente a los mismos requisitos legales, de cumplimiento, reglamentarios, secretos y de privacidad. Una ventaja nativa de un enfoque manyrepo es que los repositorios separados son obviamente capaces de tener diferentes conjuntos de desarrolladores autorizados, visibilidad, permisos, etc. Se puede unir esa característica en un monorepo, pero implica algunos costos continuos de mantenimiento en términos de personalización y mantenimiento.

Al mismo tiempo, la industria parece estar inventando enlaces ligeros entre repositorios una y otra vez. A veces, esto está en el VCS (submódulos de Git) o en el sistema de compilación. Siempre que una colección de repositorios tenga una comprensión coherente de “qué es el troncal”, “qué cambio ocurrió primero” y los mecanismos para describir las dependencias, podemos imaginar fácilmente unir una colección dispar de repositorios físicos en un VMR más grande. Si bien Piper lo ha hecho muy bien para nosotros, invertir en un VMR de gran escalabilidad y herramientas para administrarlo y confiar en la personalización lista para usar para los requisitos de políticas por repositorio podría haber sido una mejor inversión.

Tan pronto como alguien crea una cantidad suficientemente grande de proyectos compatibles e interdependientes en la comunidad de OSS y publica una vista VMR de esos paquetes, sospechamos que las prácticas de los desarrolladores de OSS comenzarán a cambiar. Vemos destellos de esto en las herramientas que *podrías* sintetizar un monorepo virtual así como en el trabajo realizado por (por ejemplo) grandes distribuciones de Linux descubriendo y publicando revisiones mutuamente compatibles de miles de paquetes. Con pruebas unitarias, CI y actualización automática de versiones para nuevos envíos a una de esas revisiones, lo que permite que el propietario de un paquete

actualizar el baúl para su paquete (de manera continua, por supuesto), creemos que el modelo se pondrá de moda en el mundo del código abierto. Después de todo, es solo una cuestión de eficiencia: un enfoque monorepo (virtual) con una regla de una versión reduce la complejidad del desarrollo de software en una dimensión completa (difícil): el tiempo.

Esperamos que el control de versiones y la gestión de dependencias evolucionen en esta dirección en los próximos 10 a 20 años: los VCS se centrarán en permitiendo repositorios más grandes con una mejor escala de rendimiento, pero también eliminando la necesidad de repositorios más grandes al proporcionar mejores mecanismos para unirlos a través de los límites del proyecto y la organización. Alguien, tal vez los grupos de gestión de paquetes existentes o los distribuidores de Linux, catalizarán un monorepo virtual estándar de facto. Dependiendo de las utilidades en ese monorepo, proporcionará un fácil acceso a un conjunto compatible de dependencias como una sola unidad. En general, reconoceremos que los números de versión son marcas de tiempo, y que permitir el sesgo de versión agrega una complejidad de dimensionalidad (tiempo) que cuesta mucho y que podemos aprender a evitar. Comienza con algo lógicamente como un monorepo.

Conclusión

Los sistemas de control de versiones son una extensión natural de los desafíos y oportunidades de colaboración que brinda la tecnología, especialmente los recursos informáticos compartidos y las redes informáticas. Históricamente han evolucionado al unísono con las normas de la ingeniería de software tal como las entendemos en ese momento.

Los primeros sistemas proporcionaban un bloqueo de granularidad de archivos simplista. A medida que los proyectos y equipos típicos de ingeniería de software crecieron, los problemas de escalamiento con ese enfoque se hicieron evidentes y nuestra comprensión del control de versiones cambió para adaptarse a esos desafíos. Luego, a medida que el desarrollo avanzaba cada vez más hacia un modelo OSS con contribuyentes distribuidos, los VCS se volvieron más descentralizados. Esperamos un cambio en la tecnología VCS que asuma una disponibilidad constante de la red, centrándose más en el almacenamiento y la creación en la nube para evitar la transmisión de archivos y artefactos innecesarios. Esto es cada vez más crítico para los proyectos de ingeniería de software grandes y duraderos, incluso si significa un cambio de enfoque en comparación con los proyectos de programación simples de un solo desarrollador/una sola máquina. Este cambio a la nube concretará lo que ha surgido con los enfoques de DVCS:

La descentralización actual de DVCS es una reacción sensata de la tecnología a las necesidades de la industria (especialmente la comunidad de código abierto). Sin embargo, la configuración de DVCS debe controlarse estrictamente y combinarse con políticas de administración de sucursales que tengan sentido para su organización. A menudo, también puede presentar problemas de escala inesperados: la operación fuera de línea de fidelidad perfecta requiere muchos más datos locales. El hecho de no controlar la complejidad potencial de una ramificación libre para todos puede conducir a una cantidad potencialmente ilimitada de gastos generales entre los desarrolladores y la implementación de ese código. Sin embargo, la tecnología compleja no necesita ser utilizada de manera compleja: como

Como vemos en los modelos de desarrollo basados en monorrepos y troncales, mantener las políticas de las sucursales simples generalmente conduce a mejores resultados de ingeniería.

La elección conduce a los costos aquí. Respaldamos encarecidamente la regla de una versión que se presenta aquí: los desarrolladores dentro de una organización no deben tener la opción de elegir dónde comprometerse o de qué versión de un componente existente depender. Hay pocas políticas de las que somos conscientes que puedan tener tal impacto en la organización: aunque puede ser molesto para los desarrolladores individuales, en conjunto, el resultado final es mucho mejor.

TL; DR

- Utilice el control de versiones para cualquier proyecto de desarrollo de software más grande que un "proyecto de juguete con un solo desarrollador que nunca se actualizará".
- **Hay un problema de escala inherente cuando hay opciones en "¿de qué versión debo depender?"**
- Las reglas de una versión son sorprendentemente importantes para la eficiencia organizacional. Eliminar opciones sobre dónde comprometerse o de qué depender puede resultar en una simplificación significativa.
- En algunos idiomas, es posible que pueda hacer un esfuerzo para esquivar esto con enfoques técnicos como el sombreado, la compilación separada, la ocultación del enlazador, etc. El trabajo para hacer que esos enfoques funcionen es trabajo totalmente perdido: sus ingenieros de software no están produciendo nada, solo están solucionando deudas técnicas.
- Investigaciones anteriores (DORA/State of DevOps/Accelerate) han demostrado que el desarrollo basado en troncos es un factor predictivo en organizaciones de desarrollo de alto rendimiento. Las sucursales de desarrollo de larga duración no son un buen plan predeterminado.
- Utilice cualquier sistema de control de versiones que tenga sentido para usted. Si su organización quiere priorizar repositorios separados para proyectos separados, probablemente sea conveniente que las dependencias entre repositorios no estén fijadas/"a la cabeza"/"basadas en el tronco". Hay un número cada vez mayor de instalaciones de VCS y sistemas de compilación que le permiten tener repositorios pequeños y detallados, así como una noción principal/troncal "virtual" consistente para toda la organización.

Búsqueda de código

*Escrito por Alexander Neubeck y Ben St. John
Editado por Lisa Carey*

Code Search es una herramienta para navegar y buscar código en Google que consta de una interfaz de usuario de frontend y varios elementos de backend. Como muchas de las herramientas de desarrollo de Google, surgió directamente de la necesidad de escalar al tamaño del código base. Code Search comenzó como una combinación de una herramienta de tipo grep para código interno con la clasificación y la interfaz de usuario de búsqueda de código externo.¹ Su lugar como herramienta clave para los desarrolladores de Google se consolidó con la integración de Kythe/Grok,² que agregó referencias cruzadas y la capacidad de saltar a definiciones de símbolos.

Esa integración cambió su enfoque de la búsqueda al código de exploración, y el desarrollo posterior de Code Search se guió en parte por el principio de "responder a la siguiente pregunta sobre el código con un solo clic". Ahora, preguntas como "¿Dónde se define este símbolo?", "¿Dónde se usa?", "¿Cómo lo incluyo?", "¿Cuándo se agregó a la base de código?" e incluso algunas como "Toda la flota, ¿Cuántos ciclos de CPU consume?" son todos responsables con uno o dos clics.

A diferencia de los entornos de desarrollo integrados (IDE) o los editores de código, Code Search está optimizado para el caso de uso de lectura, comprensión y exploración de código a escala. Para hacerlo, depende en gran medida de backends basados en la nube para buscar contenido y resolver referencias cruzadas.

¹ GSearch se ejecutó originalmente en la computadora personal de Jeff Dean, lo que una vez causó angustia en toda la empresa cuando ¡Se fue de vacaciones y lo cerraron!

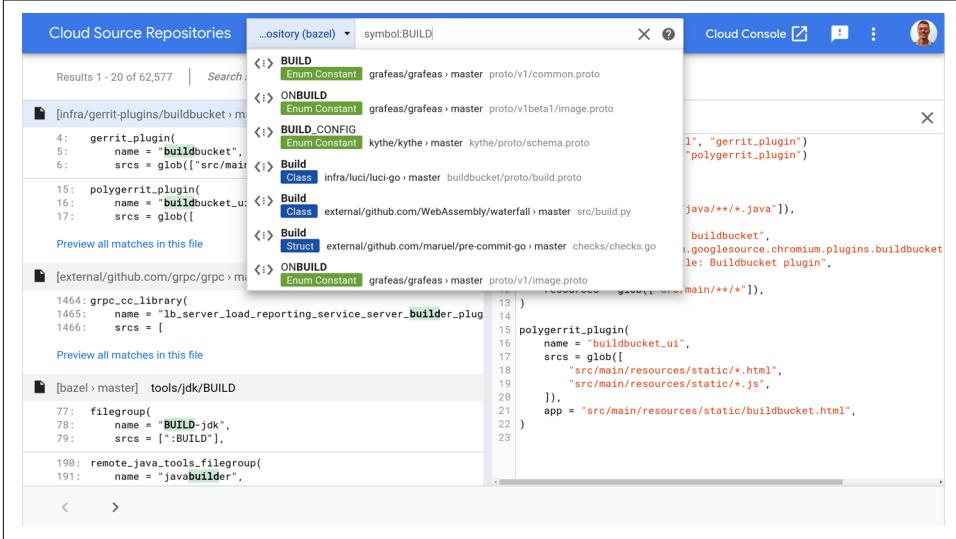
² Cerrado en 2013; ver https://en.wikipedia.org/wiki/Google_Code_Search.

³ Ahora conocido como Kythe, un servicio que proporciona referencias cruzadas (entre otras cosas): los usos de un determinado símbolo de código, por ejemplo, una función, utilizando la información de compilación completa para eliminar la ambigüedad de otros con el mismo nombre.

En este capítulo, veremos la búsqueda de código con más detalle, incluida la forma en que los Googlers la usan como parte de sus flujos de trabajo de desarrollador, por qué elegimos desarrollar una herramienta web separada para la búsqueda de código y examinaremos cómo aborda los desafíos de buscar y código de navegación a escala del repositorio de Google.

La interfaz de usuario de búsqueda de código

El cuadro de búsqueda es un elemento central de la interfaz de usuario de Code Search (ver Figura 17-1), y al igual que la búsqueda web, tiene "sugerencias" que los desarrolladores pueden usar para navegar rápidamente a archivos, símbolos o directorios. Para casos de uso más complejos, se devuelve una página de resultados con fragmentos de código. La búsqueda en sí misma se puede considerar como una "búsqueda en archivos" instantánea (como el Unix grep comando) con clasificación de relevancia y algunas mejoras específicas del código, como el resaltado de sintaxis adecuado, el conocimiento del alcance y el conocimiento de los comentarios y los literales de cadena. La búsqueda también está disponible desde la línea de comandos y se puede incorporar a otras herramientas a través de una API de llamada a procedimiento remoto (RPC). Esto es útil cuando se requiere procesamiento posterior o si el conjunto de resultados es demasiado grande para la inspección manual.



The screenshot shows the Cloud Source Repositories interface with a search bar at the top containing the query `symbol:BUILD`. Below the search bar, there are two tabs: `repository (bazel)` and `Cloud Console`. The main area displays search results for `symbol:BUILD`, which include:

- `<:> BUILD` [infra/gerrit-plugins/buildbucket..., line 4]
- `<:> ONBUILD` [infra/gerrit..., line 15]
- `<:> BUILD_CONFIG` [kythe/kythe..., line 16]
- `<:> Build` [infra/luci/uci-go..., line 17]
- `<:> Build` [external/github.com/WebAssembly/waterfall..., line 1464]
- `<:> Build` [external/github.com/maruel/pre-commit-go..., line 1465]
- `<:> ONBUILD` [grafeas/grafeas..., line 1466]
- `<:> Build` [bazel..., line 77]
- `<:> filegroup` [bazel..., line 78]
- `<:> remote_java_tools_filegroup` [bazel..., line 190]

Each result entry shows the file path, line number, and code snippet. A preview panel on the right shows the context of the `Build` symbol from the `external/github.com/maruel/pre-commit-go` repository, displaying code related to `polygerrit_plugin`.

Figura 17-1. La interfaz de usuario de búsqueda de código

Al ver un solo archivo, se puede hacer clic en la mayoría de los tokens para que el usuario pueda navegar rápidamente a la información relacionada. Por ejemplo, una llamada de función se vinculará a su definición de función, un nombre de archivo importado al archivo de origen real o una identificación de error en un comentario al informe de error correspondiente. Esto funciona con herramientas de indexación basadas en compiladores como Kythé. Al hacer clic en el nombre del símbolo, se abre un panel con todos los lugares en los que se usa el símbolo. Simi-

Por lo general, pasar el cursor sobre las variables locales en una función resaltará todas las ocurrencias de esa variable en la implementación.

Code Search también muestra el historial de un archivo, a través de su integración con Piper (ver [Capítulo 16](#)). Esto significa ver versiones anteriores del archivo, qué cambios lo han afectado, quién los escribió, saltar a ellos en Crítica (ver[capítulo 19](#)), versiones diferentes de archivos y la vista clásica de "culpa" si lo desea. Incluso los archivos eliminados se pueden ver desde una vista de directorio.

¿Cómo utilizan los Googlers la búsqueda de códigos?

Aunque una funcionalidad similar está disponible en otras herramientas, los Googlers todavía hacen un uso intensivo de la interfaz de usuario de Code Search para buscar y ver archivos y, en última instancia, para comprender el código.⁴ Se puede pensar que las tareas que los ingenieros intentan completar con Code Search responden a preguntas sobre el código, y las intenciones recurrentes se vuelven visibles.⁵

¿Dónde?

Alrededor del 16% de las búsquedas de código intentan responder a la pregunta de dónde existe una información específica en la base de código; por ejemplo, una definición o configuración de función, todos los usos de una API, o simplemente dónde se encuentra un archivo específico en el repositorio. Estas preguntas están muy dirigidas y se pueden responder de manera muy precisa con consultas de búsqueda o siguiendo enlaces semánticos, como "saltar a la definición de símbolo". Estas preguntas a menudo surgen durante tareas más grandes como refactorizaciones/limpiezas o cuando se colabora con otros ingenieros en un proyecto. Por lo tanto, es esencial que estas pequeñas lagunas de conocimiento se aborden de manera eficiente.

Code Search proporciona dos formas de ayudar: clasificar los resultados y un lenguaje de consulta enriquecido. La clasificación aborda los casos comunes y las búsquedas pueden hacerse muy específicas (p. ej., restringir rutas de código, excluir idiomas, solo considerar funciones) para tratar casos más raros.

La interfaz de usuario facilita compartir un resultado de búsqueda de código con colegas. Entonces, para revisiones de código, simplemente puede incluir el enlace, por ejemplo, "¿Ha considerado usar este mapa hash especializado: cool_hash.h?" Esto también es muy útil para la documentación, en los informes de errores y en las autopsias, y es la forma canónica de referirse al código.

⁴ Hay un ciclo virtuoso interesante que fomenta un navegador de código ubicuo: escribir código que sea fácil de navegar. Esto puede significar cosas como no anidar jerarquías demasiado profundas, lo que requiere muchos clics para pasar de los sitios de llamadas a la implementación real, y usar tipos con nombre en lugar de cosas genéricas como cadenas o números enteros, porque entonces es fácil encontrar todos los usos.

⁵ Sadowski, Caitlin, Kathryn T. Stolee y Sebastian Elbaum. "Cómo los desarrolladores buscan código: un caso Estudiar en*Actas de la 10.ª reunión conjunta sobre fundamentos de la ingeniería de software de 2015 (ESEC/FSE 2015)*. <https://doi.org/10.1145/2786805.2786855>.

dentro de Google. Incluso se puede hacer referencia a versiones anteriores del código, por lo que los enlaces pueden seguir siendo válidos a medida que evoluciona la base de código.

¿Qué?

Aproximadamente una cuarta parte de las búsquedas de código son búsquedas clásicas de archivos, para responder a la pregunta de qué está haciendo una parte específica de la base de código. Este tipo de tareas suelen ser más exploratorias, en lugar de localizar un resultado específico. Se trata de usar Code Search para leer el código fuente, para comprender mejor el código antes de realizar un cambio o para poder comprender el cambio de otra persona.

Para facilitar este tipo de tareas, Code Search introdujo la navegación a través de jerarquías de llamadas y una navegación rápida entre archivos relacionados (por ejemplo, entre archivos de encabezado, implementación, prueba y compilación). Se trata de comprender el código respondiendo fácilmente a cada una de las muchas preguntas que tiene un desarrollador al mirarlo.

¿Cómo?

El caso de uso más frecuente (alrededor de un tercio de las búsquedas de código) consiste en ver ejemplos de cómo otros han hecho algo. Por lo general, un desarrollador ya ha encontrado una API específica (p. ej., cómo leer un archivo desde un almacenamiento remoto) y quiere ver cómo se debe aplicar la API a un problema en particular (p. ej., cómo configurar la conexión remota de manera sólida y manejar ciertos tipos de errores). Code Search también se usa para encontrar la biblioteca adecuada para problemas específicos en primer lugar (por ejemplo, cómo calcular una huella digital para valores enteros de manera eficiente) y luego elegir la implementación más adecuada. Para este tipo de tareas, es típica una combinación de búsquedas y exploración de referencias cruzadas.

¿Por qué?

Relacionado con *qué* el código está funcionando, hay más consultas dirigidas por *qué* el código se comporta de manera diferente a lo esperado. Alrededor del 16 % de las búsquedas de código intentan responder a la pregunta de por qué se agregó una determinada pieza de código o por qué se comporta de cierta manera. Tales preguntas surgen a menudo durante la depuración; por ejemplo, ¿por qué ocurre un error en estas circunstancias particulares?

Una capacidad importante aquí es poder buscar y explorar el estado exacto de la base de código en un momento determinado. Al depurar un problema de producción, esto puede significar trabajar con un estado de la base de código que tiene semanas o meses de antigüedad, mientras que la depuración de fallas de prueba para código nuevo generalmente significa trabajar con cambios que tienen solo unos minutos de antigüedad. Ambos son posibles con Code Search.

¿Quién y cuándo?

Alrededor del 8% de las búsquedas de código intentan responder preguntas sobre quién o cuándo alguien introdujo una determinada pieza de código, interactuando con el sistema de control de versiones. Por ejemplo, es posible ver cuándo se introdujo una línea en particular (como la "culpa" de Git) y pasar a la revisión del código correspondiente. Este panel de historial también puede ser muy útil para encontrar a la mejor persona para preguntar sobre el código o para revisar un cambio en él.⁶

¿Por qué una herramienta web independiente?

Fuera de Google, la mayoría de las investigaciones antes mencionadas se realizan dentro de un IDE local. Entonces, ¿por qué otra herramienta más?

Escala

La primera respuesta es que la base de código de Google es tan grande que una copia local de la base de código completa, un requisito previo para la mayoría de los IDE, simplemente no cabe en una sola máquina. Incluso antes de que se supere esta barrera fundamental, la creación de índices de referencia cruzada y búsqueda local para cada desarrollador tiene un costo, un costo que a menudo se paga al inicio del IDE, lo que ralentiza la velocidad del desarrollador. O bien, sin índice, búsquedas puntuales (p. ej., grep) puede volverse dolorosamente lento. Un índice de búsqueda centralizado significa hacer este trabajo una vez, por adelantado, y significa que las inversiones en el proceso benefician a todos. Por ejemplo, el índice de búsqueda de código se actualiza de forma incremental con cada cambio enviado, lo que permite la construcción del índice con un costo lineal.⁷

En la búsqueda web normal, los eventos actuales que cambian rápidamente se mezclan con elementos que cambian más lentamente, como las páginas estables de Wikipedia. La misma técnica se puede extender a la búsqueda de código, haciendo que la indexación sea incremental, lo que reduce su costo y permite que los cambios en la base de código sean visibles para todos al instante. Cuando se envía un cambio de código, solo se deben reiniciar los archivos reales tocados, lo que permite actualizaciones paralelas e independientes del índice global.

Desafortunadamente, el índice de referencias cruzadas no se puede actualizar instantáneamente de la misma manera. La incrementalidad no es posible para él, ya que cualquier cambio de código puede influir potencialmente en todo el código base y, en la práctica, a menudo afecta a miles de archivos. Muchos (casi todos)

⁶ Dicho esto, dada la tasa de confirmaciones de cambios generados por máquinas, el seguimiento ingenuo de "culpa" tiene menos valor que en ecosistemas más reacios al cambio.

⁷ A modo de comparación, el modelo de "cada desarrollador tiene su propio IDE en su propio espacio de trabajo hace la indexación cálculo" se escala aproximadamente de forma cuadrática: los desarrolladores producen una cantidad de código aproximadamente constante por unidad de tiempo, por lo que la base de código se escala de forma lineal (incluso con un número fijo de desarrolladores). Un número lineal de IDE realiza linealmente más trabajo cada vez; esta no es una receta para un buen escalado.

de Google) deben construirse binarios completos⁸(o al menos analizado) para determinar la estructura semántica completa. Utiliza una tonelada de recursos informáticos para producir el índice diariamente (la frecuencia actual). La discrepancia entre el índice de búsqueda instantánea y el índice de referencias cruzadas diarias es una fuente de problemas raros pero recurrentes para los usuarios.

Vista de código global de configuración cero

Ser capaz de navegar de manera instantánea y efectiva por todo el código base significa que es muy fácil encontrar bibliotecas relevantes para reutilizar y buenos ejemplos para copiar. Para los IDE que construyen índices al inicio, existe la presión de tener un proyecto pequeño o un alcance visible para reducir este tiempo y evitar inundar herramientas como el autocompletado con ruido. Con la interfaz de usuario web de Code Search, no se requiere configuración (por ejemplo, descripciones de proyectos, entorno de compilación), por lo que también es muy fácil y rápido aprender sobre el código, dondequiera que ocurra, lo que mejora la eficiencia del desarrollador. Tampoco hay peligro de perder dependencias de código; por ejemplo, al actualizar una API, lo que reduce los problemas de combinación y versiones de la biblioteca.

Especialización

Quizás sorprendentemente, una de las ventajas de Code Search es que es**no**un IDE. Esto significa que la experiencia del usuario (UX) se puede optimizar para explorar y comprender el código, en lugar de editar, que suele ser la mayor parte de un IDE (p. ej., atajos de teclado, menús, clics del mouse e incluso espacio en la pantalla). Por ejemplo, debido a que no hay un cursor de texto del editor, cada clic del mouse en un símbolo puede tener sentido (por ejemplo, mostrar todos los usos o saltar a la definición), en lugar de ser una forma de mover el cursor. Esta ventaja es tan grande que es muy común que los desarrolladores tengan varias pestanas de búsqueda de código abiertas al mismo tiempo que su editor.

Integración con otras herramientas de desarrollo

Debido a que es la forma principal de ver el código fuente, Code Search es la plataforma lógica para exponer información sobre el código fuente. Libera a los creadores de herramientas de la necesidad de crear una interfaz de usuario para sus resultados y garantiza que toda la audiencia de desarrolladores conozca su trabajo sin necesidad de anunciarlo. Muchos análisis se ejecutan regularmente en todo el código base de Google, y sus resultados generalmente aparecen en Code Search. Para

⁸Kythéinstrumenta el flujo de trabajo de construcción para extraer nodos y bordes semánticos del código fuente. Esta extracción El proceso recopila gráficos de referencias cruzadas parciales para cada regla de compilación individual. En una fase posterior, estos gráficos parciales se fusionan en un gráfico global y su representación se optimiza para las consultas más comunes (ir a definición, buscar todos los usos, buscar todas las decoraciones para un archivo). Cada fase: extracción y procesamiento posterior — es aproximadamente tan caro como una construcción completa; por ejemplo, en el caso de Chromium, la construcción del índice Kythe se realiza en aproximadamente seis horas en una configuración distribuida y, por lo tanto, es demasiado costoso para que cada desarrollador lo construya en su propia estación de trabajo. Este costo computacional es la razón por la cual el índice de Kythe se calcula solo una vez al día.

Por ejemplo, para muchos idiomas, podemos detectar código "muerto" (no llamado) y marcarlo como tal cuando se examina el archivo.

En la otra dirección, el enlace de búsqueda de código a un archivo fuente se considera su "ubicación" canónica. Esto es útil para muchas herramientas de desarrollo (ver [Figura 17-2](#)). Por ejemplo, las líneas del archivo de registro normalmente contienen el nombre del archivo y el número de línea de la declaración de registro. El visor de registro de producción utiliza un enlace de búsqueda de código para conectar la instrucción de registro con el código de producción. Dependiendo de la información disponible, esto puede ser un enlace directo a un archivo en una revisión específica o una búsqueda básica de nombre de archivo con el número de línea correspondiente. Si solo hay un archivo coincidente, se abre en el número de línea correspondiente. De lo contrario, se representan fragmentos de la línea deseada en cada uno de los archivos coincidentes.

Figura 17-2. Integración de Code Search en un visor de registros

De manera similar, los marcos de pila están vinculados al código fuente, ya sea que se muestren dentro de una herramienta de informes de fallas o en la salida del registro, como se muestra en [Figura 17-3](#).

Dependiendo del lenguaje de programación, el enlace utilizará una búsqueda de nombre de archivo o símbolo. Debido a que se conoce la instantánea del repositorio en el que se creó el binario bloqueado, la búsqueda puede restringirse exactamente a esta versión. De esa manera, los enlaces siguen siendo válidos durante un largo período de tiempo, incluso si el código correspondiente se refactoriza o elimina más tarde.



Figura 17-3. Integración de búsqueda de código en marcos de pila

Los errores de compilación y las pruebas también suelen hacer referencia a una ubicación de código (p. ej., prueba X en *expedientealinea*). Estos se pueden vincular incluso para el código no enviado dado que la mayor parte del desarrollo ocurre en espacios de trabajo específicos visibles en la nube a los que se puede acceder y buscar mediante Code Search.

Finalmente, los codelabs y otra documentación se refieren a API, ejemplos e implementaciones. Dichos enlaces pueden ser consultas de búsqueda que hacen referencia a una clase o función específica, que siguen siendo válidas cuando cambia la estructura del archivo. Para fragmentos de código, la implementación más reciente en el encabezado se puede incrustar fácilmente en una página de documentación, como se demuestra en Figura 17-4, sin necesidad de contaminar el archivo fuente con marcadores de documentación adicionales.

Use a [markdown code block](#) and specify `live-snippet` as the language. Live snippets use the Code Search `cs/` query syntax to specify which code to include. For example:

```
```live-snippet
cs/file:google3/corp/g3doc/tests/regression_tests/testLiveSnippets/snippet_test.cc f
```

```

Which renders as:

```
static int Fibonacci(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1;
    for (int i = 0; i < n; ++i) {
        int t = a;
        a = b;
        b += t;
    }
    return b;
}
```

Figura 17-4. Integración de Code Search en la documentación

Exposición API

Code Search expone sus API de búsqueda, referencia cruzada y resaltado de sintaxis a las herramientas, por lo que los desarrolladores de herramientas pueden incorporar esas capacidades en sus herramientas sin necesidad de volver a implementarlas. Además, se han escrito complementos para proporcionar búsquedas y referencias cruzadas a editores e IDE como vim, emacs e IntelliJ. Estos complementos restauran parte de la energía perdida debido a la imposibilidad de indexar localmente el código base y devuelven algo de productividad al desarrollador.

Impacto de la escala en el diseño

En la sección anterior, analizamos varios aspectos de la interfaz de usuario de Code Search y por qué vale la pena tener una herramienta separada para buscar código. En las siguientes secciones, miramos un poco detrás de escena de la implementación. Primero analizamos el desafío principal, la escalabilidad, y luego algunas de las formas en que la gran escala complica la creación de un buen producto para buscar y examinar código. Después de eso, detallamos cómo abordamos algunos de esos desafíos y qué compensaciones se hicieron al crear Code Search.

el mas grande: El desafío de escalado para buscar código es el tamaño del corpus. Para un pequeño repositorio de un par de megabytes, una búsqueda de fuerza bruta congrue la búsqueda servirá. Cuando es necesario buscar cientos de megabytes, un índice local simple puede acelerar la búsqueda en un orden de magnitud o más. Cuando es necesario buscar gigabytes o terabytes de código fuente, una solución alojada en la nube con varias máquinas puede mantener tiempos de búsqueda razonables. La utilidad de una solución central aumenta con la cantidad de desarrolladores que la utilizan y el tamaño del espacio de código.

Latencia de consulta de búsqueda

Aunque damos por hecho que una interfaz de usuario rápida y con capacidad de respuesta es mejor para el usuario, la baja latencia no es gratuita. Para justificar el esfuerzo, se puede comparar con el tiempo de ingeniería ahorrado en todos los usuarios. Dentro de Google, procesamos mucho más de un millón de consultas de búsqueda de desarrolladores dentro de Code Search *por día*. Para un millón de consultas, un aumento de solo un segundo por solicitud de búsqueda corresponde a unos 35 ingenieros inactivos a tiempo completo todos los días. Por el contrario, el motor de búsqueda se puede construir y mantener con aproximadamente una décima parte de estos ingenieros. Esto significa que con alrededor de 100.000 consultas por día (correspondientes a menos de 5.000 desarrolladores), solo el argumento de latencia de un segundo es una especie de punto de equilibrio.

En realidad, la pérdida de productividad no aumenta linealmente con la latencia. **Se considera que una IU responde si las latencias son inferiores a 200 ms.** Pero después de solo un segundo, la atención del desarrollador a menudo comienza a desviarse. Si pasan otros 10 segundos, el revelador es

⁹ Debido a que las consultas son independientes, se pueden atender más usuarios al tener más servidores.

probable que cambie de contexto por completo, lo que generalmente se reconoce que tiene altos costos de productividad. La mejor manera de mantener a un desarrollador en un estado de "flujo" productivo es apuntar a una latencia de extremo a extremo inferior a 200 ms para todas las operaciones frecuentes e invertir en los backends correspondientes.

Se realiza una gran cantidad de consultas de búsqueda de código para navegar por la base de código. Idealmente, el archivo "siguiente" está a solo un clic de distancia (por ejemplo, para archivos incluidos o definiciones de símbolos), pero para la navegación general, en lugar de usar el árbol de archivos clásico, puede ser mucho más rápido buscar simplemente el archivo deseado. archivo o símbolo, idealmente sin necesidad de especificarlo completamente, y se proporcionan sugerencias para texto parcial. Esto se vuelve cada vez más cierto a medida que crece la base de código (y el árbol de archivos).

La navegación normal a un archivo específico en otra carpeta o proyecto requiere varias interacciones del usuario. Con la búsqueda, solo un par de pulsaciones de teclas pueden ser suficientes para llegar al archivo correspondiente. Para que la búsqueda sea efectiva, se puede proporcionar información adicional sobre el contexto de búsqueda (por ejemplo, el archivo visualizado actualmente) al motor de búsqueda. El contexto puede restringir la búsqueda a archivos de un proyecto específico o influir en la clasificación al preferir archivos que están cerca de otros archivos o directorios. En la interfaz de usuario de búsqueda de código,¹⁰ el usuario puede predefinir múltiples contextos y cambiar rápidamente entre ellos según sea necesario. En los editores, los archivos abiertos o editados se utilizan implícitamente como contexto para priorizar los resultados de búsqueda en su proximidad.

Uno podría considerar el poder del lenguaje de consulta de búsqueda (por ejemplo, especificar archivos, usar expresiones regulares) como otro criterio; discutimos esto en la sección de compensaciones un poco más adelante en el capítulo.

Índice de latencia

La mayoría de las veces, los desarrolladores no se dan cuenta cuando los índices están desactualizados. Solo se preocupan por un pequeño subconjunto de código, e incluso por eso generalmente no lo harán. *sabersi* hay un código más reciente. Sin embargo, para los casos en los que escribieron o revisaron el cambio correspondiente, estar desincronizados puede causar mucha confusión. Suele no importar si el cambio fue una pequeña corrección, una refactorización o una pieza de código completamente nueva: los desarrolladores simplemente esperan una vista coherente, como la que experimentan en su IDE para un proyecto pequeño.

Al escribir código, se espera la indexación instantánea del código modificado. Cuando se agregan nuevos archivos, funciones o clases, no poder encontrarlos es frustrante y rompe el flujo de trabajo normal para los desarrolladores acostumbrados a perfeccionar las referencias cruzadas. Otro ejemplo son las refactorizaciones basadas en buscar y reemplazar. No solo es más conveniente cuando el código eliminado desaparece inmediatamente de los resultados de búsqueda, sino que también es esencial que las refactorizaciones posteriores tengan en cuenta el nuevo estado. al trabajar con

¹⁰ La interfaz de usuario de Code Search también tiene un árbol de archivos clásico, por lo que también es posible navegar de esta manera.

un VCS centralizado, un desarrollador podría necesitar una indexación instantánea para el código enviado si el cambio anterior ya no forma parte del conjunto de archivos modificado localmente.

Por el contrario, a veces es útil poder retroceder en el tiempo a una instantánea anterior del código; en otras palabras, una liberación. Durante un incidente, una discrepancia entre el índice y el código en ejecución puede ser especialmente problemática porque puede ocultar causas reales o introducir distracciones irrelevantes. Este es un problema para las referencias cruzadas porque la tecnología actual para construir un índice a la escala de Google simplemente toma horas y la complejidad significa que solo se mantiene una "versión" del índice. Aunque se pueden realizar algunos parches para alinear el código nuevo con un índice anterior, este es un problema que aún debe resolverse.

Implementación de Google

La implementación particular de Code Search de Google se adapta a las características únicas de su base de código, y la sección anterior describió nuestras restricciones de diseño para crear un índice robusto y receptivo. La siguiente sección describe cómo el equipo de Code Search implementó y lanzó su herramienta a los desarrolladores de Google.

Índice de búsqueda

El código base de Google es un desafío especial para Code Search debido a su gran tamaño. En los primeros días, se adoptó un enfoque basado en trigramas. Posteriormente, Russ Cox abrió una fuenteversión simplificada. Actualmente, Code Search indexa alrededor de 1,5 TB de contenido y procesa alrededor de 200 consultas por segundo con una latencia de búsqueda del lado del servidor mediana de menos de 50 ms y una latencia de indexación mediana (tiempo entre la confirmación del código y la visibilidad en el índice) de menos de 10 segundos.

Estimemos aproximadamente los requisitos de recursos para lograr este rendimiento con un grep-Solución basada en fuerza bruta. La biblioteca RE2 que usamos para la coincidencia de expresiones regulares procesa alrededor de 100 MB/seg para datos en RAM. Dada una ventana de tiempo de 50 ms, se necesitarían 300 000 núcleos para procesar los 1,5 TB de datos. Debido a que en la mayoría de los casos, las búsquedas de subcadenas simples son suficientes, se podría reemplazar la coincidencia de expresiones regulares con una búsqueda de subcadenas especial que puede procesar alrededor de 1 GB/seg.¹¹ bajo ciertas condiciones, reduciendo el número de núcleos en 10 veces. Hasta ahora, hemos analizado solo los requisitos de recursos para procesar una sola consulta en 50 ms. Si recibimos 200 solicitudes por segundo, 10 de ellas estarán activas simultáneamente en esa ventana de 50 ms, lo que nos devolverá a 300 000 núcleos solo para la búsqueda de subcadenas.

¹¹ Ver https://blog.scalyr.com/2014/05/searching-20-gbsec-systems-engineering-before-algorithmsyhttp://volnitsky.com/project/str_search.

Si bien esta estimación ignora que la búsqueda puede detenerse una vez que se encuentra una cierta cantidad de resultados o que las restricciones de archivos se pueden evaluar de manera mucho más efectiva que las búsquedas de contenido, no requiere gastos generales de comunicación, clasificación o distribución a decenas de miles de máquinas en cuenta tampoco. Pero muestra bastante bien la escala involucrada y por qué el equipo de búsqueda de códigos de Google invierte continuamente en mejorar la indexación. A lo largo de los años, nuestro índice cambió de la solución original basada en trigramas, a través de una solución basada en matrices de sufijos personalizados, a la solución actual de n-gramas dispersos. Esta última solución es más de 500 veces más eficiente que la solución de fuerza bruta y también es capaz de responder búsquedas de expresiones regulares a una velocidad vertiginosa.

Una de las razones por las que pasamos de una solución basada en matrices de sufijos a una solución de n-gramas basada en tokens fue para aprovechar la indexación principal y la pila de búsqueda de Google. Con una solución basada en matrices de sufijos, la creación y distribución de índices personalizados se convierte en un desafío en sí mismo. Al utilizar tecnología "estándar", nos beneficiamos de todos los avances en la construcción, codificación y publicación de índices inversos realizados por el equipo central de búsqueda. La indexación instantánea es otra característica que existe en las pilas de búsqueda estándar y, en sí misma, es un gran desafío cuando se resuelve a escala.

Confiar en la tecnología estándar es una compensación entre la simplicidad de implementación y el rendimiento. Aunque la implementación de Code Search de Google se basa en índices inversos estándar, la recuperación, la coincidencia y la puntuación reales están altamente personalizadas y optimizadas. De lo contrario, algunas de las funciones de búsqueda de código más avanzadas no serían posibles. Para indexar el historial de revisiones de archivos, se nos ocurrió un esquema de compresión personalizado en el que indexar el historial completo aumentó el consumo de recursos en un factor de solo 2,5.

En los primeros días, Code Search servía todos los datos de la memoria. Con el aumento del tamaño del índice, movimos el [índice invertido](#) para destellar. Aunque el almacenamiento flash es al menos un orden de magnitud más barato que la memoria, su latencia de acceso es al menos dos órdenes de magnitud mayor. Por lo tanto, los índices que funcionan bien en la memoria pueden no ser adecuados cuando se sirven desde flash. Por ejemplo, el índice de trígrama original requiere obtener no solo una gran cantidad de índices inversos de flash, sino también bastante grandes. Con los esquemas de n-gramas, tanto el número de índices inversos como su tamaño pueden reducirse a expensas de un índice más grande.

Para admitir espacios de trabajo locales (que tienen un pequeño delta del repositorio global), tenemos varias máquinas que realizan búsquedas simples de fuerza bruta. Los datos del espacio de trabajo se cargan en la primera solicitud y luego se mantienen sincronizados escuchando los cambios en los archivos. Cuando nos quedamos sin memoria, eliminamos el espacio de trabajo menos reciente de las máquinas. Los documentos sin cambios se buscan con nuestro índice de historial. Por lo tanto, la búsqueda se restringe implícitamente al estado del repositorio con el que se sincroniza el espacio de trabajo.

Clasificación

Para una base de código muy pequeña, la clasificación no brinda muchos beneficios, porque de todos modos no hay muchos resultados. Pero cuanto más grande sea la base de código, más resultados se encontrarán y más importante será la clasificación. En el código base de Google, cualquier subcadena corta aparecerá miles, si no millones, de veces. Sin clasificación, el usuario debe verificar todos esos resultados para encontrar el correcto o debe refinar la consulta.¹² más hasta que el conjunto de resultados se reduce a solo un puñado de archivos. Ambas opciones desperdician el tiempo del desarrollador.

La clasificación generalmente comienza con una función de puntuación, que asigna un conjunto de características de cada archivo ("señales") a algún número: cuanto mayor sea la puntuación, mejor será el resultado. El objetivo de la búsqueda es entonces encontrar la parte superior *norte* de los resultados de la manera más eficiente posible. Típicamente, se distingue entre dos tipos de señales: aquellas que dependen únicamente del documento ("query *independiente*") y las que dependen de la consulta de búsqueda y de su coincidencia con el documento ("query *dependiente*"). La longitud del nombre del archivo o el lenguaje de programación de un archivo serían ejemplos de señales independientes de la consulta, mientras que si una coincidencia es una definición de función o una cadena literal es una señal dependiente de la consulta.

Consultar señales independientes

Algunas de las señales independientes de consultas más importantes son la cantidad de vistas de archivos y la cantidad de referencias a un archivo. Las vistas de archivos son importantes porque indican qué archivos consideran importantes los desarrolladores y, por lo tanto, es más probable que deseen encontrarlos. Por ejemplo, las funciones de utilidad en las bibliotecas base tienen un alto número de visualizaciones. No importa si la biblioteca ya es estable y ya no cambia o si la biblioteca se está desarrollando activamente. El mayor inconveniente de esta señal es el bucle de retroalimentación que crea. Al puntuar más alto los documentos vistos con frecuencia, aumenta la posibilidad de que los desarrolladores los vean y disminuye la posibilidad de que otros documentos lleguen a la cima.*norte*. Este problema se conoce como *explotación versus exploración*, para las que existen varias soluciones (p. ej., experimentos de búsqueda A/B avanzados o conservación de datos de entrenamiento). En la práctica, no parece perjudicial mostrar demasiado los elementos de puntuación más alta: simplemente se ignoran cuando son irrelevantes y se toman si se necesita un ejemplo genérico. Sin embargo, es un problema para los archivos nuevos, que aún no tienen suficiente información para una buena señal.¹³

También usamos el número de referencias a un archivo, que es paralelo al original [algoritmo de rango de página](#), reemplazando los enlaces web como referencias con los diversos tipos de declaraciones de "incluir/importar" presentes en la mayoría de los idiomas. Podemos extender el concepto hasta construir

¹² A diferencia de la búsqueda web, agregar más caracteres a una consulta de búsqueda de código siempre reduce el conjunto de resultados (aparte de unas pocas raras excepciones a través de términos de expresión regular).

¹³ Es probable que esto se pueda corregir de alguna manera usando la actualidad de alguna forma como una señal, tal vez haciendo algo similar a la búsqueda web que se ocupa de nuevas páginas, pero aún no lo hacemos.

dependencias (referencias a nivel de biblioteca/módulo) y hasta funciones y clases. Esta relevancia global a menudo se denomina "prioridad" del documento.

Cuando se utilizan referencias para la clasificación, se deben tener en cuenta dos desafíos. En primer lugar, debe poder extraer información de referencia de forma fiable. En los primeros días, Code Search de Google extraía declaraciones de inclusión/importación con expresiones regulares simples y luego aplicaba heurística para convertirlas en rutas de archivo completas. Con la creciente complejidad de una base de código, tales heurísticas se volvieron propensas a errores y difíciles de mantener. Internamente, reemplazamos esta parte con la información correcta del gráfico de Kythe.

Refactorizaciones a gran escala, como [bibliotecas centrales de código abierto](#), presentan un segundo desafío. Dichos cambios no ocurren atómicamente en una sola actualización de código; más bien, deben implementarse en múltiples etapas. Por lo general, se introducen indirectas, ocultando, por ejemplo, el movimiento de archivos de los usos. Estos tipos de indireccionamientos reducen el rango de página de los archivos movidos y dificultan que los desarrolladores descubran la nueva ubicación. Además, las vistas de archivos generalmente se pierden cuando los archivos se mueven, lo que empeora aún más la situación. Debido a que tales reestructuraciones globales de la base de código son comparativamente raras (la mayoría de las interfaces se mueven rara vez), la solución más simple es aumentar manualmente los archivos durante dichos períodos de transición. (O espere hasta que se complete la migración y los procesos naturales clasifiquen el archivo en su nueva ubicación).

Señales dependientes de la consulta

Las señales independientes de la consulta se pueden calcular fuera de línea, por lo que el costo computacional no es una preocupación importante, aunque puede ser alto. Por ejemplo, para el rango de "página", la señal depende de todo el corpus y requiere un procesamiento por lotes similar a MapReduce para calcular. Consulta *dependientes* las señales, que deben calcularse para cada consulta, deberían ser económicas de calcular. Esto significa que están restringidos a la consulta y la información rápidamente accesible desde el índice.

A diferencia de la búsqueda web, no solo hacemos coincidir los tokens. Sin embargo, si hay coincidencias de token limpias (es decir, el término de búsqueda coincide con el contenido con algún tipo de ruptura, como un espacio en blanco, alrededor), se aplica un impulso adicional y se considera la distinción entre mayúsculas y minúsculas. Esto significa, por ejemplo, que una búsqueda de "Punto" obtendrá una puntuación más alta frente a "Punto * p" que contra "appuntoed al consejo."

Para mayor comodidad, una búsqueda predeterminada coincide con el nombre de archivo y los símbolos calificados¹⁴ además del contenido real del archivo. Un usuario *puede* especificar el tipo particular de coincidencia, pero no es necesario. La puntuación aumenta las coincidencias de símbolos y nombres de archivo por encima de lo normal

¹⁴ En los lenguajes de programación, un símbolo como una función "Alerta" a menudo se define en un ámbito particular, como una clase ("Monitor") o espacio de nombres ("absl"). Entonces, el nombre calificado podría ser absl:Monitor::Alert, y esto se puede encontrar, incluso si no aparece en el texto real.

coincidencias de contenido para reflejar la intención inferida del desarrollador. Al igual que con las búsquedas web, los desarrolladores pueden agregar más términos a la búsqueda para que las consultas sean más específicas. Es muy común que una consulta sea "calificada" con sugerencias sobre el nombre del archivo (por ejemplo, "base" o "miproyecto"). La puntuación aprovecha esto al impulsar los resultados donde gran parte de la consulta ocurre en la ruta completa del resultado potencial, colocando dichos resultados por delante de aquellos que contienen solo las palabras en lugares aleatorios en su contenido.

Recuperación

Antes de que se pueda puntuar un documento, se encuentran los candidatos que probablemente coincidan con la consulta de búsqueda. Esta fase se llama recuperación. Debido a que no es práctico recuperar todos los documentos, pero solo los documentos recuperados pueden puntuarse, la recuperación y la puntuación deben funcionar bien juntas para encontrar los documentos más relevantes. Un ejemplo típico es buscar un nombre de clase. Dependiendo de la popularidad de la clase, puede tener miles de usos, pero potencialmente solo una definición. Si la búsqueda no se restringió explícitamente a las definiciones de clase, la recuperación de un número fijo de resultados podría detenerse antes de que se alcanzara el archivo con la definición única. Obviamente, el problema se vuelve más desafiante a medida que crece la base de código.

El principal desafío para la fase de recuperación es encontrar los pocos archivos altamente relevantes entre la mayor parte de los menos interesantes. Una solución que funciona bastante bien se llama *recuperación suplementaria*. La idea es reescribir la consulta original en otras más especializadas. En nuestro ejemplo, esto significaría que una consulta complementaria restringiría la búsqueda solo a definiciones y nombres de archivos y agregaría los documentos recién recuperados al resultado de la fase de recuperación. En una implementación ingenua de recuperación suplementaria, es necesario puntuar más documentos, pero la información de puntuación parcial adicional obtenida se puede utilizar para evaluar completamente solo los documentos más prometedores de la fase de recuperación.

Diversidad de resultados

Otro aspecto de la búsqueda es la diversidad de resultados, lo que significa tratar de dar los mejores resultados en múltiples categorías. Un ejemplo simple sería proporcionar las coincidencias de Java y Python para un nombre de función simple, en lugar de llenar la primera página de resultados con uno u otro.

Esto es especialmente importante cuando la intención del usuario no está clara. Uno de los desafíos con la diversidad es que hay muchas categorías diferentes, como funciones, clases, nombres de archivos, resultados locales, usos, pruebas, ejemplos, etc., en las que se pueden agrupar los resultados, pero no hay una sola. mucho espacio en la interfaz de usuario para mostrar los resultados de todos ellos o incluso de todas las combinaciones, ni tampoco sería siempre deseable. La búsqueda de código de Google no hace esto tan bien como la búsqueda web, pero la lista desplegable de resultados sugeridos (como las funciones de autocompletar de la búsqueda web) se modifica para proporcionar un conjunto diverso de nombres de archivos principales, definiciones y coincidencias en la búsqueda del usuario. espacio de trabajo actual.

Compensaciones seleccionadas

Implementar Code Search dentro de una base de código del tamaño de Google, y mantenerla receptiva, implicó hacer una variedad de compensaciones. Estos se indican en la siguiente sección.

Integridad: Repositorio en Head

Hemos visto que una base de código más grande tiene consecuencias negativas para la búsqueda; por ejemplo, indexación más lenta y costosa, consultas más lentas y resultados más ruidosos. ¿Se pueden reducir estos costos sacrificando la integridad? en otras palabras, ¿dejando algún contenido fuera del índice? La respuesta es sí, pero con precaución. Los archivos que no son de texto (binarios, imágenes, videos, sonido, etc.) generalmente no están destinados a ser leídos por humanos y se eliminan de su nombre de archivo. Debido a que son enormes, esto ahorra muchos recursos. Un caso más límite involucra archivos JavaScript generados. Debido a la ofuscación y la pérdida de estructura, son prácticamente ilegibles para los humanos, por lo que excluirlos del índice suele ser una buena compensación, ya que reduce los recursos de indexación y el ruido a costa de la integridad. Empíricamente,

Sin embargo, eliminar archivos del índice tiene un gran inconveniente. Para que los desarrolladores confíen en Code Search, deben poder confiar en él. Desafortunadamente, generalmente es imposible dar retroalimentación sobre resultados de búsqueda incompletos para una búsqueda específica si los archivos soltados no fueron indexados en primer lugar. La confusión resultante y la pérdida de productividad para los desarrolladores es un alto precio a pagar por los recursos ahorrados. Incluso si los desarrolladores son plenamente conscientes de las limitaciones, si aún necesitan realizar su búsqueda, lo harán de manera ad hoc y propensa a errores. Dados estos costos raros pero potencialmente altos, optamos por errar en el lado de la indexación demasiado, con límites bastante altos que se eligen principalmente para evitar el abuso y garantizar la estabilidad del sistema en lugar de ahorrar recursos.

En la otra dirección, los archivos generados no están en el código base, pero a menudo serían útiles para indexar. Actualmente no lo son, porque indexarlos requeriría integrar las herramientas y la configuración para crearlos, lo que sería una fuente masiva de complejidad, confusión y latencia.

Integridad: todos los resultados frente a los más relevantes

La búsqueda normal sacrifica la integridad por la velocidad, esencialmente apostando a que la clasificación garantizará que los mejores resultados contengan todos los resultados deseados. Y, de hecho, para Code Search, la búsqueda clasificada es el caso más común en el que el usuario busca algo en particular, como una definición de función, potencialmente entre millones de coincidencias. Sin embargo, a veces los desarrolladores quieren *todas* los resultados; por ejemplo, encontrar todas las ocurrencias de un símbolo particular para refactorizar. Necesar todos los resultados es común para

análisis, herramientas o refactorización, como una búsqueda y reemplazo global. La necesidad de ofrecer todos los resultados es una diferencia fundamental con respecto a la búsqueda web en la que se pueden tomar muchos atajos, como considerar solo los elementos mejor clasificados.

Ser capaz de entregar *todas* los resultados para conjuntos de resultados muy grandes tienen un alto costo, pero sentimos que era necesario para las herramientas y para que los desarrolladores confiaran en los resultados. Sin embargo, debido a que para la mayoría de las consultas solo unos pocos resultados son relevantes (o solo hay unas pocas coincidencias¹⁵), no queríamos sacrificar la velocidad promedio por la integridad potencial.

Para lograr ambos objetivos con una arquitectura, dividimos el código base en fragmentos con archivos ordenados por su prioridad. Luego, generalmente necesitamos considerar solo las coincidencias con los archivos de alta prioridad de cada fragmento. Esto es similar a cómo funciona la búsqueda web. Sin embargo, si se solicita, Code Search puede obtener *todas* los resultados de cada fragmento, para garantizar la búsqueda de todos los resultados.¹⁶ Esto nos permite abordar ambos casos de uso, sin que las búsquedas típicas se vean ralentizadas por la capacidad utilizada con menos frecuencia de devolver conjuntos de resultados grandes y completos. Los resultados también se pueden entregar en orden alfabético, en lugar de clasificarlos, lo cual es útil para algunas herramientas.

Entonces, aquí la compensación fue una implementación más compleja y una API frente a mayores capacidades, en lugar de la latencia más obvia frente a la integridad.

Compleitud: Head Versus Branches Versus All History Versus Workspaces

Relacionado con la dimensión del tamaño del corpus está la cuestión de qué versiones de código se deben indexar: específicamente, si se debe indexar algo más que la instantánea actual del código ("cabeza"). La complejidad del sistema, el consumo de recursos y el costo general aumentan drásticamente si se indexa más de una revisión de archivo. Hasta donde sabemos, ningún IDE indexa nada más que la versión actual del código. Al observar los sistemas de control de versiones distribuidos como Git o Mercurial, gran parte de su eficiencia proviene de la compresión de sus datos históricos. Pero la compactación de estas representaciones se pierde cuando se construyen índices inversos. Otro problema es que es difícil indexar de manera eficiente las estructuras gráficas, que son la base de los sistemas de control de versiones distribuidos.

Aunque es difícil indexar varias versiones de un repositorio, hacerlo permite explorar cómo ha cambiado el código y encontrar el código eliminado. Dentro de Google, Code Search indexa el historial (lineal) de Piper. Esto significa que el código base puede ser

15 Un análisis de las consultas mostró que alrededor de un tercio de las búsquedas de los usuarios tienen menos de 20 resultados.

16 En la práctica, sucede aún más detrás de escena para que las respuestas no se vuelvan dolorosamente enormes y se desarrolle. Los usuarios no derriban todo el sistema al realizar búsquedas que coinciden con casi todo (imáginate buscar la letra "i" o un solo espacio).

buscado en una instantánea arbitraria del código, por código eliminado, o incluso por código autorizado por ciertas personas.

Un gran beneficio es que el código obsoleto ahora puede simplemente eliminarse de la base de código. Antes, el código a menudo se movía a directorios marcados como obsoletos para que aún se pudiera encontrar más tarde. El índice de historial completo también sentó las bases para buscar de manera efectiva en los espacios de trabajo de las personas (cambios no enviados), que se sincronizan con una instantánea específica del código base. Para el futuro, un índice histórico abre la posibilidad de utilizar señales interesantes para clasificar, como la autoría, la actividad del código, etc.

Los espacios de trabajo son muy diferentes del repositorio global:

- Cada desarrollador puede tener sus propios espacios de trabajo.
- Normalmente hay una pequeña cantidad de archivos modificados dentro de un espacio de trabajo.
- Los archivos en los que se está trabajando cambian con frecuencia.
- Un espacio de trabajo existe solo durante un período de tiempo relativamente corto.

Para proporcionar valor, un índice de espacio de trabajo debe reflejar exactamente el estado actual del espacio de trabajo.

Expresividad: Token Versus Substring Versus Regex

El efecto de la escala está muy influenciado por el conjunto de funciones de búsqueda admitidas. Code Search admite la búsqueda de expresiones regulares (regex), lo que agrega potencia al lenguaje de consulta, lo que permite especificar o excluir grupos completos de términos, y se pueden usar en cualquier texto, lo que es especialmente útil para documentos e idiomas para los que se requiere una semántica más profunda. las herramientas no existen.

Los desarrolladores también están acostumbrados a usar expresiones regulares en otras herramientas (p. ej., grep) contextos, por lo que brindan una búsqueda poderosa sin aumentar la carga cognitiva del desarrollador. Este poder tiene un costo dado que crear un índice para consultarlos de manera eficiente es un desafío. ¿Qué opciones más sencillas existen?

Un índice basado en tokens (es decir, palabras) escala bien porque almacena solo una fracción del código fuente real y es compatible con los motores de búsqueda estándar. La desventaja es que muchos casos de uso son complicados o incluso imposibles de realizar de manera eficiente con un índice basado en tokens cuando se trata de código fuente, que otorga significado a muchos caracteres que normalmente se ignoran cuando se tokenizan. Por ejemplo, buscar "función()" frente a "función(x)", "(x ^ y)" o "== myClass" es difícil o imposible en la mayoría de las búsquedas basadas en tokens.

Otro problema de la tokenización es que la tokenización de los identificadores de código está mal definida. Los identificadores se pueden escribir de muchas maneras, como CamelCase, snake_case, o incluso simplemente combinarlos sin ningún separador de palabras. Encontrar un identificador cuando se recuerdan solo algunas de las palabras es un desafío para un índice basado en tokens.

La tokenización tampoco suele preocuparse por el caso de las letras ("r" frente a "R") y, a menudo, desdibujará las palabras; por ejemplo, reduciendo "buscando" y "buscado" a la misma búsqueda de token de raíz. Esta falta de precisión es un problema importante al buscar código. Finalmente, la tokenización hace que sea imposible buscar en espacios en blanco u otros delimitadores de palabras (comas, paréntesis), que pueden ser muy importantes en el código.

Un siguiente paso¹⁷en el poder de búsqueda está la búsqueda completa de subcademas en la que se puede buscar cualquier secuencia de caracteres. Una forma bastante eficiente de proporcionar esto es a través de un índice basado en trigramas.¹⁸En su forma más simple, el tamaño del índice resultante sigue siendo mucho más pequeño que el tamaño del código fuente original. Sin embargo, el tamaño pequeño tiene el costo de una precisión de recuperación relativamente baja en comparación con otros índices de subcadena. Esto significa consultas más lentas porque las no coincidencias deben filtrarse del conjunto de resultados. Aquí es donde se debe encontrar un buen compromiso entre el tamaño del índice, la latencia de búsqueda y el consumo de recursos que depende en gran medida del tamaño de la base de código, la disponibilidad de recursos y las búsquedas por segundo.

Si hay un índice de subcadena disponible, es fácil ampliarlo para permitir búsquedas de expresiones regulares. La idea básica es convertir el autómata de expresión regular en un conjunto de búsquedas de subcademas. Esta conversión es sencilla para un índice de trigramas y se puede generalizar a otros índices de subcademas. Debido a que no existe un índice de expresión regular perfecto, siempre será posible construir consultas que den como resultado una búsqueda de fuerza bruta. Sin embargo, dado que solo una pequeña fracción de las consultas de los usuarios son expresiones regulares complejas, en la práctica, la aproximación a través de índices de subcademas funciona muy bien.

Conclusión

Code Search creció a partir de un reemplazo orgánico paragrep en una herramienta central que impulsa la productividad de los desarrolladores, aprovechando la tecnología de búsqueda web de Google en el camino. Sin embargo, ¿qué significa esto para ti? Si está en un proyecto pequeño que cabe fácilmente en su IDE, probablemente no mucho. Si usted es responsable de la productividad de los ingenieros en una base de código más grande, es probable que obtenga algunas ideas.

El más importante es quizás obvio: comprender el código es clave para desarrollarlo y mantenerlo, y esto significa que invertir en comprender el código generará dividendos que pueden ser difíciles de medir, pero que son reales. Cada función que agregamos a Code Search fue y es utilizada por los desarrolladores para ayudarlos en su trabajo diario (ciertamente, algunas más que otras). Dos de las características más importantes, la integración de Kythe (es decir, agregar comprensión de código semántico) y encontrar ejemplos de trabajo, también son las más claramente relacionadas con la comprensión del código (frente, por ejemplo, a encontrarlo o ver

17 Existen otras variedades intermedias, como la construcción de un índice de prefijo/sufijo, pero generalmente brindan menos expresividad en las consultas de búsqueda sin dejar de tener una alta complejidad y costos de indexación.

18 Russ Cox, "Coincidencia de expresiones regulares con un índice de trigramas o cómo funcionó la búsqueda de código de Google."

cómo ha cambiado). En términos del impacto de la herramienta, nadie usa una herramienta que no sabe que existe, por lo que también es importante informar a los desarrolladores sobre las herramientas disponibles: en Google, es parte de la capacitación "Noogler", la capacitación de incorporación para los nuevos ingenieros de software contratados.

Para usted, esto podría significar configurar un perfil de indexación estándar para IDE, compartir conocimientos sobre egrep, ejecutar ctags o configurar algunas herramientas de indexación personalizadas, como Code Search. Hagas lo que hagas, es casi seguro que se usará, y se usará más, y de maneras diferentes a las que esperabas, y tus desarrolladores se beneficiarán.

TL; DR

- Ayudar a sus desarrolladores a comprender el código puede ser un gran impulso para la productividad de ingeniería. En Google, la herramienta clave para esto es Code Search.
- Code Search tiene un valor adicional como base para otras herramientas y como un lugar central y estándar al que se vincula toda la documentación y las herramientas de desarrollo.
- El enorme tamaño del código base de Google creó una herramienta personalizada, a diferencia de, por ejemplo, grepo la indexación de un IDE—necesario.
- Como herramienta interactiva, Code Search debe ser rápida y permitir un flujo de trabajo de "preguntas y respuestas". Se espera que tenga baja latencia en todos los aspectos: búsqueda, navegación e indexación.
- Se usará ampliamente solo si es de confianza, y solo se confiará si indexa todo el código, da todos los resultados y da primero los resultados deseados. Sin embargo, las versiones anteriores, menos poderosas, fueron útiles y utilizadas, siempre que se entendieran sus límites.

Sistemas de construcción y filosofía de construcción

Escrito por Erik Kuefler
Editado por Lisa Carey

Si le pregunta a los ingenieros de Google qué es lo que más les gusta de trabajar en Google (además de la comida gratis y los productos geniales), es posible que escuche algo sorprendente: a los ingenieros les encanta el sistema de compilación. Google ha dedicado una enorme cantidad de esfuerzo de ingeniería a lo largo de su vida para crear su propio sistema de compilación desde cero, con el objetivo de garantizar que nuestros ingenieros puedan compilar código de manera rápida y confiable. El esfuerzo ha tenido tanto éxito que Blaze, el componente principal del sistema de compilación, ha sido reimplementado varias veces por ex empleados de Google que han dejado la empresa.² En 2015, Google finalmente abrió una implementación de Blaze llamada **bazel**.

Propósito de un sistema de compilación

Fundamentalmente, todos los sistemas de compilación tienen un propósito sencillo: transforman el código fuente escrito por los ingenieros en archivos binarios ejecutables que las máquinas pueden leer. Un buen sistema de compilación generalmente intentará optimizar dos propiedades importantes:

Rápido

Un desarrollador debería poder escribir un solo comando para ejecutar la compilación y recuperar el binario resultante, a menudo en tan solo unos segundos.

1 En una encuesta interna, el 83% de los Googlers informaron estar satisfechos con el sistema de compilación, lo que lo convierte en el cuarto herramienta más satisfactoria de las 19 encuestadas. La herramienta promedio tuvo una calificación de satisfacción del 69%.

2 Ver <https://buck.build/> y <https://www.pantsbuild.org/index.html>.

Correcto

Cada vez que un desarrollador ejecuta una compilación en cualquier máquina, debe obtener el mismo resultado (suponiendo que los archivos fuente y otras entradas sean las mismas).

Muchos sistemas de compilación más antiguos intentan hacer concesiones entre la velocidad y la corrección tomando atajos que pueden conducir a compilaciones inconsistentes. El objetivo principal de Bazel es evitar tener que elegir entre velocidad y corrección, proporcionando un sistema de compilación estructurado para garantizar que siempre sea posible compilar código de manera eficiente y consistente.

Los sistemas de construcción no son solo para humanos; también permiten que las máquinas creen compilaciones automáticamente, ya sea para pruebas o para lanzamientos a producción. De hecho, la gran mayoría de las compilaciones en Google se activan automáticamente en lugar de que los ingenieros las activen directamente. Casi todas nuestras herramientas de desarrollo se vinculan con el sistema de compilación de alguna manera, lo que brinda una gran cantidad de valor a todos los que trabajan en nuestra base de código. Aquí hay una pequeña muestra de flujos de trabajo que aprovechan nuestro sistema de compilación automatizado:

- El código se crea, prueba y envía automáticamente a producción sin intervención humana. Diferentes equipos hacen esto a diferentes ritmos: algunos equipos presionan semanalmente, otros diariamente y otros tan rápido como el sistema puede crear y validar nuevas compilaciones. (ver[capítulo 24](#)).
- Los cambios del desarrollador se prueban automáticamente cuando se envían para revisión de código (ver[capítulo 19](#)) para que tanto el autor como el revisor puedan ver de inmediato cualquier problema de compilación o prueba causado por el cambio.
- Los cambios se vuelven a probar inmediatamente antes de fusionarlos en el tronco, lo que hace que sea mucho más difícil enviar cambios importantes.
- Los autores de bibliotecas de bajo nivel pueden probar sus cambios en todo el código base, lo que garantiza que sus cambios estén seguros en millones de pruebas y archivos binarios.
- Los ingenieros pueden crear cambios a gran escala (LSC) que afectan a decenas de miles de archivos de origen a la vez (p. ej., cambiar el nombre de un símbolo común) y al mismo tiempo pueden enviar y probar esos cambios de forma segura. Discutimos los LSC con mayor detalle en [capítulo 22](#).

Todo esto es posible gracias a la inversión de Google en su sistema de compilación. Aunque Google puede ser único en su escala, cualquier organización de cualquier tamaño puede obtener beneficios similares al hacer un uso adecuado de un sistema de construcción moderno. Este capítulo describe lo que Google considera un "sistema de compilación moderno" y cómo utilizar dichos sistemas.

¿Qué sucede sin un sistema de compilación?

Los sistemas de compilación permiten escalar su desarrollo. Como ilustraremos en la siguiente sección, nos encontramos con problemas de escalado sin un entorno de compilación adecuado.

¡Pero todo lo que necesito es un compilador!

La necesidad de un sistema de compilación puede no ser obvia de inmediato. Después de todo, la mayoría de nosotros probablemente no usamos un sistema de compilación cuando aprendimos a programar por primera vez; probablemente comenzamos invocando herramientas como CCGoJava directamente desde la línea de comandos, o el equivalente en un entorno de desarrollo integrado (IDE). Mientras todo nuestro código fuente esté en el mismo directorio, un comando como este funciona bien:

```
javac *.java
```

Esto le indica al compilador de Java que tome cada archivo fuente de Java en el directorio actual y lo convierta en un archivo de clase binario. En el caso más simple, esto es todo lo que necesitamos.

Sin embargo, las cosas se vuelven más complicadas tan pronto como nuestro código se expande. javac es lo suficientemente inteligente como para buscar en los subdirectorios de nuestro directorio actual para encontrar el código que importamos. Pero no tiene forma de encontrar el código almacenado en otras partes del sistema de archivos (quizás una biblioteca compartida por varios de nuestros proyectos). Obviamente, también solo sabe cómo construir código Java. Los sistemas grandes a menudo involucran diferentes piezas escritas en una variedad de lenguajes de programación con redes de dependencias entre esas piezas, lo que significa que ningún compilador para un solo lenguaje puede construir el sistema completo.

Tan pronto como tengamos que lidiar con código de varios idiomas o varias unidades de compilación, la creación de código ya no es un proceso de un solo paso. Ahora debemos pensar de qué depende nuestro código y construir esas piezas en el orden correcto, posiblemente usando un conjunto diferente de herramientas para cada pieza. Si cambiamos alguna de las dependencias, debemos repetir este proceso para evitar depender de binarios obsoletos. Para una base de código incluso de tamaño moderado, este proceso rápidamente se vuelve tedioso y propenso a errores.

El compilador tampoco sabe nada sobre cómo manejar dependencias externas, como archivos JAR de terceros en Java. A menudo, lo mejor que podemos hacer sin un sistema de compilación es descargar la dependencia de Internet, pegarla en un liberación carpeta en el disco duro y configure el compilador para leer bibliotecas desde ese directorio. Con el tiempo, es fácil olvidar qué bibliotecas pusimos allí, de dónde provienen y si todavía están en uso. Y buena suerte manteniéndolos actualizados a medida que los mantenedores de la biblioteca lanzan nuevas versiones.

Shell Scripts al rescate?

Suponga que su proyecto de pasatiempo comienza de manera tan simple que puede construirlo usando solo un compilador, pero comienza a encontrarse con algunos de los problemas descritos anteriormente. Tal vez todavía no crea que necesita un sistema de construcción real y puede automatizar las partes tediosas usando algunos scripts de shell simples que se encargan de construir las cosas en el orden correcto. Esto ayuda por un tiempo, pero muy pronto comienzas a encontrarte con más problemas:

- Se vuelve tedioso. A medida que su sistema se vuelve más complejo, comienza a pasar casi tanto tiempo trabajando en sus scripts de compilación como en el código real. La depuración de scripts de shell es dolorosa, con más y más hacks superpuestos.
- Es lento. Para asegurarse de que no estaba confiando accidentalmente en bibliotecas obsoletas, tiene su secuencia de comandos de compilación compilando cada dependencia en orden cada vez que la ejecuta. Piensa en agregar algo de lógica para detectar qué partes deben reconstruirse, pero eso suena terriblemente complejo y propenso a errores para una secuencia de comandos. O piensa en especificar qué partes deben reconstruirse cada vez, pero luego vuelve al punto de partida.
- Buenas noticias: ¡es hora de un lanzamiento! Mejor ve a averiguar todos los argumentos que necesitas para pasar alfrascocomando a **hacer su construcción final**. Y recuerde cómo cargarlo y enviarlo al repositorio central. Y cree e impulse las actualizaciones de la documentación y envíe una notificación a los usuarios. Hmm, tal vez esto requiera otro guión...
- ¡Desastre! Su disco duro falla y ahora necesita volver a crear todo el sistema. Fuiste lo suficientemente inteligente como para mantener todos tus archivos fuente bajo control de versiones, pero ¿qué pasa con las bibliotecas que descargaste? ¿Puedes encontrarlos todos de nuevo y asegurarte de que tenían la misma versión que cuando los descargaste por primera vez? Sus scripts probablemente dependían de la instalación de herramientas particulares en lugares particulares
 - ¿Puede restaurar ese mismo entorno para que los scripts vuelvan a funcionar? ¿Qué pasa con todas esas variables de entorno que configuró hace mucho tiempo para que el compilador funcionara correctamente y luego las olvidó?
- A pesar de los problemas, su proyecto es lo suficientemente exitoso como para que pueda comenzar a contratar más ingenieros. Ahora se da cuenta de que no hace falta un desastre para que surjan los problemas anteriores: debe pasar por el mismo doloroso proceso de arranque cada vez que un nuevo desarrollador se une a su equipo. Y a pesar de sus mejores esfuerzos, todavía hay pequeñas diferencias en el sistema de cada persona. Con frecuencia, lo que funciona en la máquina de una persona no funciona en la de otra, y cada vez se necesitan algunas horas de depuración de rutas de herramientas o versiones de biblioteca para descubrir dónde está la diferencia.
- Decide que necesita automatizar su sistema de compilación. En teoría, esto es tan simple como obtener una computadora nueva y configurarla para ejecutar su script de compilación todas las noches usando cron. Todavía necesita pasar por el doloroso proceso de configuración, pero ahora no tiene el beneficio de que un cerebro humano pueda detectar y resolver problemas menores. Ahora, cada mañana, cuando ingresa, ve que la compilación de anoche falló porque ayer un desarrollador realizó un cambio que funcionó en su sistema pero no funcionó en el sistema de compilación automatizado. Cada vez es una solución simple, pero sucede con tanta frecuencia que terminas dedicando mucho tiempo cada día a descubrir y aplicar estas soluciones simples.

- Las compilaciones se vuelven cada vez más lentas a medida que crece el proyecto. Un día, mientras espera que se complete una compilación, observa con tristeza el escritorio inactivo de su compañero de trabajo, que está de vacaciones, y desea que haya una manera de aprovechar toda esa potencia informática desperdienciada.

Te has encontrado con un problema clásico de escala. Para un solo desarrollador que trabaje en un par de cientos de líneas de código como máximo durante una semana o dos (que podría haber sido toda la experiencia hasta ahora de un desarrollador junior que acaba de graduarse de la universidad), todo lo que necesita es un compilador. Los guiones pueden llevarte un poco más lejos. Pero tan pronto como necesite coordinarse entre múltiples desarrolladores y sus máquinas, incluso un script de compilación perfecto no es suficiente porque se vuelve muy difícil tener en cuenta las diferencias menores en esas máquinas. En este punto, este enfoque simple falla y es hora de invertir en un sistema de compilación real.

Sistemas de construcción modernos

Afortunadamente, todos los problemas con los que comenzamos a encontrarnos ya han sido resueltos muchas veces por los sistemas de compilación de propósito general existentes. Fundamentalmente, no son tan diferentes del enfoque de bricolaje basado en secuencias de comandos mencionado anteriormente en el que estábamos trabajando: ejecutan los mismos compiladores debajo del capó, y es necesario comprender esas herramientas subyacentes para poder saber cuál es el sistema de compilación. realmente haciendo Pero estos sistemas existentes han pasado por muchos años de desarrollo, haciéndolos mucho más robustos y flexibles que los scripts que usted mismo podría intentar hackear juntos.

Se trata de dependencias

Al revisar los problemas descritos anteriormente, un tema se repite una y otra vez: administrar su propio código es bastante sencillo, pero administrar sus dependencias es mucho más difícil ([y capítulo 21](#) se dedica a cubrir este problema en detalle). Hay todo tipo de dependencias: a veces hay una dependencia de una tarea (p. ej., "envíe la documentación antes de marcar una versión como completa") y, a veces, hay una dependencia de un artefacto (p. ej., "Necesito tener la última versión de la biblioteca de visión artificial para construir mi código"). A veces, tiene dependencias internas en otra parte de su base de código y, a veces, tiene dependencias externas en el código o los datos que pertenecen a otro equipo (ya sea en su organización o en un tercero). Pero en cualquier caso, la idea de "Necesito eso antes de poder tener esto" es algo que se repite repetidamente en el diseño de sistemas de compilación, y administrar las dependencias es quizás el trabajo más fundamental de un sistema de compilación.

Sistemas de compilación basados en tareas

Los scripts de shell que comenzamos a desarrollar en la sección anterior eran un ejemplo de un primitivo *sistema de construcción basado en tareas*. En un sistema de compilación basado en tareas, la unidad de trabajo fundamental es la tarea. Cada tarea es un script de algún tipo que puede ejecutar cualquier tipo de lógica, y las tareas especifican otras tareas como dependencias que deben ejecutarse antes que ellas. La mayoría de los principales sistemas de compilación que se utilizan hoy en día, como Ant, Maven, Gradle, Grunt y Rake, se basan en tareas.

En lugar de scripts de shell, la mayoría de los sistemas de compilación modernos requieren que los ingenieros creen *archivos de compilación* que describen cómo realizar la compilación. Tome este ejemplo de la [Manual de hormigas](#):

```
<proyecto nombre="Mi proyecto" predeterminado="dist" basedir="."> >
  <descripción>
    archivo de compilación de ejemplo
    simple </descripción>
    <!-- establecer propiedades globales para esta compilación -->
    <propiedad nombre="origen" ubicación="origen"/>
    <propiedad nombre="construir" ubicación="construir"/>
    <propiedad nombre="dist" ubicación="dist"/>

    <objetivo nombre="en eso">
      <!-- Crear la marca de tiempo -->
      <sello de t/>
      <!-- Crea la estructura de directorios de compilación utilizada por compile -->
      <mkdirdir="${{construir}}"/> </objetivo>

    <objetivo nombre="compilar" depende ="en eso"
              descripción="compilar la fuente">
      <!-- Compila el código Java de ${src} en ${build} --> <javac
      srcdir="${{origen}}" dirdestino="${{construir}}"/> </objetivo>

    <objetivo nombre="dist" depende ="compilar"
              descripción="generar la distribución"> <!--
              Crear el directorio de distribución --> <mkdirdir=
              "${{dist}}/lib"/>
      <!-- Coloque todo en ${build} en el archivo MyProject-${{DSTAMP}}.jar --> <frasco
      archivojar="${{dist}}/lib/MiProyecto-${{DESTAMP}}.jar" basedir="${{construir}}"/> </
      objetivo>

    <objetivo nombre="limpio"
              descripción="limpiar">
      <!-- Eliminar los árboles de directorios ${build} y ${dist} --> <borrar
      dir="${{construir}}"/> <eliminardir="${{distancia}}"/> </objetivo>

  </proyecto>
```

El archivo de compilación está escrito en XML y define algunos metadatos simples sobre la compilación junto con una lista de tareas (el <objetivo>etiquetas en el XML³). Cada tarea ejecuta una lista de posibles comandos definidos por Ant, que aquí incluyen crear y eliminar directorios, ejecutar javac,y creando un archivo JAR. Este conjunto de comandos se puede ampliar mediante complementos proporcionados por el usuario para cubrir cualquier tipo de lógica. Cada tarea también puede definir las tareas de las que depende mediante el dependenciaatributo. Estas dependencias forman un gráfico acíclico (verFigura 18-1).

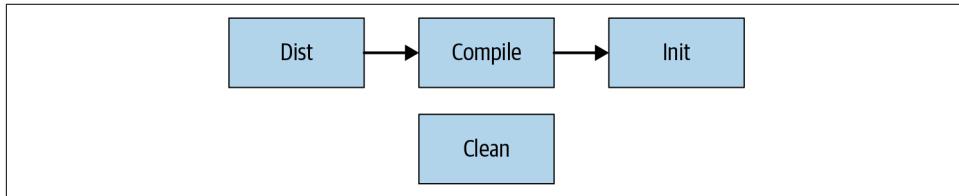


Figura 18-1. Un gráfico acíclico que muestra dependencias

Los usuarios realizan compilaciones proporcionando tareas a la herramienta de línea de comandos de Ant. Por ejemplo, cuando un usuario escribe `dist` en la línea de comando, Ant sigue los siguientes pasos:

1. Carga un archivo llamado `construir.xml` en el directorio actual y lo analiza para crear la estructura gráfica que se muestra en Figura 18-1.
2. Busca la tarea nombrada `dist` que se proporcionó en la línea de comando y descubre que tiene una dependencia en la tarea denominada `compile`.
3. Busca la tarea nombrada `compile` y descubre que tiene una dependencia en la tarea llamada `en`.
4. Busca la tarea nombrada `en` y descubre que no tiene dependencias.
5. Ejecuta los comandos definidos en la tarea `en`.
6. Ejecuta los comandos definidos en la tarea `compile` dado que todas las dependencias de esa tarea se han ejecutado.
7. Ejecuta los comandos definidos en la tarea `dist` dado que todas las dependencias de esa tarea se han ejecutado.

Al final, el código ejecutado por Ant al ejecutar la tarea `dist` es equivalente al siguiente script de shell:

```
./createTimestamp.sh
construir mkdir/
javac src/* -d construir/
```

³ Ant usa la palabra "objetivo" para representar lo que llamamos una "tarea" en este capítulo, y usa la palabra "tarea" para referirse a lo que llamamos "comandos".

```
mkdir -p distribución/lib/  
jar cf dist/lib/MiProyecto-psfecha --iso-8601).jar construir/*
```

Cuando se elimina la sintaxis, el archivo de compilación y el script de compilación en realidad no son muy diferentes. Pero ya hemos ganado mucho haciendo esto. Podemos crear nuevos archivos de compilación en otros directorios y vincularlos entre sí. Podemos agregar fácilmente nuevas tareas que dependen de las tareas existentes de manera arbitraria y compleja. Solo necesitamos pasar el nombre de una sola tarea al hormigaherramienta de línea de comandos, y se encargará de determinar todo lo que debe ejecutarse.

Ant es una pieza de software muy antigua, lanzada originalmente en 2000, ¡no es lo que muchas personas considerarían un sistema de compilación "moderno" en la actualidad! Otras herramientas como Maven y Gradle han mejorado Ant en los años intermedios y esencialmente lo reemplazaron al agregar funciones como la administración automática de dependencias externas y una sintaxis más limpia sin ningún XML. Pero la naturaleza de estos sistemas más nuevos sigue siendo la misma: permiten a los ingenieros escribir scripts de compilación de manera modular y basada en principios como tareas y proporcionan herramientas para ejecutar esas tareas y administrar las dependencias entre ellas.

El lado oscuro de los sistemas de compilación basados en tareas

Debido a que estas herramientas esencialmente permiten a los ingenieros definir cualquier secuencia de comandos como una tarea, son extremadamente poderosas y le permiten hacer prácticamente cualquier cosa que pueda imaginar con ellas. Pero ese poder viene con inconvenientes, y los sistemas de compilación basados en tareas pueden volverse difíciles de trabajar a medida que sus scripts de compilación se vuelven más complejos. El problema con tales sistemas es que en realidad terminan dando *demasiado poder para los ingenieros y no suficiente poder para el sistema*. Debido a que el sistema no tiene idea de lo que están haciendo los scripts, el rendimiento se ve afectado, ya que debe ser muy conservador en la forma en que programa y ejecuta los pasos de compilación. Y no hay forma de que el sistema confirme que cada secuencia de comandos está haciendo lo que debe, por lo que las secuencias de comandos tienden a crecer en complejidad y terminan siendo otra cosa que necesita depuración.

Dificultad de paralelizar los pasos de compilación. Las estaciones de trabajo de desarrollo modernas suelen ser bastante potentes, con varios núcleos que, en teoría, deberían ser capaces de ejecutar varios pasos de compilación en paralelo. Pero los sistemas basados en tareas a menudo no pueden parallelizar la ejecución de tareas incluso cuando parece que deberían poder hacerlo. Suponga que la tarea A depende de las tareas B y C. Debido a que las tareas B y C no dependen entre sí, ¿es seguro ejecutarlas al mismo tiempo para que el sistema pueda llegar más rápidamente a la tarea A? Tal vez, si no tocan ninguno de los mismos recursos. Pero tal vez no, tal vez ambos usen el mismo archivo para rastrear sus estados y ejecutarlos al mismo tiempo provoque un conflicto. En general, no hay forma de que el sistema lo sepa, por lo que tiene que arriesgarse a estos conflictos (lo que lleva a problemas de compilación raros pero muy difíciles de depurar), o tiene que restringir la compilación completa para que se ejecute en un solo subproceso en un solo proceso. Esto puede ser un gran desperdicio de una poderosa máquina de desarrollo y descarta por completo la posibilidad de distribuir la compilación en varias máquinas.

Dificultad para realizar compilaciones incrementales. Un buen sistema de construcción permitirá a los ingenieros realice compilaciones incrementales confiables de modo que un pequeño cambio no requiera que todo el código base se reconstruya desde cero. Esto es especialmente importante si el sistema de compilación es lento y no puede paralelizar los pasos de compilación por las razones mencionadas anteriormente. Pero desafortunadamente, los sistemas de compilación basados en tareas también tienen problemas aquí. Debido a que las tareas pueden hacer cualquier cosa, en general no hay forma de comprobar si ya se han realizado. Muchas tareas simplemente toman un conjunto de archivos fuente y ejecutan un compilador para crear un conjunto de archivos binarios; por lo tanto, no es necesario volver a ejecutarlos si los archivos de origen subyacentes no han cambiado. Pero sin información adicional, el sistema no puede decir esto con certeza: tal vez la tarea descargue un archivo que podría haber cambiado, o tal vez escriba una marca de tiempo que podría ser diferente en cada ejecución. Para garantizar la corrección,

Algunos sistemas de compilación intentan habilitar compilaciones incrementales al permitir que los ingenieros especifiquen las condiciones bajo las cuales se debe volver a ejecutar una tarea. A veces esto es factible, pero a menudo es un problema mucho más complicado de lo que parece. Por ejemplo, en lenguajes como C++ que permiten que otros archivos incluyan archivos directamente, es imposible determinar el conjunto completo de archivos que deben observarse en busca de cambios sin analizar las fuentes de entrada. Los ingenieros a menudo terminarán tomando atajos, y estos atajos pueden conducir a problemas raros y frustrantes donde el resultado de una tarea se reutiliza incluso cuando no debería ser así. Cuando esto sucede con frecuencia, los ingenieros adquieren el hábito de ejecutar limpioantes de cada compilación para obtener un estado nuevo, anulando por completo el propósito de tener una compilación incremental en primer lugar. Averiguar cuándo se debe volver a ejecutar una tarea es sorprendentemente sutil, y es un trabajo mejor manejado por máquinas que por humanos.

Dificultad para mantener y depurar scripts. Finalmente, los scripts de compilación impuestos por task. Los sistemas de compilación basados en tareas son difíciles de trabajar. Aunque a menudo reciben menos escrutinio, los scripts de compilación son código como el sistema que se está construyendo, y son lugares fáciles para que los errores se escondan. Estos son algunos ejemplos de errores que son muy comunes cuando se trabaja con un sistema de compilación basado en tareas:

- La tarea A depende de la tarea B para producir un archivo particular como salida. El propietario de la tarea B no se da cuenta de que otras tareas dependen de ella, por lo que la cambia para producir resultados en una ubicación diferente. Esto no se puede detectar hasta que alguien intente ejecutar la tarea A y descubra que falla.
- La tarea A depende de la tarea B, que depende de la tarea C, que produce un archivo en particular como resultado que necesita la tarea A. El propietario de la tarea B decide que ya no necesita depender de la tarea C, ¡lo que hace que la tarea A falle a pesar de que la tarea B no se preocupa en absoluto por la tarea C!
- El desarrollador de una nueva tarea accidentalmente hace una suposición acerca de la máquina que ejecuta la tarea, como la ubicación de una herramienta o el valor de una determinada

Variables de entorno. La tarea funciona en su máquina, pero falla cada vez que otro desarrollador la intenta.

- Una tarea contiene un componente no determinista, como descargar un archivo de Internet o agregar una marca de tiempo a una compilación. Ahora, las personas obtendrán resultados potencialmente diferentes cada vez que ejecuten la compilación, lo que significa que los ingenieros no siempre podrán reproducir y corregir las fallas de los demás o las fallas que ocurren en un sistema de compilación automatizado.
- Las tareas con múltiples dependencias pueden crear condiciones de carrera. Si la tarea A depende tanto de la tarea B como de la tarea C, y la tarea B y C modifican el mismo archivo, la tarea A obtendrá un resultado diferente según cuál de las tareas B y C termine primero.

No existe una forma de propósito general para resolver estos problemas de rendimiento, corrección o mantenibilidad dentro del marco basado en tareas que se presenta aquí. Mientras los ingenieros puedan escribir código arbitrario que se ejecute durante la compilación, el sistema no puede tener suficiente información para poder ejecutar compilaciones siempre de manera rápida y correcta. Para resolver el problema, necesitamos tomar algo de poder de las manos de los ingenieros y devolverlo a las manos del sistema y reconceptualizar el papel del sistema no como ejecutar tareas, sino como producir artefactos. Este es el enfoque que adopta Google con Blaze y Bazel, y se describirá en la siguiente sección.

Sistemas de construcción basados en artefactos

Para diseñar un mejor sistema de compilación, debemos dar un paso atrás. El problema con los sistemas anteriores es que daban demasiado poder a los ingenieros individuales al permitirles definir sus propias tareas. Tal vez en lugar de dejar que los ingenieros definan las tareas, podemos tener una pequeña cantidad de tareas definidas por el sistema que los ingenieros pueden configurar de forma limitada. Probablemente podríamos deducir el nombre de la tarea más importante del nombre de este capítulo: la tarea principal de un sistema de compilación debería ser *construir/código*. Los ingenieros todavía tendrían que decirle al sistema qué para construir, pero el cómo de hacer la compilación se dejaría al sistema.

Este es exactamente el enfoque adoptado por Blaze y los otros *sistemas de compilación basados en artefactos* (que incluyen Bazel, Pants y Buck). Al igual que con los sistemas de compilación basados en tareas, todavía tenemos archivos de compilación, pero el contenido de esos archivos de compilación es muy diferente. En lugar de ser un conjunto imperativo de comandos en un lenguaje de secuencias de comandos completo de Turing que describe cómo producir una salida, los archivos de compilación en Blaze son un *manifiesto declarativo* que describe un conjunto de artefactos para compilar, sus dependencias y un conjunto limitado de opciones que afectan la forma en que se compilan. Cuando los ingenieros corren `resplandor` en la línea de comandos, especifican un conjunto de objetivos para construir (el "qué"), y Blaze es responsable de configurar, ejecutar y programar los pasos de compilación (el "cómo"). Debido a que el sistema de compilación ahora tiene control total sobre qué herramientas se ejecutan y cuándo, puede hacer

garantías mucho más sólidas que le permiten ser mucho más eficiente sin dejar de garantizar la corrección.

Una perspectiva funcional

Es fácil hacer una analogía entre los sistemas de construcción basados en artefactos y la programación funcional. Los lenguajes de programación imperativos tradicionales (p. ej., Java, C y Python) especifican listas de declaraciones que se ejecutarán una tras otra, de la misma manera que los sistemas de compilación basados en tareas permiten a los programadores definir una serie de pasos para ejecutar. Los lenguajes de programación funcional (por ejemplo, Haskell y ML), por el contrario, están estructurados más como una serie de ecuaciones matemáticas. En los lenguajes funcionales, el programador describe un cálculo a realizar, pero deja los detalles de cuándo exactamente cómo se ejecuta ese cálculo al compilador. Esto se corresponde con la idea de declarar un manifiesto en un sistema de compilación basado en artefactos y dejar que el sistema descubra cómo ejecutar la compilación.

Muchos problemas no se pueden expresar fácilmente usando programación funcional, pero los que sí se benefician enormemente de ella: el lenguaje a menudo es capaz de paralelizar trivialmente tales programas y hacer fuertes garantías sobre su corrección que serían imposibles en un lenguaje imperativo. Los problemas más fáciles de expresar usando programación funcional son aquellos que simplemente involucran transformar un dato en otro usando una serie de reglas o funciones. Y eso es exactamente lo que es un sistema de compilación: todo el sistema es efectivamente una función matemática que toma archivos fuente (y herramientas como el compilador) como entradas y produce archivos binarios como salidas. Por lo tanto, no sorprende que funcione bien basar un sistema de compilación en los principios de la programación funcional.

Concretando con Bazel. Bazel es la versión de código abierto de la herramienta de compilación interna de Google, Blaze, y es un buen ejemplo de un sistema de compilación basado en artefactos. Así es como se ve un archivo de compilación (normalmente llamado BUILD) en Bazel:

```
java_binario(  
    nombre = "MiBinario",  
    srcs = ["MiBinario.java"],  
    deps = [  
        ":milib",  
    ],  
)  
  
biblioteca_java(  
    nombre = "milib",  
    srcs = ["MiBiblioteca.java", "MiAyudante.java"],  
    visibilidad = ["//java/com/ejemplo/miproducto:_subpaquetes_"],  
    deps = [  
        "//java/com/ejemplo/común", "//java/com/  
        ejemplo/miproducto/otralib",  
        "@com_google_common_guava_guava//jar",
```

```
],  
)
```

en `bazel`, *CONSTRUIR* los archivos definen *objetivos*—los dos tipos de objetivos aquí son `java_binary` y `biblioteca.java`. Cada objetivo corresponde a un artefacto que puede ser creado por el sistema: binarios los objetivos producen binarios que se pueden ejecutar directamente, y biblioteca los destinos producen bibliotecas que pueden ser utilizadas por binarios u otras bibliotecas. Todo objetivo tiene *un nombre* (que define cómo se hace referencia en la línea de comando y por otros objetivos), *srcs* (que definen los archivos de origen que se deben compilar para crear el artefacto para el destino), y *deps* (que definen otros objetivos que deben construirse antes de este objetivo y enlazarse con él). Las dependencias pueden estar dentro del mismo paquete (por ejemplo, `MiBinario`'s dependencia de ":milib"), en un paquete diferente en la misma jerarquía de origen (por ejemplo, `milib`'s dependencia de `//java/com/ejemplo/común`), o en un artefacto de terceros fuera de la jerarquía de origen (por ejemplo, `milib`'dependencia de `"@com_google_common_guava_guava//jar"`). Cada jerarquía de fuentes se denomina *trabajo-espacio* y se identifica por la presencia de un especial *ESPACIO DE TRABAJO*/archivo en la raíz.

Al igual que con Ant, los usuarios realizan compilaciones utilizando la herramienta de línea de comandos de Bazel. para construir el `MiBinario` objetivo, un usuario ejecutaría compilación de `bazel: MyBinary`. Al ingresar ese comando por primera vez en un repositorio limpio, Bazel haría lo siguiente:

1. Analizar cada *CONSTRUIR* en el espacio de trabajo para crear un gráfico de dependencias entre artefactos.
2. Usa la gráfica para determinar las *dependencias transitivas* de `MiBinario`; es decir, cada objetivo que `MiBinario` depende y todos los objetivos de los que dependen esos objetivos, recursivamente.
3. Cree (o descargue para dependencias externas) cada una de esas dependencias, en orden. Bazel comienza creando cada destino que no tiene otras dependencias y realiza un seguimiento de las dependencias que aún deben construirse para cada destino. Tan pronto como se construyen todas las dependencias de un destino, Bazel comienza a construir ese destino. Este proceso continúa hasta que cada uno de `MiBinario`'s han construido dependencias transitivas.
4. Construir `MiBinario` para producir un binario ejecutable final que vincula todas las dependencias que se crearon en el paso 3.

Fundamentalmente, puede parecer que lo que está sucediendo aquí no es muy diferente de lo que sucedió al usar un sistema de compilación basado en tareas. De hecho, el resultado final es el mismo binario, y el proceso para producirlo involucró analizar un montón de pasos para encontrar dependencias entre ellos y luego ejecutar esos pasos en orden. Pero hay diferencias críticas. El primero aparece en el paso 3: debido a que Bazel sabe que cada objetivo solo producirá una biblioteca de Java, sabe que todo lo que tiene que hacer es ejecutar el compilador de Java en lugar de un script arbitrario definido por el usuario, por lo que sabe que es seguro correr

estos pasos en paralelo. Esto puede producir una mejora de rendimiento de orden de magnitud sobre los objetivos de construcción de uno en uno en una máquina multinúcleo, y solo es posible porque el enfoque basado en artefactos deja al sistema de construcción a cargo de su propia estrategia de ejecución para que pueda ofrecer garantías más sólidas sobre paralelismo.

Sin embargo, los beneficios se extienden más allá del paralelismo. Lo siguiente que nos brinda este enfoque se vuelve evidente cuando el desarrollador escribe compilación de bazel: MyBinary una segunda vez sin hacer ningún cambio: Bazel saldrá en menos de un segundo con un mensaje que dice que el destino está actualizado. Esto es posible debido al paradigma de programación funcional del que hablamos antes: Bazel sabe que cada destino es el resultado solo de ejecutar un compilador de Java, y sabe que la salida del compilador de Java depende solo de sus entradas, por lo que mientras como las entradas no han cambiado, la salida se puede reutilizar. Y este análisis funciona en todos los niveles; si MiBinario.javacambios, Bazel sabe reconstruir MiBinario pero reutilizar miplib. Si un archivo fuente para //java/com/ejemplo/común cambios, Bazel sabe reconstruir esa biblioteca, miplib, y mi binario, pero reutilizar //java/com/ejemplo/miproducto/otralib. Porque Bazel sabe sobre las propiedades de las herramientas que ejecuta en cada paso, es capaz de reconstruir solo el conjunto mínimo de artefactos cada vez mientras garantiza que no producirá compilaciones obsoletas.

Reformular el proceso de construcción en términos de artefactos en lugar de tareas es sutil pero poderoso. Al reducir la flexibilidad expuesta al programador, el sistema de compilación puede saber más sobre lo que se está haciendo en cada paso de la compilación. Puede usar este conocimiento para hacer que la compilación sea mucho más eficiente al paralelizar los procesos de compilación y reutilizar sus resultados. Pero este es realmente solo el primer paso, y estos bloques de construcción de paralelismo y reutilización formarán la base para un sistema de construcción distribuido y altamente escalable que se discutirá más adelante.

Otros ingeniosos trucos de Bazel

Los sistemas de construcción basados en artefactos resuelven fundamentalmente los problemas de paralelismo y reutilización que son inherentes a los sistemas de construcción basados en tareas. Pero todavía hay algunos problemas que surgieron anteriormente que no hemos abordado. Bazel tiene formas inteligentes de resolver cada uno de estos, y deberíamos discutirlos antes de continuar.

Herramientas como dependencias. Un problema con el que nos encontramos anteriormente fue que las compilaciones dependían de las herramientas instaladas en nuestra máquina, y la reproducción de compilaciones entre sistemas podía ser difícil debido a las diferentes versiones o ubicaciones de las herramientas. El problema se vuelve aún más difícil cuando su proyecto usa lenguajes que requieren diferentes herramientas según la plataforma en la que se construyen o compilan (por ejemplo, Windows versus Linux), y cada una de esas plataformas requiere un conjunto ligeramente diferente de herramientas para hacer el mismo trabajo.

Bazel resuelve la primera parte de este problema al tratar las herramientas como dependencias para cada objetivo. Todos biblioteca_java en el espacio de trabajo depende implícitamente de un compilador de Java, que por defecto es un compilador conocido pero que puede configurarse globalmente en el

nivel del espacio de trabajo. Cada vez que Blaze construye unbiblioteca_java, verifica para asegurarse de que el compilador especificado esté disponible en una ubicación conocida y lo descarga si no es así. Al igual que cualquier otra dependencia, si el compilador de Java cambia, todos los artefactos que dependían de él deberán reconstruirse. Cada tipo de objetivo definido en Bazel usa esta misma estrategia de declarar las herramientas que necesita para ejecutarse, asegurando que Bazel pueda iniciarlas sin importar lo que exista en el sistema donde se ejecuta.

Bazel resuelve la segunda parte del problema, la independencia de la plataforma, utilizando[cadenas de herramientas](#). En lugar de que los objetivos dependan directamente de sus herramientas, en realidad dependen de los tipos de cadenas de herramientas. Una cadena de herramientas contiene un conjunto de herramientas y otras propiedades que definen cómo se construye un tipo de objetivo en una plataforma particular. El espacio de trabajo puede definir la cadena de herramientas particular que se usará para un tipo de cadena de herramientas en función del host y la plataforma de destino. Para obtener más detalles, consulte el manual de Bazel.

Ampliación del sistema de compilación. Bazel viene con objetivos para varios lenguajes de programación populares listos para usar, pero los ingenieros siempre querrán hacer más; parte del beneficio de los sistemas basados en tareas es su flexibilidad para soportar cualquier tipo de proceso de construcción, y sería mejor no renunciar a eso en un sistema de construcción basado en artefactos. Afortunadamente, Bazel permite que sus tipos de objetivos admitidos se amplíen en[agregando reglas personalizadas](#).

Para definir una regla en Bazel, el autor de la regla declara las entradas que requiere la regla (en forma de atributos pasados en elCONSTRUIRArchivo) y el conjunto fijo de salidas que produce la regla. El autor también define la*comportamiento* que será generado por esa regla. Cada acción declara sus entradas y salidas, ejecuta un ejecutable particular o escribe una cadena particular en un archivo, y se puede conectar a otras acciones a través de sus entradas y salidas. Esto significa que las acciones son la unidad componible de nivel más bajo en el sistema de compilación.

— una acción puede hacer lo que quiera siempre que use solo sus entradas y salidas declaradas, y Bazel se encargará de programar acciones y almacenar en caché sus resultados según corresponda.

El sistema no es infalible dado que no hay forma de evitar que un desarrollador de acciones haga algo como introducir un proceso no determinista como parte de su acción. Pero esto no sucede muy a menudo en la práctica, y empujar las posibilidades de abuso hasta el nivel de acción disminuye en gran medida las oportunidades de errores. Las reglas que admiten muchos lenguajes y herramientas comunes están ampliamente disponibles en línea, y la mayoría de los proyectos nunca necesitarán definir sus propias reglas. Incluso para aquellos que lo hacen, las definiciones de reglas solo necesitan definirse en un lugar central en el repositorio, lo que significa que la mayoría de los ingenieros podrán usar esas reglas sin tener que preocuparse por su implementación.

Aislamiento del medio ambiente. Parece que las acciones pueden encontrarse con los mismos problemas que las tareas en otros sistemas. ¿No es posible escribir acciones que escriban en el mismo archivo y terminen en conflicto entre sí? En realidad, Bazel hace que estos conflictos sean imposibles usando[sandboxing](#). En los sistemas compatibles, cada acción es iso-

lated de cualquier otra acción a través de un sandbox de sistema de archivos. Efectivamente, cada acción puede ver solo una vista restringida del sistema de archivos que incluye las entradas que ha declarado y las salidas que ha producido. Esto lo aplican sistemas como LXC en Linux, la misma tecnología detrás de Docker. Esto significa que es imposible que las acciones entren en conflicto entre sí porque no pueden leer ningún archivo que no declaren, y cualquier archivo que escriban pero no declaren se eliminará cuando finalice la acción. Bazel también usa sandboxes para restringir la comunicación de acciones a través de la red.

Hacer que las dependencias externas sean deterministas. Todavía queda un problema pendiente: compilar los sistemas a menudo necesitan descargar dependencias (ya sean herramientas o bibliotecas) de fuentes externas en lugar de construirlas directamente. Esto se puede ver en el ejemplo a través de la `@com_google_common_guava_guava//jardependencia`, que descarga un JAR archivo de Maven.

Depender de archivos fuera del espacio de trabajo actual es arriesgado. Esos archivos podrían cambiar en cualquier momento, lo que podría requerir que el sistema de compilación verifique constantemente si están actualizados. Si un archivo remoto cambia sin un cambio correspondiente en el código fuente del espacio de trabajo, también puede dar lugar a compilaciones irreproducibles: una compilación puede funcionar un día y fallar al siguiente sin motivo aparente debido a un cambio de dependencia inadvertido. Finalmente, una dependencia externa puede presentar un gran riesgo de seguridad cuando es propiedad de un tercero:⁴ si un atacante puede infiltrarse en ese servidor de terceros, puede reemplazar el archivo de dependencia con algo de su propio diseño, lo que potencialmente le otorga un control total sobre su entorno de compilación y su salida.

El problema fundamental es que queremos que el sistema de compilación esté al tanto de estos archivos sin tener que verificarlos en el control de código fuente. La actualización de una dependencia debe ser una elección consciente, pero esa elección debe hacerse una vez en un lugar central en lugar de ser administrada por ingenieros individuales o automáticamente por el sistema. Esto se debe a que incluso con un modelo "Live at Head", aún queremos que las compilaciones sean deterministas, lo que implica que si verifica una confirmación de la semana pasada, debería ver sus dependencias como eran entonces en lugar de como son ahora.

Bazel y algunos otros sistemas de compilación abordan este problema al requerir un archivo de manifiesto para todo el espacio de trabajo que enumere un *hash criptográfico* para cada dependencia externa en el espacio de trabajo.⁵ El hash es una forma concisa de representar de manera única el archivo sin verificar todo el archivo en el control de código fuente. Cada vez que se hace referencia a una nueva dependencia externa desde un espacio de trabajo, el hash de esa dependencia se agrega al manifiesto, ya sea de forma manual o automática. Cuando Bazel ejecuta una compilación, verifica el hash real de su dependencia almacenada en caché con el hash esperado definido en el manifiesto y vuelve a descargar el archivo solo si el hash es diferente.

⁴ Tal "[cadena de suministro de software](#)" Los ataques son cada vez más comunes.

⁵ Ir añadido recientemente [soporte preliminar para módulos que usan exactamente el mismo sistema](#).

Si el artefacto que descargamos tiene un hash diferente al declarado en el manifiesto, la compilación fallará a menos que se actualice el hash en el manifiesto. Esto se puede hacer automáticamente, pero ese cambio debe aprobarse y verificarse en el control de código fuente antes de que la compilación acepte la nueva dependencia. Esto significa que siempre hay un registro de cuándo se actualizó una dependencia y una dependencia externa no puede cambiar sin un cambio correspondiente en la fuente del espacio de trabajo. También significa que, al verificar una versión anterior del código fuente, se garantiza que la compilación usará las mismas dependencias que estaba usando en el momento en que se registró esa versión (o de lo contrario, fallará si esas dependencias se eliminan). ya no está disponible).

Por supuesto, aún puede ser un problema si un servidor remoto deja de estar disponible o comienza a entregar datos corruptos; esto puede causar que todas sus compilaciones comiencen a fallar si no tiene otra copia de esa dependencia disponible. Para evitar este problema, recomendamos que, para cualquier proyecto no trivial, refleje todas sus dependencias en servidores o servicios en los que confie y controle. De lo contrario, siempre estará a merced de un tercero para la disponibilidad de su sistema de compilación, incluso si los hashes registrados garantizan su seguridad.

Construcciones distribuidas

La base de código de Google es enorme: con más de dos mil millones de líneas de código, las cadenas de dependencias pueden llegar a ser muy profundas. Incluso los binarios simples en Google a menudo dependen de decenas de miles de objetivos de compilación. A esta escala, es simplemente imposible completar una compilación en un tiempo razonable en una sola máquina: ningún sistema de compilación puede eludir las leyes fundamentales de la física impuestas al hardware de una máquina. La única forma de hacer que esto funcione es con un sistema de compilación que admita *compilaciones distribuidas* en el que las unidades de trabajo que realiza el sistema se distribuyen en un número arbitrario y escalable de máquinas. Suponiendo que hayamos dividido el trabajo del sistema en unidades lo suficientemente pequeñas (más sobre esto más adelante), esto nos permitiría completar cualquier construcción de cualquier tamaño tan rápido como estemos dispuestos a pagar. Esta escalabilidad es el santo grial por el que hemos estado trabajando al definir un sistema de compilación basado en artefactos.

Almacenamiento en caché remoto

El tipo más simple de compilación distribuida es aquella que solo aprovecha *almacenamiento en caché remoto*, que se muestra en [Figura 18-2](#).

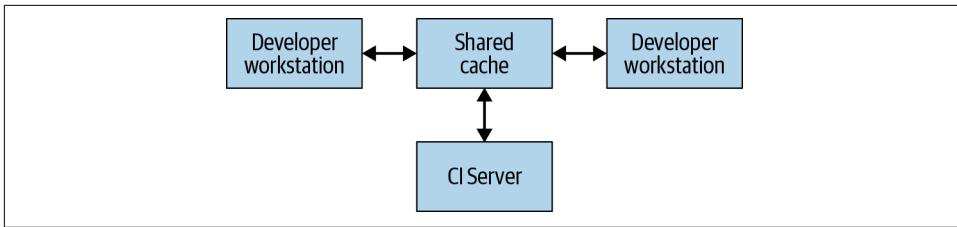


Figura 18-2. Una compilación distribuida que muestra el almacenamiento en caché remoto

Todos los sistemas que realizan compilaciones, incluidas las estaciones de trabajo de desarrolladores y los sistemas de integración continua, comparten una referencia a un servicio de caché remoto común. Este servicio puede ser un sistema de almacenamiento rápido y local a corto plazo como Redis o un servicio en la nube como Google Cloud Storage. Cada vez que un usuario necesita construir un artefacto, ya sea directamente o como una dependencia, el sistema primero verifica con el caché remoto para ver si ese artefacto ya existe allí. Si es así, puede descargar el artefacto en lugar de construirlo. De lo contrario, el sistema crea el artefacto por sí mismo y vuelve a cargar el resultado en la memoria caché. Esto significa que las dependencias de bajo nivel que no cambian muy a menudo se pueden crear una vez y compartir entre los usuarios en lugar de tener que reconstruirlas cada usuario. En Google, muchos artefactos se sirven desde un caché en lugar de crearlos desde cero.

Para que un sistema de almacenamiento en caché remoto funcione, el sistema de compilación debe garantizar que las compilaciones sean completamente reproducibles. Es decir, para cualquier objetivo de compilación, debe ser posible determinar el conjunto de entradas para ese objetivo de modo que el mismo conjunto de entradas produzca exactamente el mismo resultado en cualquier máquina. Esta es la única forma de garantizar que los resultados de descargar un artefacto sean los mismos que los resultados de construirlo uno mismo. Afortunadamente, Bazel brinda esta garantía y, por lo tanto, apoya [almacenamiento en caché remoto](#). Tenga en cuenta que esto requiere que cada artefacto en el caché tenga una clave tanto en su objetivo como en un hash de sus entradas; de esa manera, diferentes ingenieros podrían realizar diferentes modificaciones en el mismo objetivo al mismo tiempo, y el caché remoto almacenaría todo el artefactos resultantes y servirlos apropiadamente sin conflicto.

Por supuesto, para que haya algún beneficio de un caché remoto, la descarga de un artefacto debe ser más rápida que su construcción. Esto no siempre es el caso, especialmente si el servidor de caché está lejos de la máquina que realiza la compilación. La red y el sistema de compilación de Google se ajustan cuidadosamente para poder compartir rápidamente los resultados de la compilación. Al configurar el almacenamiento en caché remoto en su organización, tenga cuidado de considerar las latencias de la red y realice experimentos para asegurarse de que el caché realmente esté mejorando el rendimiento.

Ejecución remota

El almacenamiento en caché remoto no es una verdadera compilación distribuida. Si se pierde el caché o si realiza un cambio de bajo nivel que requiere que se reconstruya todo, aún debe realizar la compilación completa localmente en su máquina. El verdadero objetivo es apoyar *ejecución remota*, en el cual

el trabajo real de hacer la construcción se puede repartir entre cualquier número de trabajadores.

Figura 18-3 representa un sistema de ejecución remota.

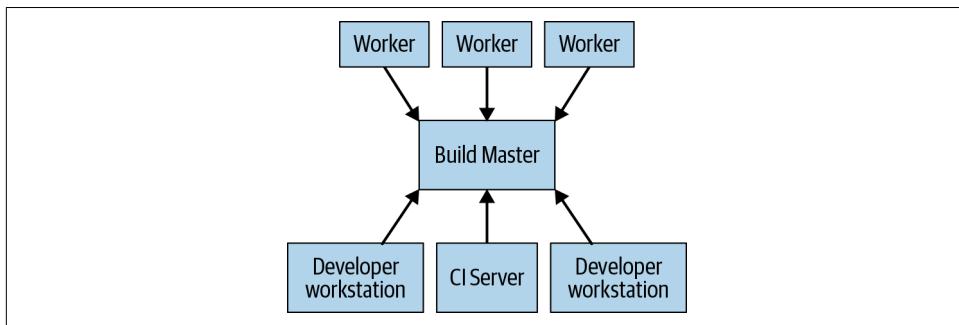


Figura 18-3. Un sistema de ejecución remota

La herramienta de compilación que se ejecuta en la máquina de cada usuario (donde los usuarios son ingenieros humanos o sistemas de compilación automatizados) envía solicitudes a un maestro de compilación central. El maestro de compilación divide las solicitudes en sus acciones de componentes y programa la ejecución de esas acciones en un grupo escalable de trabajadores. Cada trabajador realiza las acciones que se le solicitan con las entradas especificadas por el usuario y escribe los artefactos resultantes. Estos artefactos se comparten entre las otras máquinas que ejecutan acciones que los requieren hasta que se puede producir el resultado final y enviarlo al usuario.

La parte más complicada de implementar un sistema de este tipo es administrar la comunicación entre los trabajadores, el maestro y la máquina local del usuario. Los trabajadores pueden depender de artefactos intermedios producidos por otros trabajadores, y el resultado final debe enviarse de vuelta a la máquina local del usuario. Para hacer esto, podemos construir sobre la memoria caché distribuida descrita anteriormente haciendo que cada trabajador escriba sus resultados y lea sus dependencias de la memoria caché. El maestro impide que los trabajadores continúen hasta que todo de lo que dependen haya terminado, en cuyo caso podrán leer sus entradas desde el caché. El producto final también se almacena en caché, lo que permite que la máquina local lo descargue. Tenga en cuenta que también necesitamos un medio separado para exportar los cambios locales en el árbol de fuentes del usuario para que los trabajadores puedan aplicar esos cambios antes de construir.

Para que esto funcione, todas las partes de los sistemas de compilación basados en artefactos descritos anteriormente deben unirse. Los entornos de construcción deben ser completamente autodescriptivos para que podamos hacer girar a los trabajadores sin intervención humana. Los procesos de compilación en sí mismos deben ser completamente autónomos porque cada paso puede ejecutarse en una máquina diferente. Los resultados deben ser completamente deterministas para que cada trabajador pueda confiar en los resultados que recibe de otros trabajadores. Tales garantías son extremadamente difíciles de proporcionar para un sistema basado en tareas, lo que hace que sea casi imposible construir un sistema de ejecución remota confiable encima de uno.

Compilaciones distribuidas en Google. Desde 2008, Google ha estado utilizando un sistema de compilación distribuido que emplea almacenamiento en caché remoto y ejecución remota, que se ilustra en Figura 18-4.

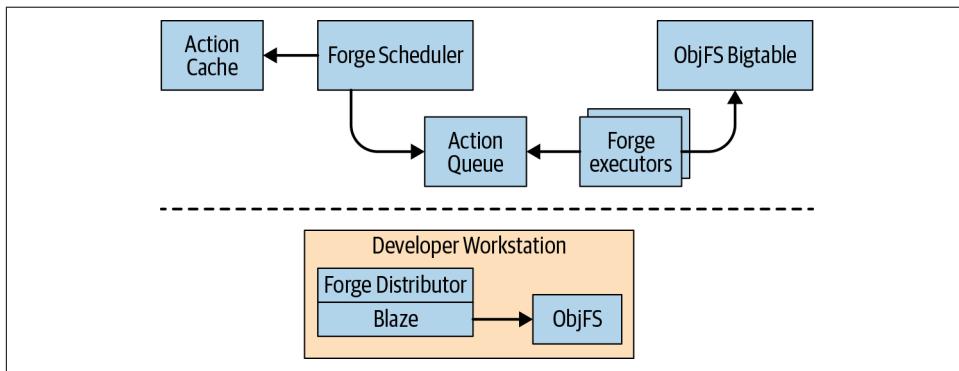


Figura 18-4. Sistema de compilación distribuida de Google

El caché remoto de Google se llama ObjFS. Consiste en un backend que almacena los resultados de compilación en **mesas grandes** distribuidos a lo largo de nuestra flota de máquinas de producción y un demonio FUSE frontend llamado objfsd que se ejecuta en la máquina de cada desarrollador. El demonio FUSE permite a los ingenieros explorar los resultados de la compilación como si fueran archivos normales almacenados en la estación de trabajo, pero con el contenido del archivo descargado a pedido solo para los pocos archivos que el usuario solicita directamente. La entrega de contenidos de archivos bajo demanda reduce en gran medida el uso de la red y del disco, y el sistema es capaz de **construir el doble de rápido** en comparación con cuando almacenamos todos los resultados de compilación en el disco local del desarrollador.

El sistema de ejecución remota de Google se llama Forge. Un cliente de Forge en Blaze llamado Distribuidor envía solicitudes para cada acción a un trabajo que se ejecuta en nuestros centros de datos llamado Programador. El Programador mantiene una memoria caché de los resultados de las acciones, lo que le permite devolver una respuesta inmediatamente si la acción ya ha sido creada por otro usuario del sistema. Si no, coloca la acción en una cola. Un gran grupo de trabajos de Ejecutor lee continuamente acciones de esta cola, las ejecuta y almacena los resultados directamente en ObjFS Bigtables. Estos resultados están disponibles para los ejecutores para futuras acciones, o para ser descargados por el usuario final a través de objfsd.

El resultado final es un sistema que se escala para admitir de manera eficiente todas las compilaciones realizadas en Google. Y la escala de las compilaciones de Google es realmente enorme: Google ejecuta millones de compilaciones, ejecuta millones de casos de prueba y produce petabytes de resultados de compilación a partir de miles de millones de líneas de código fuente cada año. Dicho sistema no solo permite a nuestros ingenieros crear bases de código complejas rápidamente, sino que también nos permite implementar una gran cantidad de herramientas y sistemas automatizados que dependen de nuestra compilación. Invertimos muchos años de esfuerzo en el desarrollo de este sistema, pero hoy en día las herramientas de código abierto están fácilmente disponibles para que cualquier organización pueda implementar un sistema similar. Aunque puede tomar tiempo

y energía para implementar un sistema de construcción de este tipo, el resultado final puede ser verdaderamente mágico para los ingenieros y, a menudo, bien vale la pena el esfuerzo.

Tiempo, escala, compensaciones

Los sistemas de compilación tienen que ver con hacer que el código sea más fácil de trabajar a escala y con el tiempo. Y como todo en la ingeniería de software, existen ventajas y desventajas al elegir qué tipo de sistema de compilación usar. El enfoque de bricolaje que usa scripts de shell o invocaciones directas de herramientas funciona solo para los proyectos más pequeños que no necesitan lidiar con el cambio de código durante un largo período de tiempo, o para lenguajes como Go que tienen un sistema de compilación incorporado.

Elegir un sistema de compilación basado en tareas en lugar de confiar en scripts de bricolaje mejora en gran medida la capacidad de escalar de su proyecto, lo que le permite automatizar compilaciones complejas y reproducir más fácilmente esas compilaciones en todas las máquinas. La compensación es que debe comenzar a pensar en cómo se estructura su compilación y lidiar con la sobrecarga de escribir archivos de compilación (aunque las herramientas automatizadas a menudo pueden ayudar con esto). Esta compensación tiende a valer la pena para la mayoría de los proyectos, pero para proyectos particularmente triviales (por ejemplo, aquellos contenidos en un solo archivo fuente), la sobrecarga podría no ser suficiente para usted.

Los sistemas de compilación basados en tareas comienzan a encontrarse con algunos problemas fundamentales a medida que el proyecto se amplía aún más, y estos problemas se pueden remediar mediante el uso de un sistema de compilación basado en artefactos. Dichos sistemas de compilación desbloquean un nivel completamente nuevo de escala porque las compilaciones enormes ahora se pueden distribuir en muchas máquinas, y miles de ingenieros pueden estar más seguros de que sus compilaciones son consistentes y reproducibles. Al igual que con tantos otros temas de este libro, la compensación aquí es la falta de flexibilidad: los sistemas basados en artefactos no le permiten escribir tareas genéricas en un lenguaje de programación real, sino que requieren que trabaje dentro de las limitaciones del sistema. Por lo general, esto no es un problema para los proyectos que están diseñados para trabajar con sistemas basados en artefactos desde el principio.

Los cambios en el sistema de compilación de un proyecto pueden ser costosos, y ese costo aumenta a medida que el proyecto se vuelve más grande. Es por eso que Google cree que casi todos los proyectos nuevos se benefician de la incorporación de un sistema de compilación basado en artefactos como Bazel desde el principio. Dentro de Google, esencialmente todo el código, desde pequeños proyectos experimentales hasta Google Search, se crea con Blaze.

Manejo de módulos y dependencias

Los proyectos que usan sistemas de compilación basados en artefactos como Bazel se dividen en un conjunto de módulos, con módulos que expresan dependencias entre sí a través de *CONSTRUIArchivos Adecuado*

La organización de estos módulos y dependencias puede tener un gran efecto tanto en el rendimiento del sistema de compilación como en la cantidad de trabajo que se necesita para mantenerlo.

Uso de módulos granulares y la regla 1:1:1

La primera pregunta que surge al estructurar una compilación basada en artefactos es decidir cuánta funcionalidad debe abarcar un módulo individual. En Bazel, un “módulo” está representado por un objetivo que especifica una unidad construible como `unbiblioteca_javao` un `ir_binario`. En un extremo, todo el proyecto podría estar contenido en un solo módulo colocando un archivo `BUILD` en la raíz y reuniendo recursivamente todos los archivos fuente de ese proyecto. En el otro extremo, casi todos los archivos de origen podrían convertirse en su propio módulo, lo que requeriría efectivamente que cada archivo se incluya en una lista. *CONSTRUIR*archivar todos los demás archivos de los que depende.

La mayoría de los proyectos se encuentran en algún lugar entre estos extremos, y la elección implica un equilibrio entre el rendimiento y la mantenibilidad. El uso de un solo módulo para todo el proyecto puede significar que nunca tendrá que tocar el *CONSTRUIR*excepto cuando se agrega una dependencia externa, pero significa que el sistema de compilación siempre necesitará compilar todo el proyecto a la vez. Esto significa que no podrá parallelizar o distribuir partes de la compilación, ni podrá almacenar en caché partes que ya están compiladas. Un módulo por archivo es lo contrario: el sistema de compilación tiene la máxima flexibilidad en el almacenamiento en caché y la programación de los pasos de la compilación, pero los ingenieros deben esforzarse más en mantener las listas de dependencias cada vez que cambian qué archivos hacen referencia a qué.

Aunque la granularidad exacta varía según el idioma (y, a menudo, incluso dentro del idioma), Google tiende a favorecer módulos significativamente más pequeños de lo que normalmente se escribiría en un sistema de compilación basado en tareas. Un binario de producción típico en Google probablemente dependerá de decenas de miles de objetivos, e incluso un equipo de tamaño moderado puede poseer varios cientos de objetivos dentro de su base de código. Para lenguajes como Java que tienen una fuerte noción integrada de empaquetado, cada directorio normalmente contiene un solo paquete, destino y *CONSTRUIR*archivo (`Pants`, otro sistema de compilación basado en Blaze, lo llama la *regla 1:1:1*). Los lenguajes con convenciones de empaquetado más débiles con frecuencia definirán múltiples objetivos por *CONSTRUIR*expediente.

Los beneficios de los objetivos de compilación más pequeños realmente comienzan a mostrarse a escala porque conducen a compilaciones distribuidas más rápidas y una necesidad menos frecuente de reconstruir objetivos. Las ventajas se vuelven aún más convincentes después de que las pruebas entran en escena, ya que los objetivos más detallados significan que el sistema de compilación puede ser mucho más inteligente al ejecutar solo un subconjunto limitado de pruebas que podrían verse afectadas por cualquier cambio dado. Debido a que Google cree en los beneficios sistémicos de usar objetivos más pequeños, hemos logrado algunos avances para mitigar las desventajas al invertir en herramientas para administrar automáticamente *CONSTRUIR*archivos para evitar sobrecargar a los desarrolladores. muchos de [estas herramientas](#)ahora son de código abierto.

Minimización de la visibilidad del módulo

Bazel y otros sistemas de compilación permiten que cada objetivo especifique una visibilidad: una propiedad que especifica qué otros objetivos pueden depender de ella. Los objetivos pueden ser público, en cuyo caso pueden ser referenciados por cualquier otro objetivo en el espacio de trabajo; privado, en cuyo caso sólo pueden ser referenciados desde dentro del mismo *CONSTRUIR*/expediente; o visible solo para una lista explícitamente definida de otros objetivos. Una visibilidad es esencialmente lo opuesto a una dependencia: si el objetivo A quiere depender del objetivo B, el objetivo B debe hacerse visible para el objetivo A.

Al igual que en la mayoría de los lenguajes de programación, generalmente es mejor minimizar la visibilidad tanto como sea posible. Por lo general, los equipos de Google harán que los objetivos sean públicos solo si esos objetivos representan bibliotecas ampliamente utilizadas disponibles para cualquier equipo de Google. Los equipos que requieren que otros se coordinen con ellos antes de usar su código mantendrán una lista blanca de objetivos de clientes como la visibilidad de su objetivo. Los objetivos de implementación internos de cada equipo se limitarán únicamente a directorios propiedad del equipo, y la mayoría *CONSTRUIR* los archivos tendrán un solo destino que no es privado.

Gestión de dependencias

Los módulos deben poder referirse entre sí. La desventaja de dividir una base de código en módulos detallados es que necesita administrar las dependencias entre esos módulos (aunque las herramientas pueden ayudar a automatizar esto). Expresar estas dependencias por lo general termina siendo la mayor parte del contenido en un *CONSTRUIR*/expediente.

Dependencias internas

En un proyecto grande dividido en módulos detallados, es probable que la mayoría de las dependencias sean internas; es decir, en otro objetivo definido y construido en el mismo repositorio fuente. Las dependencias internas se diferencian de las dependencias externas en que se compilan desde el origen en lugar de descargarse como un artefacto preconstruido mientras se ejecuta la compilación. Esto también significa que no existe la noción de "versión" para las dependencias internas: un destino y todas sus dependencias internas siempre se crean en la misma confirmación/revisión en el repositorio.

Una cuestión que debe manejarse con cuidado con respecto a las dependencias internas es cómo tratar *dependencias transitivas* ([Figura 18-5](#)). Supongamos que el objetivo A depende del objetivo B, que depende de un objetivo de biblioteca común C. ¿Debería el objetivo A poder usar las clases definidas en el objetivo C?

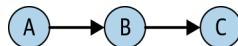


Figura 18-5. Dependencias transitivas

En lo que respecta a las herramientas subyacentes, no hay problema con esto; tanto B como C estarán vinculados al objetivo A cuando se construya, por lo que cualquier símbolo definido en C es conocido por A. Blaze permitió esto durante muchos años, pero a medida que Google creció, comenzamos a ver problemas. Supongamos que B se refactorizó de tal manera que ya no necesitaba depender de C. Si luego se eliminara la dependencia de B de C, A y cualquier otro destino que usara C a través de una dependencia de B se romperían. Efectivamente, las dependencias de un objetivo se convirtieron en parte de su contrato público y nunca se pudieron cambiar de manera segura. Esto significó que las dependencias se acumularon con el tiempo y las compilaciones en Google comenzaron a ralentizarse.

Google finalmente resolvió este problema al introducir un "modo de dependencia transitiva estricta" en Blaze. En este modo, Blaze detecta si un objetivo intenta hacer referencia a un símbolo sin depender directamente de él y, si es así, falla con un error y un comando de shell que se puede usar para insertar automáticamente la dependencia. Implementar este cambio en todo el código base de Google y refactorizar cada uno de nuestros millones de objetivos de compilación para enumerar explícitamente sus dependencias fue un esfuerzo de varios años, pero valió la pena. Nuestras compilaciones ahora son mucho más rápidas dado que los objetivos tienen menos dependencias innecesarias,⁶ y los ingenieros están facultados para eliminar las dependencias que no necesitan sin preocuparse por romper los objetivos que dependen de ellos.

Como de costumbre, hacer cumplir dependencias transitivas estrictas implicó una compensación. Hizo que los archivos de compilación fueran más detallados, ya que las bibliotecas de uso frecuente ahora deben enumerarse explícitamente en muchos lugares en lugar de extraerse de manera incidental, y los ingenieros deben esforzarse más para agregar dependencias a *CONSTRUIR*Archivos. Desde entonces, hemos desarrollado herramientas que reducen este trabajo al detectar automáticamente muchas dependencias que faltan y agregarlas a un *CONSTRUIR*Archivos sin ninguna intervención del desarrollador. Pero incluso sin tales herramientas, hemos encontrado que la compensación vale la pena a medida que la base de código escala: agregar explícitamente una dependencia a *CONSTRUIREl* archivo es un costo único, pero lidiar con dependencias transitivas implícitas puede causar problemas continuos siempre que exista el objetivo de compilación. **Bazel impone dependencias transitivas estrictas**en código Java por defecto.

Dependencias externas

Si una dependencia no es interna, debe ser externa. Las dependencias externas son las de los artefactos que se compilan y almacenan fuera del sistema de compilación. La dependencia se importa directamente desde un *repositorio de artefactos*(generalmente se accede a través de Internet) y se usa tal cual en lugar de construirse desde la fuente. Una de las mayores diferencias entre las dependencias externas e internas es que las dependencias externas tienen *versiones*, y esas versiones existen independientemente del código fuente del proyecto.

6 Por supuesto, en realidad *quitando*estas dependencias fue un proceso completamente separado. Pero exigir que cada objetivo declarar explícitamente lo que utilizó fue un primer paso crítico. Ver [capítulo 22](#)para obtener más información sobre cómo Google realiza cambios a gran escala como este.

Gestión de dependencias automática versus manual. Los sistemas de compilación pueden permitir las versiones de dependencias externas para gestionar de forma manual o automática. Cuando se administra manualmente, el archivo de compilación enumera explícitamente la versión que desea descargar del repositorio de artefactos, a menudo usando una cadena de versión semántica como "1.1.4". Cuando se administra automáticamente, el archivo fuente especifica un rango de versiones aceptables y el sistema de compilación siempre descarga la última. Por ejemplo, Gradle permite que una versión de dependencia se declare como "1.+" para especificar que cualquier versión secundaria o parche de una dependencia es aceptable siempre que la versión principal sea 1.

Las dependencias administradas automáticamente pueden ser convenientes para proyectos pequeños, pero generalmente son una receta para el desastre en proyectos de tamaño no trivial o en los que está trabajando más de un ingeniero. El problema con las dependencias administradas automáticamente es que no tiene control sobre cuándo se actualiza la versión. No hay forma de garantizar que las partes externas no realicen actualizaciones de última hora (incluso cuando afirman usar el control de versiones semántico), por lo que una compilación que funcionó un día podría fallar al siguiente sin una forma fácil de detectar qué cambió o revertirlo a un estado de trabajo. Incluso si la compilación no falla, puede haber cambios sutiles en el comportamiento o el rendimiento que son imposibles de rastrear.

Por el contrario, debido a que las dependencias administradas manualmente requieren un cambio en el control de código fuente, se pueden descubrir y revertir fácilmente, y es posible verificar una versión anterior del repositorio para compilar con dependencias más antiguas. Bazel requiere que las versiones de todas las dependencias se especifiquen manualmente. Incluso a escalas moderadas, los gastos generales de la gestión manual de versiones valen la pena por la estabilidad que proporciona.

La regla de una versión. Las diferentes versiones de una biblioteca generalmente están representadas por diferentes artefactos, por lo que, en teoría, no hay razón para que las diferentes versiones de la misma dependencia externa no puedan declararse en el sistema de compilación con diferentes nombres. De esa forma, cada objetivo podría elegir qué versión de la dependencia quería usar. Google ha descubierto que esto causa muchos problemas en la práctica, por lo que aplicamos un estricto *Regla de una versión* para todas las dependencias de terceros en nuestra base de código interna.

El mayor problema de permitir múltiples versiones es la *dependencia de diamantes* asunto. Suponga que el objetivo A depende del objetivo B y de v1 de una biblioteca externa. Si el objetivo B se refactoriza posteriormente para agregar una dependencia en v2 de la misma biblioteca externa, el objetivo A se romperá porque ahora depende implícitamente de dos versiones diferentes de la misma biblioteca. Efectivamente, nunca es seguro agregar una nueva dependencia de un objetivo a una biblioteca de terceros con varias versiones, porque cualquiera de los usuarios de ese objetivo ya podría estar dependiendo de una versión diferente. Seguir la Regla de una versión hace que este conflicto sea imposible: si un destino agrega una dependencia en una biblioteca de terceros, cualquier dependencia existente ya estará en esa misma versión, por lo que pueden coexistir felizmente.

Examinaremos esto más a fondo en el contexto de un gran monorepo en [capítulo 21](#).

Dependencias externas transitivas. Tratar con las dependencias transitivas de una dependencia externa puede ser particularmente difícil. Muchos repositorios de artefactos como Maven Central permiten que los artefactos especifiquen dependencias en versiones particulares de otros artefactos en el repositorio. Las herramientas de compilación como Maven o Gradle a menudo descargarán recursivamente cada dependencia transitiva de forma predeterminada, lo que significa que agregar una sola dependencia en su proyecto podría causar la descarga de docenas de artefactos en total.

Esto es muy conveniente: al agregar una dependencia en una nueva biblioteca, sería un gran dolor tener que rastrear cada una de las dependencias transitivas de esa biblioteca y agregarlas todas manualmente. Pero también hay una gran desventaja: debido a que diferentes bibliotecas pueden depender de diferentes versiones de la misma biblioteca de terceros, esta estrategia necesariamente viola la regla de una versión y conduce al problema de dependencia de diamantes. Si su objetivo depende de dos bibliotecas externas que usan diferentes versiones de la misma dependencia, no se sabe cuál obtendrá. Esto también significa que la actualización de una dependencia externa podría causar fallas aparentemente no relacionadas en todo el código base si la nueva versión comienza a generar versiones en conflicto de algunas de sus dependencias.

Por esta razón, Bazel no descarga automáticamente las dependencias transitivas. Y, desafortunadamente, no hay una bala de plata: la alternativa de Bazel es requerir un archivo global que enumere cada una de las dependencias externas del repositorio y una versión explícita utilizada para esa dependencia en todo el repositorio. Afortunadamente, [Bazel proporciona herramientas](#) que pueden generar automáticamente dicho archivo que contiene las dependencias transitivas de un conjunto de artefactos de Maven. Esta herramienta se puede ejecutar una vez para generar la inicial *ESPACIO DE TRABAJO* para un proyecto, y ese archivo se puede actualizar manualmente para ajustar las versiones de cada dependencia.

Una vez más, la elección aquí es entre conveniencia y escalabilidad. Los proyectos pequeños pueden preferir no tener que preocuparse por administrar las dependencias transitivas y pueden salirse con la suya usando dependencias transitivas automáticas. Esta estrategia se vuelve cada vez menos atractiva a medida que crece la organización y la base de código, y los conflictos y los resultados inesperados se vuelven cada vez más frecuentes. A escalas más grandes, el costo de administrar manualmente las dependencias es mucho menor que el costo de lidiar con los problemas causados por la administración automática de dependencias.

Almacenamiento en caché de los resultados de compilación mediante dependencias externas. Las dependencias externas son más a menudo proporcionado por terceros que lanzan versiones estables de bibliotecas, quizás sin proporcionar el código fuente. Algunas organizaciones también pueden optar por hacer que parte de su propio código esté disponible como artefactos, lo que permite que otras piezas de código dependan de ellos como terceros en lugar de dependencias internas. En teoría, esto puede acelerar las construcciones si los artefactos son lentos para construir pero rápidos para descargar.

Sin embargo, esto también presenta muchos gastos generales y complejidad: alguien debe ser responsable de construir cada uno de esos artefactos y cargarlos en el artefacto.

repositorio, y los clientes deben asegurarse de estar actualizados con la última versión. La depuración también se vuelve mucho más difícil porque diferentes partes del sistema se habrán creado desde diferentes puntos en el repositorio y ya no hay una vista consistente del árbol fuente.

Una mejor manera de resolver el problema de los artefactos que tardan mucho tiempo en compilarse es utilizar un sistema de compilación que admita el almacenamiento en caché remoto, como se describió anteriormente. Dicho sistema de compilación guardará los artefactos resultantes de cada compilación en una ubicación compartida entre ingenieros, por lo que si un desarrollador depende de un artefacto que otra persona construyó recientemente, el sistema de compilación lo descargará automáticamente en lugar de compilarlo. Esto proporciona todos los beneficios de rendimiento de depender directamente de los artefactos y, al mismo tiempo, garantiza que las construcciones sean tan consistentes como si siempre se hubieran creado a partir de la misma fuente. Esta es la estrategia utilizada internamente por Google, y Bazel se puede configurar para usar un caché remoto.

Seguridad y confiabilidad de las dependencias externas. Dependiendo de los artefactos de terceros fuentes del partido es inherentemente riesgoso. Existe un riesgo de disponibilidad si la fuente de terceros (por ejemplo, un repositorio de artefactos) deja de funcionar, ya que toda la compilación podría detenerse si no puede descargar una dependencia externa. También existe un riesgo de seguridad: si un atacante compromete el sistema de terceros, el atacante podría reemplazar el artefacto al que se hace referencia con uno de su propio diseño, lo que les permitiría injectar código arbitrario en su compilación.

Ambos problemas se pueden mitigar reflejando los artefactos de los que depende en los servidores que controla y bloqueando su sistema de compilación para que no acceda a repositorios de artefactos de terceros como Maven Central. La contrapartida es que estos espejos requieren esfuerzo y recursos para mantenerlos, por lo que la elección de usarlos a menudo depende de la escala del proyecto. El problema de seguridad también se puede evitar por completo con poca sobrecarga al requerir que se especifique el hash de cada artefacto de terceros en el repositorio de origen, lo que hace que la compilación falle si se manipula el artefacto.

Otra alternativa que elude por completo el problema es *vender* las dependencias de su proyecto. Cuando un proyecto vende sus dependencias, las verifica en el control de código fuente junto con el código fuente del proyecto, ya sea como fuente o como binarios. Efectivamente, esto significa que todas las dependencias externas del proyecto se convierten en dependencias internas. Google usa este enfoque internamente, verificando cada biblioteca de terceros a la que se hace referencia en Google en un *tercerodirectorio* en la raíz del árbol de fuentes de Google. Sin embargo, esto funciona en Google solo porque el sistema de control de fuente de Google está diseñado a la medida para manejar un monorepo extremadamente grande, por lo que la venta podría no ser una opción para otras organizaciones.

Conclusión

Un sistema de construcción es una de las partes más importantes de una organización de ingeniería. Cada desarrollador interactuará con él potencialmente docenas o cientos de veces al día y, en muchas situaciones, puede ser el paso limitante para determinar su productividad. Esto significa que vale la pena invertir tiempo y pensar en hacer las cosas bien.

Como se discutió en este capítulo, una de las lecciones más sorprendentes que Google ha aprendido es que *limitar el poder y la flexibilidad de los ingenieros puede mejorar su productividad*. Pudimos desarrollar un sistema de compilación que satisface nuestras necesidades no dando a los ingenieros la libertad de definir cómo se realizan las compilaciones, sino desarrollando un marco altamente estructurado que limita la elección individual y deja las decisiones más interesantes en manos de herramientas automatizadas. Y a pesar de lo que pueda pensar, a los ingenieros no les molesta esto: a los empleados de Google les encanta que este sistema funcione principalmente por sí solo y les permita concentrarse en las partes interesantes de escribir sus aplicaciones en lugar de lidiar con la lógica de compilación. Poder confiar en la compilación es poderoso: las compilaciones incrementales simplemente funcionan, y casi nunca es necesario borrar los cachés de compilación o ejecutar un paso "limpio".

Tomamos esta idea y la usamos para crear un tipo completamente nuevo *de basado en artefactos* sistema constructivo, contrastando con el tradicional *basado en tareas* construir sistemas. Este replanteamiento de la compilación centrándose en artefactos en lugar de tareas es lo que permite que nuestras compilaciones se amplíen a una organización del tamaño de Google. En el extremo, permite un sistema de construcción distribuido que es capaz de aprovechar los recursos de un clúster informático completo para acelerar la productividad de los ingenieros. Si bien es posible que su organización no sea lo suficientemente grande como para beneficiarse de una inversión de este tipo, creemos que los sistemas de compilación basados en artefactos se escalan tanto hacia abajo como hacia arriba: incluso para proyectos pequeños, los sistemas de compilación como Bazel pueden brindar beneficios significativos en términos de rapidez y corrección.

El resto de este capítulo exploró cómo administrar las dependencias en un mundo basado en artefactos. Llegamos a la conclusión de que *los módulos de grano fino escalan mejor que los módulos de grano grueso*. También discutimos las dificultades de administrar versiones de dependencia, describiendo el *Oregla de la nueva versión* y la observación de que todas las dependencias deben ser *versionado manual y explícitamente*. Tales prácticas evitan errores comunes como el problema de la dependencia de los diamantes y permiten que una base de código alcance la escala de Google de miles de millones de líneas de código en un solo repositorio con un sistema de compilación unificado.

TL; DR

- Es necesario un sistema de compilación con todas las funciones para mantener la productividad de los desarrolladores a medida que crece la organización.
- El poder y la flexibilidad tienen un costo. Restringir el sistema de compilación de manera adecuada lo hace más fácil para los desarrolladores.

- Los sistemas de compilación organizados en torno a artefactos tienden a escalar mejor y a ser más confiables que los sistemas de compilación organizados en torno a tareas.
- Al definir artefactos y dependencias, es mejor apuntar a módulos detallados. Los módulos detallados pueden aprovechar mejor el paralelismo y las compilaciones incrementales.
- Las dependencias externas se deben versionar explícitamente bajo el control de código fuente. Confiar en las versiones "más recientes" es una receta para el desastre y las compilaciones irreproducibles.

Crítica: herramienta de revisión de código de Google

Escrito por Caitlin Sadowski, Ilham Kurnia y Ben Rohlfs
Editado por Lisa Carey

como viste en [Capítulo 9](#), la revisión de código es una parte vital del desarrollo de software, particularmente cuando se trabaja a escala. El objetivo principal de la revisión del código es mejorar la legibilidad y la mantenibilidad del código base, y esto se apoya fundamentalmente en el proceso de revisión. Sin embargo, tener un proceso de revisión de código bien definido en solo una parte de la historia de revisión de código. Las herramientas que respaldan ese proceso también juegan un papel importante en su éxito.

En este capítulo, veremos qué hace que las herramientas de revisión de código sean exitosas a través del apreciado sistema interno de Google, *Crítica*. La crítica tiene soporte explícito para las motivaciones principales de la revisión del código, proporcionando a los revisores y autores una vista de la revisión y la capacidad de comentar sobre el cambio. Critique también tiene soporte para controlar qué código se verifica en la base de código, discutido en la sección sobre cambios de "puntuación". La información de revisión de código de Critique también puede ser útil al hacer arqueología de código, siguiendo algunas decisiones técnicas que se explican en las interacciones de revisión de código (p. ej., cuando faltan comentarios en línea). Aunque Critique no es la única herramienta de revisión de código utilizada en Google, es la más popular por un amplio margen.

Principios de herramientas de revisión de código

Anteriormente mencionamos que Critique brinda funcionalidad para respaldar los objetivos de la revisión de código (veremos esta funcionalidad con más detalle más adelante en este capítulo), pero ¿por qué tiene tanto éxito? La crítica ha sido moldeada por la cultura de desarrollo de Google, que incluye la revisión de código como parte central del flujo de trabajo. Esta influencia cultural se traduce en un conjunto de principios rectores que Critique fue diseñado para enfatizar:

Sencillez

La interfaz de usuario (IU) de Critique se basa en facilitar la revisión del código sin muchas opciones innecesarias y con una interfaz fluida. La interfaz de usuario se carga rápidamente, la navegación es fácil y admite teclas de acceso rápido, y hay marcadores visuales claros para el estado general de si se ha revisado un cambio.

Fundación de confianza

La revisión de código no es para ralentizar a otros; en cambio, es para empoderar a otros. Confiar en los colegas tanto como sea posible hace que funcione. Esto podría significar, por ejemplo, confiar en los autores para realizar cambios y no requerir una fase de revisión adicional para verificar que los comentarios menores realmente se aborden. La confianza también se manifiesta al hacer que los cambios sean accesibles abiertamente (para verlos y revisarlos) en Google.

Comunicación genérica

Los problemas de comunicación rara vez se resuelven mediante herramientas. Critique prioriza formas genéricas para que los usuarios comenten sobre los cambios de código, en lugar de protocolos complicados. Critique alienta a los usuarios a explicar lo que quieren en sus comentarios o incluso sugiere algunas ediciones en lugar de hacer que el modelo de datos y el proceso sean más complejos. La comunicación puede salir mal incluso con la mejor herramienta de revisión de código porque los usuarios son humanos.

Integración de flujo de trabajo

Critique tiene varios puntos de integración con otras herramientas básicas de desarrollo de software. Los desarrolladores pueden navegar fácilmente para ver el código que se está revisando en nuestra herramienta de búsqueda y exploración de código, editar el código en nuestra herramienta de edición de código basada en la web o ver los resultados de las pruebas asociadas con un cambio de código.

Entre estos principios rectores, la simplicidad probablemente haya tenido el mayor impacto en la herramienta. Hubo muchas características interesantes que consideramos agregar, pero decidimos no hacer que el modelo fuera más complicado para admitir un pequeño grupo de usuarios.

La simplicidad también tiene una tensión interesante con la integración del flujo de trabajo. Consideramos, pero finalmente decidimos no crear una herramienta "Code Central" con edición, revisión y búsqueda de código en una sola herramienta. Aunque Critique tiene muchos puntos de contacto con otras herramientas, decidimos conscientemente mantener la revisión del código como el enfoque principal. Las características están vinculadas desde Critique pero implementadas en diferentes subsistemas.

Flujo de revisión de código

Las revisiones de código se pueden ejecutar en muchas etapas del desarrollo de software, como se ilustra en [Figura 19-1](#). Las revisiones críticas suelen tener lugar antes de que se pueda confirmar un cambio en el código base, también conocido como *revisiones previas*. A pesar de [Capítulo 9](#) contiene una breve descripción del flujo de revisión de código, aquí lo ampliamos para describir aspectos clave de

Crítica que ayuda en cada etapa. Veremos cada etapa con más detalle en las siguientes secciones.

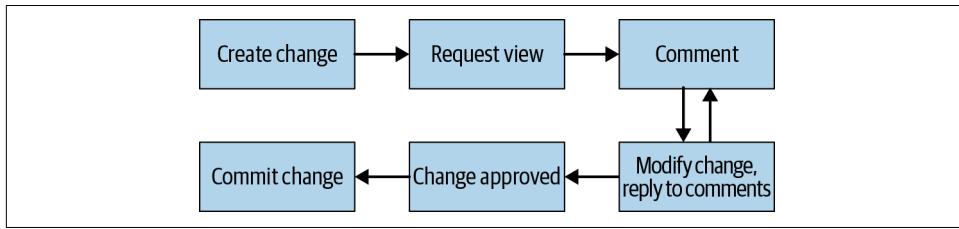


Figura 19-1. El flujo de revisión de código

Los pasos típicos de revisión son los siguientes:

- 1. Crea un cambio.** Un usuario crea un cambio en el código base en su espacio de trabajo. Esto autorluego sube un *instantánea* (mostrando un parche en un momento determinado) a Critique, que desencadena la ejecución de analizadores de código automáticos (ver capítulo 20).
- 2. Solicitar revisión.** Una vez que el autor está satisfecho con la diferencia del cambio y el resultado de los analizadores que se muestran en Critique, envía el cambio por correo a uno o más revisores.
- 3. Comentario.** revisores sobre el cambio en Critique y redacta los comentarios sobre la diferencia. Los comentarios están marcados por defecto como *irresuelto*, lo que significa que son cruciales para que el autor los aborde. Además, los revisores pueden agregar *resuelto* a los comentarios que son opcionales o informativos. Los resultados de los analizadores de código automáticos, si están presentes, también son visibles para los revisores. Una vez que un revisor ha redactado un conjunto de comentarios, debe publicarlos para que el autor los vea; esto tiene la ventaja de permitir que un revisor proporcione una idea completa sobre un cambio de forma atómica, después de haber revisado todo el cambio. Cualquier persona puede comentar sobre los cambios, proporcionando una "revisión automática" cuando lo consideren necesario.
- 4. Modificar el cambio y responder a los comentarios.** El autor modifica el cambio, carga nuevas instantáneas basadas en los comentarios y responde a los revisores. El autor aborda (al menos) todos los comentarios no resueltos, ya sea cambiando el código o simplemente respondiendo al comentario y cambiando el tipo de comentario para que sea *resuelto*. El autor y los revisores pueden observar las diferencias entre cualquier par de instantáneas para ver qué cambió. Los pasos 3 y 4 pueden repetirse varias veces.
- 5. Cambio de aprobación.** Cuando los revisores están satisfechos con el último estado del cambio, aprueban el cambio y lo marcan como "me parece bien" (LGTM). Opcionalmente pueden incluir comentarios a la dirección. Una vez que se considera que un cambio es bueno para el envío, se marca claramente en verde en la interfaz de usuario para mostrar este estado.
- 6. Confirmar un cambio.** Siempre que se apruebe el cambio (de lo que hablaremos en breve), el autor puede desencadenar el proceso de confirmación del cambio. Si automático

Los analizadores y otros ganchos de confirmación previa (llamados "envíos previos") no encuentran ningún problema, el cambio se confirma en la base de código.

Incluso después de que se inicia el proceso de revisión, todo el sistema proporciona una flexibilidad significativa para desviarse del flujo de revisión regular. Por ejemplo, los revisores pueden anular la asignación del cambio o asignarlo explícitamente a otra persona, y el autor puede posponer la revisión por completo. En casos de emergencia, el autor puede forzar su cambio y hacer que se revise después de la confirmación.

Notificaciones

A medida que un cambio avanza a través de las etapas descritas anteriormente, Critique publica notificaciones de eventos que pueden ser utilizadas por otras herramientas de apoyo. Este modelo de notificación permite que Critique se centre en ser una herramienta de revisión de código principal en lugar de una herramienta de propósito general, sin dejar de estar integrada en el flujo de trabajo del desarrollador. Las notificaciones permiten una separación de preocupaciones, de modo que Critique puede simplemente emitir eventos y otros sistemas se construyen a partir de esos eventos.

Por ejemplo, los usuarios pueden instalar una extensión de Chrome que consume estas notificaciones de eventos. Cuando un cambio requiere la atención del usuario, por ejemplo, porque es su turno de revisar el cambio o falla algún envío previo, la extensión muestra una notificación de Chrome con un botón para ir directamente al cambio o silenciar la notificación. Descubrimos que a algunos desarrolladores realmente les gusta la notificación inmediata de las actualizaciones de cambios, pero otros eligen no usar esta extensión porque consideran que es demasiado perjudicial para su flujo.

Critique también administra los correos electrónicos relacionados con un cambio; Los eventos importantes de Critique desencadenan notificaciones por correo electrónico. Además de mostrarse en la interfaz de usuario de Critique, algunos hallazgos del analizador están configurados para enviar los resultados también por correo electrónico. Critique también procesa las respuestas de correo electrónico y las traduce a comentarios, lo que ayuda a los usuarios que prefieren un flujo basado en correo electrónico. Tenga en cuenta que para muchos usuarios, los correos electrónicos no son una característica clave de la revisión de código; usan la vista del tablero de Critique (que se analiza más adelante) para administrar las reseñas.

Etapa 1: Crear un cambio

Una herramienta de revisión de código debe brindar soporte en todas las etapas del proceso de revisión y no debe ser un cuello de botella para confirmar cambios. En el paso de revisión previa, facilitar a los autores del cambio el pulido de un cambio antes de enviarlo para su revisión ayuda a reducir el tiempo que tardan los revisores en inspeccionar el cambio. Critique muestra las diferencias de cambio con perillas para ignorar los cambios de espacios en blanco y resaltar los cambios de solo movimiento. La crítica también muestra los resultados de compilaciones, pruebas y analizadores estáticos, incluidas las comprobaciones de estilo (como se explica en [Capítulo 9](#)).

Mostrarle a un autor la diferencia de un cambio le da la oportunidad de usar un sombrero diferente: el de un revisor de código. Critique le permite al autor de un cambio ver la diferencia de sus cambios como lo hará su revisor, y también ver los resultados del análisis automático. La crítica también admite la realización de modificaciones ligeras al cambio desde la herramienta de revisión y sugiere revisores apropiados. Al enviar la solicitud, el autor también puede incluir comentarios preliminares sobre el cambio, lo que brinda la oportunidad de preguntar directamente a los revisores sobre cualquier pregunta abierta. Dar a los autores la oportunidad de ver un cambio tal como lo hacen sus revisores evita malentendidos.

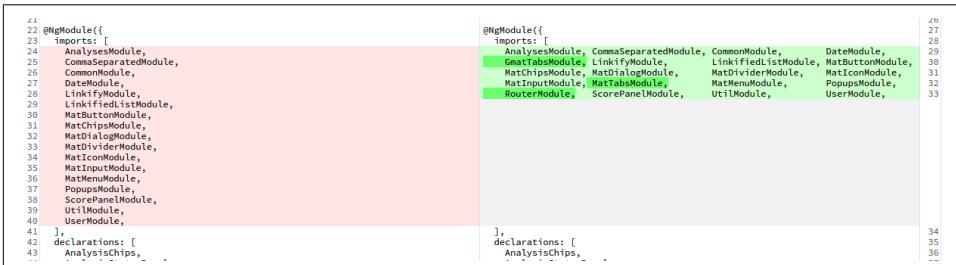
Para proporcionar más contexto a los revisores, el autor también puede vincular el cambio a un error específico. Critique utiliza un servicio de autocompletar para mostrar errores relevantes, priorizando los errores asignados al autor.

diferenciando

El núcleo del proceso de revisión de código es comprender el cambio de código en sí. Los cambios más grandes suelen ser más difíciles de entender que los más pequeños. Por lo tanto, optimizar la diferencia de un cambio es un requisito fundamental para una buena herramienta de revisión de código.

En Critique, este principio se traduce en múltiples capas (ver [Figura 19-2](#)). El componente de diferenciación, a partir de un algoritmo de subsecuencia común más largo optimizado, se mejora con lo siguiente:

- Resaltado de sintaxis
- Referencias cruzadas (impulsadas por Kythe; ver [capítulo 17](#))
- Diferenciación intralínea que muestra la diferencia en la factorización a nivel de carácter en los límites de las palabras ([Figura 19-2](#))
- Una opción para ignorar las diferencias de espacios en blanco en un grado variable
- Detección de movimientos, en la que los fragmentos de código que se mueven de un lugar a otro se marcan como movidos (en lugar de marcarlos como eliminados aquí y agregados allí, como lo haría un algoritmo de diferencias ingenuo)



```

4A
22 @NgModule({
23   imports: [
24     AnalyticsModule,
25     CommaSeparatedModule,
26     CommonModule,
27     DateModule,
28     MatDialogModule,
29     LinkifyModule,
30     LinkifiedListModule,
31     MatButtonModule,
32     MatChipsModule,
33     MatDialogModule,
34     MatDividerModule,
35     MatIconModule,
36     MatInputModule,
37     PopupsModule,
38     ScorePanelModule,
39     UtilModule,
40     UserModule,
41   ],
42   declarations: [
43     AnalysisChips,
44   ],
45 }
46 )
47 
```

The screenshot shows a GitHub-style diff view. On the left, the original code is shown with line numbers 22 through 47. On the right, the modified code is shown with line numbers 27 through 33. The changes are highlighted in green. The first change (line 22) adds `@NgModule({` and the closing brace. The second change (line 27) adds `imports: [`. The third change (line 28) adds `AnalyticsModule,`. The fourth change (line 29) adds `CommaSeparatedModule,`. The fifth change (line 30) adds `CommonModule,`. The sixth change (line 31) adds `DateModule,`. The seventh change (line 32) adds `MatButtonModule,`. The eighth change (line 33) adds `MatChipsModule,`. The ninth change (line 34) adds `MatDialogModule,`. The tenth change (line 35) adds `MatDividerModule,`. The eleventh change (line 36) adds `MatIconModule,`. The twelfth change (line 37) adds `MatInputModule,`. The thirteenth change (line 38) adds `PopupsModule,`. The fourteenth change (line 39) adds `ScorePanelModule,`. The fifteenth change (line 40) adds `UtilModule,`. The sixteenth change (line 41) adds `UserModule,`. The seventeenth change (line 42) adds `declarations: [`. The eighteenth change (line 43) adds `AnalysisChips,`.

Figura 19-2. Diferencias intralínea que muestran diferencias de nivel de caracteres

Los usuarios también pueden ver la diferencia en varios modos diferentes, como superpuestos y uno al lado del otro. Cuando desarrollamos Critique, decidimos que era importante tener diferencias en paralelo para facilitar el proceso de revisión. Las diferencias una al lado de la otra ocupan mucho espacio: para hacerlas realidad, tuvimos que simplificar la estructura de la vista de diferencias, por lo que no hay borde ni relleno, solo las diferencias y los números de línea. También tuvimos que jugar con una variedad de fuentes y tamaños hasta que obtuvimos una vista diferente que se adapta incluso al límite de línea de 100 caracteres de Java para la resolución típica de ancho de pantalla cuando se lanzó Critique (1440 píxeles).

Critique admite además una variedad de herramientas personalizadas que proporcionan diferencias de artefactos producidos por un cambio, como una diferencia de captura de pantalla de la interfaz de usuario modificada por un cambio o archivos de configuración generados por un cambio.

Para que el proceso de navegar por las diferencias sea fluido, tuvimos cuidado de no desperdiciar espacio y dedicamos un gran esfuerzo a garantizar que las diferencias se carguen rápidamente, incluso para imágenes y archivos grandes y/o cambios. También proporcionamos atajos de teclado para navegar rápidamente a través de los archivos mientras visita solo las secciones modificadas.

Cuando los usuarios profundizan hasta el nivel del archivo, Critique proporciona un widget de interfaz de usuario con una visualización compacta de la cadena de versiones instantáneas de un archivo; los usuarios pueden arrastrar y soltar para seleccionar qué versiones comparar. Este widget colapsa automáticamente instantáneas similares, enfocando la atención en instantáneas importantes. Ayuda al usuario a comprender la evolución de un archivo dentro de un cambio; por ejemplo, qué instantáneas tienen cobertura de prueba, ya se han revisado o tienen comentarios. Para abordar las preocupaciones de escala, Critique precarga todo, por lo que cargar diferentes instantáneas es muy rápido.

Resultados de análisis

Cargar una instantánea de los analizadores de código de desencadenadores de cambios ([ver capítulo 20](#)). Critique muestra los resultados del análisis en la página de cambios, resumidos por chips de estado del analizador que se muestran debajo de la descripción del cambio, como se muestra en [Figura 19-3](#), y detallado en la pestaña Análisis, como se ilustra en [Figura 19-4](#).

Los analizadores pueden marcar hallazgos específicos para resaltarlos en rojo para una mayor visibilidad. Los analizadores que todavía están en proceso se representan con fichas amarillas, y las fichas grises se muestran de lo contrario. En aras de la simplicidad, Critique no ofrece otras opciones para marcar o resaltar los hallazgos: la capacidad de acción es una opción binaria. Si un analizador produce algunos resultados ("hallazgos"), al hacer clic en el chip se abren los hallazgos. Al igual que los comentarios, los resultados se pueden mostrar dentro de la diferencia, pero con un estilo diferente para que se distingan fácilmente. A veces, los hallazgos también incluyen sugerencias de arreglos, que el autor puede previsualizar y elegir aplicar desde Crítica.

The screenshot shows the Critique web interface for a pull request titled "Change 243497582 by ilham". The status is "Pending". The summary box contains the following text:

```
Implement pizza supplier (1/6).
Add a skeleton for the pizza supplier system.
We follow the organic framework for establishing the connection between the basic ingredients to the supplier.
```

Below the summary, there are buttons for "Modify", "Revert", and "Submit".

On the right, there is a sidebar with "Score" (LGTM - Missing), "Analysis" (Actionable: Presubmit:CheckProtoSyntax), and "Done: Presubmit".

The main area shows a table of differences:

File	Comments	Inline	Modified	Delta
<input type="checkbox"/> pizza/BUILD Added		Diff	3:05 PM	7
<input type="checkbox"/> pizza/PizzaSupplierApp.java Added		Diff	3:06 PM	22
<input type="checkbox"/> pizza/pizza.proto Added		Hide	3:05 PM	28

Below the table, a code snippet from /dev/null is shown:

```
package google.pizza;

import google.Timestamp;

message Ingredient {
    required int64 id = 1;
    required string name = 2;
    required int64 unit = 3;
    optional Timestamp season = 4;
```

A tooltip for the "Presubmit:CheckProtoSyntax" action is displayed, stating: "Your change contains one or more new .proto files without a syntax statement. Each .proto file must have a syntax statement in order to submit." It includes a link to "Not useful".

Figura 19-3. Cambiar resumen y vista de diferencias

The screenshot shows the Critique interface for the same pull request. The status is "Pending". The summary box contains the same text as in Figure 19-3.

The main area shows a table of findings:

Category	Status	Snapshot	First finding snippet
Presubmit:CheckProtoSyntax	✓	2 (Latest)	Actionable: Your change contains one or more new .proto files without a syntax statement. Each .proto file must have a syntax statement in order to submit.
Presubmit	✓	2 (Latest)	Presubmits finished with status SUCCESS. Reported 1 notice(s), 0 warning(s), 1 error(s). NOTES: Presubmits were invoked with ...

Figura 19-4. Resultados de análisis

Por ejemplo, suponga que un linter encuentra una violación de estilo de espacios adicionales al final de la línea. La página de cambio mostrará un chip para ese linter. Desde el chip, el autor puede ir rápidamente a la diferencia que muestra el código infractor para comprender la infracción de estilo con dos clics. La mayoría de las infracciones de linter también incluyen sugerencias de corrección. Con un clic,

el autor puede obtener una vista previa de la sugerencia de corrección (por ejemplo, eliminar los espacios adicionales) y, con otro clic, aplicar la corrección en el cambio.

Integración estrecha de herramientas

Google tiene herramientas construidas sobre Piper, su repositorio de código fuente monolítico (ver [capítulo 16](#)), como las siguientes:

- Cider, un IDE en línea para editar el código fuente almacenado en la nube
- Code Search, una herramienta para buscar código en la base de código
- Tricorder, una herramienta para mostrar resultados de análisis estáticos (mencionado anteriormente)
- Rapid, una herramienta de lanzamiento que empaqueta e implementa binarios que contienen una serie de cambios
- Zapfhahn, una herramienta de cálculo de cobertura de prueba

Además, hay servicios que brindan contexto sobre metadatos de cambio (por ejemplo, sobre usuarios involucrados en un cambio o errores vinculados). Critique es un crisol natural para un acceso rápido con un solo clic/desplazamiento o incluso soporte de interfaz de usuario integrado para estos sistemas, aunque debemos tener cuidado de no sacrificar la simplicidad. Por ejemplo, desde una página de cambios en Crítica, el autor debe hacer clic solo una vez para comenzar a editar el cambio en Sidra. Hay soporte para navegar entre referencias cruzadas usando Kythe o ver el estado de la línea principal del código en Búsqueda de código (ver [capítulo 17](#)). Critique vincula a la herramienta de lanzamiento para que los usuarios puedan ver si un cambio enviado está en un lanzamiento específico. Para estas herramientas, Critique favorece los enlaces en lugar de la incrustación para no distraer la atención de la experiencia de revisión central. Una excepción aquí es la cobertura de la prueba: la información de si una línea de código está cubierta por una prueba se muestra con diferentes colores de fondo en la medianilla de línea en la vista de diferencias del archivo (no todos los proyectos usan esta herramienta de cobertura).

Tenga en cuenta que es posible una estrecha integración entre Critique y el espacio de trabajo de un desarrollador debido al hecho de que los espacios de trabajo se almacenan en un sistema de archivos basado en FUSE, al que se puede acceder más allá de la computadora de un desarrollador en particular. La Fuente de la Verdad está alojada en la nube y es accesible para todas estas herramientas.

Etapa 2: Solicitud de revisión

Una vez que el autor esté satisfecho con el estado del cambio, puede enviarlo para su revisión, como se muestra en [Figura 19-5](#). Esto requiere que el autor elija a los revisores. Dentro de un equipo pequeño, encontrar un revisor puede parecer simple, pero incluso allí es útil distribuir las revisiones de manera uniforme entre los miembros del equipo y considerar situaciones como quién está de vacaciones. Para abordar esto, los equipos pueden proporcionar un alias de correo electrónico para las revisiones de código entrantes. El alias es utilizado por una herramienta llamada *GwsQ* llamado así por el equipo inicial que usó esta técnica:

Google Web Server) que asigna revisores específicos en función de la configuración vinculada al alias. Por ejemplo, un autor de cambios puede asignar una revisión a algún-equipo-lista-alias, y GwsQ seleccionará a un miembro específico de algún-equipo-lista-alias para realizar la revisión.

The screenshot shows a 'Modify Changelist' dialog box. At the top, it says 'Changelist Description' and contains the text: 'Implement pizza supplier (1/6).', 'Add a skeleton for the pizza supplier system.', and 'We follow the organic framework for establishing the connection between the basic ingredients to the supplier.' Below this, there are input fields for 'Reviewers' (containing 'caitlin'), 'CC', 'Bugs', and 'Fixes'. At the bottom are 'Save' and 'Cancel' buttons, and tabs for 'Comments', 'Inline', 'Modified', and 'Delta'.

Figura 19-5. Solicitando revisores

Dado el tamaño del código base de Google y la cantidad de personas que lo modifican, puede ser difícil averiguar quién está mejor calificado para revisar un cambio fuera de su propio proyecto. Encontrar revisores es un problema a tener en cuenta a la hora de llegar a una determinada escala. La crítica debe lidiar con la escala. Critique ofrece la funcionalidad de proponer conjuntos de revisores que son suficientes para aprobar el cambio. La utilidad de selección de revisores tiene en cuenta los siguientes factores:

- Quién es el propietario del código que se está modificando (consulte la siguiente sección)
- Quién está más familiarizado con el código (es decir, quién lo cambió recientemente)
- Quién está disponible para revisión (es decir, no fuera de la oficina y preferiblemente en la misma zona horaria)
- La configuración del alias del equipo GwsQ

La asignación de un revisor a un cambio desencadena una solicitud de revisión. Esta solicitud ejecuta "envíos previos" o ganchos de confirmación previa aplicables al cambio; los equipos pueden configurar los requisitos previos relacionados con sus proyectos de muchas maneras. Los ganchos más comunes incluyen los siguientes:

- Agregar automáticamente listas de correo electrónico a los cambios para aumentar la conciencia y la transparencia
- Ejecución de conjuntos de pruebas automatizados para el proyecto
- Hacer cumplir las invariantes específicas del proyecto tanto en el código (para hacer cumplir las restricciones de estilo de código locales) como en las descripciones de cambios (para permitir la generación de notas de la versión u otras formas de seguimiento)

Dado que la ejecución de pruebas requiere muchos recursos, en Google son parte de los envíos previos (se ejecutan cuando se solicita una revisión y cuando se confirman los cambios) en lugar de para cada instantánea como las comprobaciones de Tricorder. La crítica muestra el resultado de ejecutar los ganchos de una manera similar a cómo se muestran los resultados del analizador, con una distinción adicional para resaltar el hecho de que un resultado fallido bloquea el envío del cambio para su revisión o confirmación. Critique notifica al autor por correo electrónico si los envíos previos fallan.

Etapas 3 y 4: comprender y comentar un cambio

Una vez que comienza el proceso de revisión, el autor y los revisores trabajan en conjunto para alcanzar el objetivo de realizar cambios de alta calidad.

Comentando

Hacer comentarios es la segunda acción más común que los usuarios realizan en Crítica después de ver los cambios ([Figura 19-6](#)). Comentar en Critique es gratis para todos. Cualquiera — no solo el autor del cambio y los revisores asignados— pueden comentar sobre un cambio.

Critique también ofrece la capacidad de realizar un seguimiento del progreso de la revisión a través del estado por persona. Los revisores tienen casillas de verificación para marcar archivos individuales en la última instantánea como revisados, lo que ayuda al revisor a realizar un seguimiento de lo que ya ha visto. Cuando el autor modifica un archivo, la casilla de verificación "revisado" para ese archivo se borra para todos los revisores porque se actualizó la última instantánea.

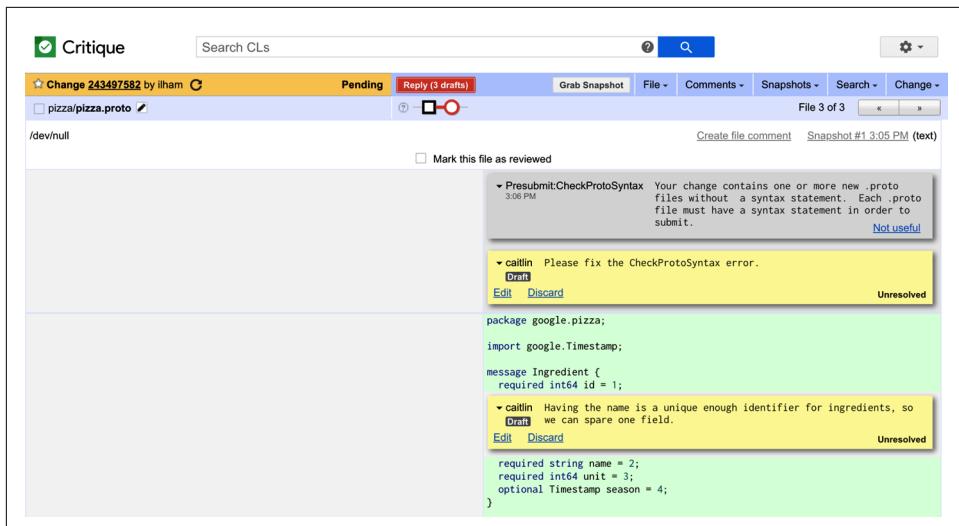


Figura 19-6. Comentando la vista de diferencias

Cuando un revisor ve un hallazgo relevante del analizador, puede hacer clic en el botón "Corrija por favor" para crear un comentario sin resolver que le pide al autor que aborde el hallazgo. Los revisores también pueden sugerir una solución a un cambio editando en línea la última versión del archivo. La crítica transforma esta sugerencia en un comentario con una solución adjunta que puede ser aplicada por el autor.

Critique no dicta qué comentarios deben crear los usuarios, pero para algunos comentarios comunes, Critique proporciona atajos rápidos. El autor del cambio puede hacer clic en el botón "Listo" en el panel de comentarios para indicar cuándo se ha abordado el comentario de un revisor, o en el botón "Reconocer" para confirmar que se ha leído el comentario, que normalmente se usa para comentarios informativos u opcionales. Ambos tienen el efecto de resolver el hilo de comentarios si no está resuelto. Estos accesos directos simplifican el flujo de trabajo y reducen el tiempo necesario para responder a los comentarios de revisión.

Como se mencionó anteriormente, los comentarios se redactan sobre la marcha, pero luego se "publican" atómicamente, como se muestra en Figura 19-7. Esto permite a los autores y revisores asegurarse de que están satisfechos con sus comentarios antes de enviarlos.

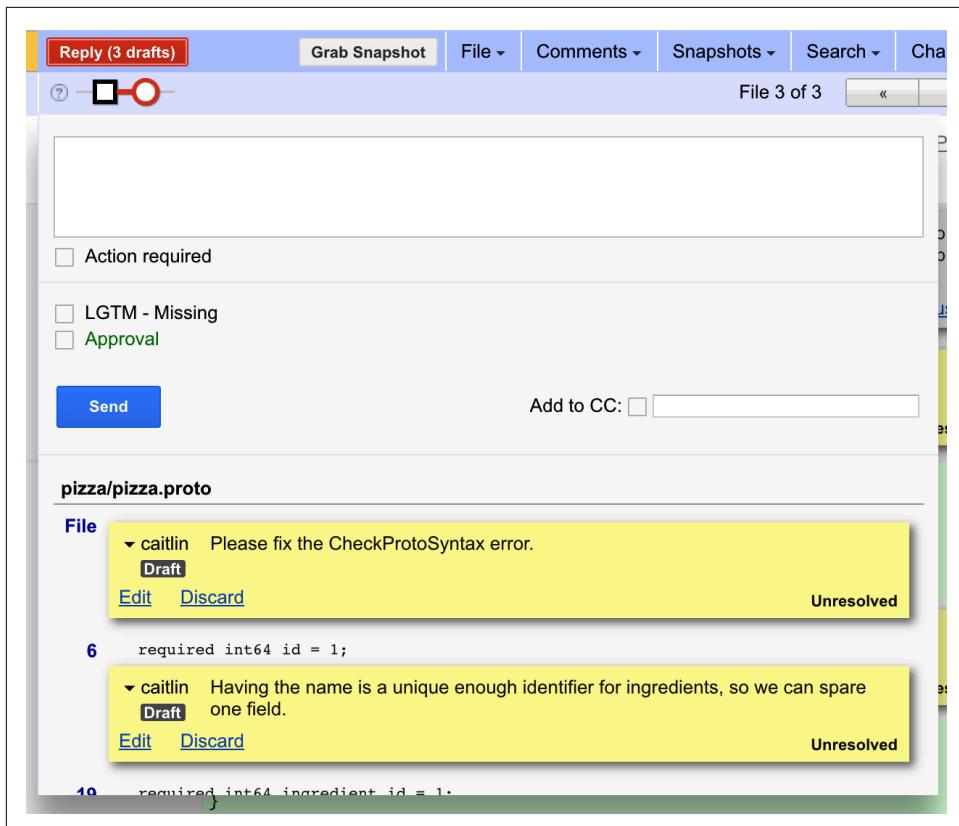


Figura 19-7. Preparación de comentarios al autor.

Comprender el estado de un cambio

La crítica proporciona una serie de mecanismos para aclarar en qué parte de la fase de comentario e iteración se encuentra actualmente un cambio. Estos incluyen una función para determinar quién debe tomar medidas a continuación y una vista de tablero del estado de revisión/autor para todos los cambios en los que está involucrado un desarrollador en particular.

Función "A quién le toca"

Un factor importante para acelerar el proceso de revisión es comprender cuándo es su turno de actuar, especialmente cuando hay varios revisores asignados a un cambio. Este podría ser el caso si el autor desea que un ingeniero de software y la persona de experiencia del usuario responsable de la función revisen su cambio, o el SRE que lleva el buscapersonas para el servicio. La crítica ayuda a definir quién se espera que observe el cambio a continuación mediante la gestión de un *conjunto de atención* por cada cambio.

El conjunto de atención comprende el conjunto de personas sobre las que actualmente está bloqueado un cambio. Cuando un revisor o autor está en el grupo de atención, se espera que responda de manera oportuna. Critique trata de ser inteligente al actualizar el conjunto de atención cuando un usuario publica sus comentarios, pero los usuarios también pueden administrar el conjunto de atención por sí mismos. Su utilidad aumenta aún más cuando hay más revisores en el cambio. El conjunto de atención aparece en Critique al mostrar los nombres de usuario relevantes en negrita.

Después de que implementamos esta función, a nuestros usuarios les resultó difícil imaginar el estado anterior. La opinión predominante es: ¿cómo nos las arreglamos sin esto? La alternativa antes de que implementáramos esta función era chatear entre los revisores y los autores para saber quién estaba lidiando con un cambio. Esta característica también enfatiza la naturaleza por turnos de la revisión de código; siempre es el turno de al menos una persona para actuar.

Panel de control y sistema de búsqueda

La página de destino de Critique es la página del panel de control del usuario, como se muestra en Figura 19-8. La página del panel está dividida en secciones personalizables por el usuario, cada una de las cuales contiene una lista de resúmenes de cambios.

Needs attention 4 Changes						
Change	Author	Status	Last Action	Reviewers	Size	Description
42972248	ilham	Pending	Apr 11 by gwsq	caitlin	XS	Implement pizza supplier (6/6).
42974683	ilham	Pending	Apr 11 by tap	caitlin	S	Implement pizza supplier (5/6).
37099895	ilham	Pending	Apr 11 by ilham	caitlin	M	Implement pizza supplier (4/6).
27761071	caitlin	Pending	Jan 8 by ilham	ilham	XS	Implement pizza maker (3/3).

Incoming reviews 6 Changes						
Change	Author	Status	Last Action	Reviewers	Size	Description
42972248	ilham	Pending	Apr 11 by gwsq	caitlin	XS	Implement pizza supplier (6/6).
42974683	ilham	Pending	Apr 11 by tap	caitlin	S	Implement pizza supplier (5/6).
37099895	ilham	Pending	Apr 11 by ilham	caitlin	M	Implement pizza supplier (4/6).
42161351	ilham	LGTM	Apr 9 by caitlin	caitlin	XS	Implement pizza supplier (3/6).
40374250	ilham	Unresolved	Apr 4 by caitlin	caitlin	XS	Implement pizza supplier (2/6).
36387832	ilham	Unresolved	Mar 5 by caitlin	caitlin	L	Implement pizza supplier (1/6).

Outgoing reviews 3 Changes						
Change	Author	Status	Last Action	Reviewers	Size	Description
27761071	caitlin	Pending	Jan 8 by caitlin	ilham	XS	Implement pizza maker (3/3).
15068925	caitlin	Pending	Jan 6 by caitlin	ilham	S	Implement pizza maker (2/3).
15416497	caitlin	Pending	Jan 2 by caitlin	ilham	M	Implement pizza maker (1/3).

Figura 19-8. vista de tablero

La página del tablero funciona con un sistema de búsqueda llamado *Búsqueda de lista de cambios*. La búsqueda de lista de cambios indexa el estado más reciente de todos los cambios disponibles (tanto antes como después del envío) en todos los usuarios de Google y les permite buscar cambios relevantes de forma regular.

Consultas basadas en expresiones. Cada sección del panel se define mediante una consulta a Changelist Search. Hemos dedicado tiempo a asegurarnos de que Changelist Search sea lo suficientemente rápido para un uso interactivo; todo se indexa rápidamente para que los autores y los revisores no se ralenticen, a pesar de que tenemos una gran cantidad de cambios simultáneos que ocurren simultáneamente en Google.

Para optimizar la experiencia del usuario (UX), la configuración predeterminada del tablero de Critique es que la primera sección muestre los cambios que requieren la atención del usuario, aunque esto es personalizable. También hay una barra de búsqueda para realizar consultas personalizadas sobre todos los cambios y examinar los resultados. Como revisor, en su mayoría solo necesita el conjunto de atención. Como autor, en su mayoría solo necesita echar un vistazo a lo que aún está esperando revisión para ver si necesita hacer ping a algún cambio. Si bien nos hemos alejado de la personalización en algunas otras partes de la interfaz de usuario de Critique, descubrimos que a los usuarios les gusta configurar sus tableros de manera diferente sin restar valor a la experiencia fundamental, similar a la forma en que todos organizan sus correos electrónicos de manera diferente.¹

Etapa 5: Aprobaciones de cambios (puntuación de un cambio)

Mostrar si un revisor piensa que un cambio es bueno se reduce a proporcionar inquietudes y sugerencias a través de comentarios. También debe haber algún mecanismo para proporcionar un "OK" de alto nivel en un cambio. En Google, la puntuación de un cambio se divide en tres partes:

- LGTM ("me parece bien")
- Aprobación
- El número de comentarios sin resolver

Un sello LGTM de un revisor significa que "He revisado este cambio, creo que cumple con nuestros estándares y creo que está bien confirmarlo después de abordar los comentarios no resueltos". Un sello de aprobación de un revisor significa que "como guardián, permito que este cambio se confirme en el código base". Un revisor puede marcar los comentarios como no resueltos, lo que significa que el autor deberá actuar en consecuencia. Cuando el cambio tiene al menos una LGTM, suficientes aprobaciones y ningún comentario sin resolver, el autor puede confirmar el cambio. Tenga en cuenta que cada cambio requiere un LGTM independientemente del estado de aprobación, lo que garantiza que al menos dos pares de ojos vieron el cambio. Esta simple regla de puntuación le permite a Critique informar al autor cuando un cambio está listo para confirmarse (se muestra de manera prominente como un encabezado de página verde).

¹ Los revisores "globales" centralizados para cambios a gran escala (LSC) son particularmente propensos a personalizar este tablero para evitar que se inunde durante un LSC ([ver capítulo 22](#)).

Tomamos una decisión consciente en el proceso de construcción de Critique para simplificar este esquema de calificación. Inicialmente, Critique tenía una calificación de "Necesita más trabajo" y también una "LGTM++". El modelo al que nos hemos movido es hacer que la LGTM/Aprobación sea siempre positiva. Si un cambio definitivamente necesita una segunda revisión, los revisores principales pueden agregar comentarios pero sin LGTM/Approval. Después de que un cambio pase a un estado mayormente bueno, los revisores generalmente confiarán en los autores para que se encarguen de las ediciones pequeñas; las herramientas no requieren LGTM repetidos, independientemente del tamaño del cambio.

Este esquema de calificación también ha tenido una influencia positiva en la cultura de revisión de códigos. Los revisores no pueden simplemente rechazar un cambio sin comentarios útiles; todos los comentarios negativos de los revisores deben estar vinculados a algo específico que se corregirá (por ejemplo, un comentario no resuelto). La frase "comentario no resuelto" también se eligió para sonar relativamente agradable.

La crítica incluye un panel de puntuación, junto a las fichas de análisis, con la siguiente información:

- Quién ha realizado el cambio por LGTM
- Qué aprobaciones aún se requieren y por qué
- ¿Cuántos comentarios sin resolver siguen abiertos?

Presentar la información de puntuación de esta manera ayuda al autor a comprender rápidamente lo que aún debe hacer para comprometer el cambio.

LGTM y Homologación *son durorequisitos* y sólo pueden ser otorgados por revisores. Los revisores también pueden revocar su LGTM y Aprobación en cualquier momento antes de que se confirme el cambio. Los comentarios no resueltos *son suaverequisitos*; el autor puede marcar un comentario como "resuelto" a medida que responde. Esta distinción promueve y se basa en la confianza y la comunicación entre el autor y los revisores. Por ejemplo, un revisor puede LGTM el cambio acompañado de comentarios no resueltos sin luego verificar con precisión si los comentarios están verdaderamente abordados, lo que resalta la confianza que el revisor deposita en el autor. Esta confianza es particularmente importante para ahorrar tiempo cuando hay una diferencia significativa en las zonas horarias entre el autor y el revisor. Demostrar confianza también es una buena manera de generar confianza y fortalecer los equipos.

Etapa 6: cometer un cambio

Por último, pero no menos importante, Critique tiene un botón para confirmar el cambio después de la revisión para evitar el cambio de contexto a una interfaz de línea de comandos.

Después de la confirmación: historial de seguimiento

Además del uso principal de Critique como herramienta para revisar los cambios en el código fuente antes de que se confirmen en el repositorio, Critique también se usa como herramienta para la arqueología de cambios. Para la mayoría de los archivos, los desarrolladores pueden ver una lista del historial anterior de cambios que modificaron un archivo en particular en el sistema de búsqueda de código (ver [capítulo 17](#)), o navegue directamente a un cambio. Cualquiera en Google puede navegar por el historial de un cambio a archivos generalmente visibles, incluidos los comentarios y la evolución del cambio. Esto permite auditorías futuras y se utiliza para comprender más detalles sobre por qué se realizaron cambios o cómo se introdujeron errores. Los desarrolladores también pueden usar esta característica para aprender cómo se diseñaron los cambios, y los datos de revisión de código en conjunto se usan para producir capacitaciones.

La crítica también admite la capacidad de comentar después de confirmar un cambio; por ejemplo, cuando se descubre un problema más tarde o un contexto adicional puede ser útil para alguien que investiga el cambio en otro momento. Critique también admite la capacidad de revertir cambios y ver si un cambio en particular ya se ha revertido.

Estudio de caso: Gerrit

Aunque Critique es la herramienta de revisión más utilizada en Google, no es la única. Critique no está disponible externamente debido a sus estrechas interdependencias con nuestro gran repositorio monolítico y otras herramientas internas. Debido a esto, los equipos de Google que trabajan en proyectos de código abierto (incluidos Chrome y Android) o proyectos internos que no pueden o no quieren hospedarse en el repositorio monolítico utilizan una herramienta de revisión de código diferente: Gerrit.

Gerrit es una herramienta de revisión de código abierto e independiente que está estrechamente integrada con el sistema de control de versiones de Git. Como tal, ofrece una interfaz de usuario web para muchas funciones de Git, incluida la exploración de código, la fusión de ramas, las confirmaciones de selección selectiva y, por supuesto, la revisión de código. Además, Gerrit tiene un modelo de permisos detallado que podemos usar para restringir el acceso a repositorios y sucursales.

Tanto Critique como Gerrit tienen el mismo modelo para las revisiones de código en el sentido de que cada confirmación se revisa por separado. Gerrit admite la acumulación de confirmaciones y su carga para revisión individual. También permite que la cadena se comprometa atómicamente después de revisarla.

Al ser de código abierto, Gerrit admite más variantes y una gama más amplia de casos de uso; El rico sistema de complementos de Gerrit permite una estrecha integración en entornos personalizados. Para respaldar estos casos de uso, Gerrit también admite un sistema de puntuación más sofisticado. Un revisor puede vetar un cambio colocando una puntuación de -2, y el sistema de puntuación es altamente configurable.



Puede obtener más información sobre Gerrit y verlo en acción en <https://www.gerritcodereview.com>.

Conclusión

Hay una serie de compensaciones implícitas cuando se utiliza una herramienta de revisión de código. Critique incorpora una serie de funciones y se integra con otras herramientas para que el proceso de revisión sea más sencillo para sus usuarios. El tiempo dedicado a las revisiones de código es tiempo que no se dedica a la codificación, por lo que cualquier optimización del proceso de revisión puede ser una ganancia de productividad para la empresa. Tener solo dos personas en la mayoría de los casos (autor y revisor) acordando el cambio antes de que pueda comprometerse mantiene alta la velocidad. Google valora mucho los aspectos educativos de la revisión de código, aunque son más difíciles de cuantificar.

Para minimizar el tiempo que se tarda en revisar un cambio, el proceso de revisión del código debe fluir sin problemas, informando a los usuarios de manera sucinta sobre los cambios que necesitan su atención e identificando posibles problemas antes de que lleguen los revisores humanos (los analizadores y la integración continua detectan los problemas). . Cuando es posible, los resultados del análisis rápido se presentan antes de que finalicen los análisis de mayor duración.

Hay varias formas en que la Crítica necesita apoyar las cuestiones de escala. La herramienta Critique debe escalar a la gran cantidad de solicitudes de revisión producidas sin sufrir una degradación en el rendimiento. Debido a que Critique se encuentra en la ruta crítica para lograr que los cambios se confirmen, debe cargarse de manera eficiente y ser útil para situaciones especiales, como cambios inusualmente grandes.² La interfaz debe admitir la gestión de las actividades de los usuarios (como encontrar cambios relevantes) en la gran base de código y ayudar a los revisores y autores a navegar por la base de código. Por ejemplo, Critique ayuda a encontrar revisores apropiados para un cambio sin tener que averiguar el panorama de propiedad/mantenedor (una característica que es particularmente importante para cambios a gran escala, como las migraciones de API que pueden afectar a muchos archivos).

Critique favorece un proceso obstinado y una interfaz simple para mejorar el flujo de trabajo de revisión general. Sin embargo, Critique permite cierta personalización: los analizadores personalizados y los envíos previos brindan un contexto específico sobre los cambios, y se pueden aplicar algunas políticas específicas del equipo (como exigir LGTM de varios revisores).

² Aunque la mayoría de los cambios son pequeños (menos de 100 líneas), Critique se usa a veces para revisar grandes refactorizaciones, cambios que pueden tocar cientos o miles de archivos, especialmente para LSC que deben ejecutarse atómicamente ([ver capítulo 22](#)).

La confianza y la comunicación son fundamentales para el proceso de revisión del código. Una herramienta puede mejorar la experiencia, pero no puede reemplazarla. La estrecha integración con otras herramientas también ha sido un factor clave en el éxito de Critique.

TL; DR

- La confianza y la comunicación son fundamentales para el proceso de revisión del código. Una herramienta puede mejorar la experiencia, pero no puede reemplazarla.
- La estrecha integración con otras herramientas es clave para una excelente experiencia de revisión de código.
- Las pequeñas optimizaciones del flujo de trabajo, como la adición de un "conjunto de atención" explícito, pueden aumentar la claridad y reducir sustancialmente la fricción.

Análisis estático

*Escrito por Caitlin Sadowski
Editado por Lisa Carey*

El análisis estático se refiere a los programas que analizan el código fuente para encontrar posibles problemas, como errores, antipatrones y otros problemas que se pueden diagnosticar.*sin ejecutar el programa.* La parte "estática" se refiere específicamente a analizar el código fuente en lugar de un programa en ejecución (denominado análisis "dinámico"). El análisis estático puede encontrar errores en los programas antes de que se registren como código de producción. Por ejemplo, el análisis estático puede identificar expresiones constantes que se desbordan, pruebas que nunca se ejecutan o cadenas de formato no válidas en declaraciones de registro que fallarían cuando se ejecutaran.*.Sin embargo, el análisis estático es útil para algo más que encontrar errores. A través del análisis estático en Google, codificamos las mejores prácticas, ayudamos a mantener el código actualizado para las versiones modernas de la API y evitamos o reducimos la deuda técnica.* Ejemplos de estos análisis incluyen verificar que se mantengan las convenciones de nomenclatura, marcar el uso de API obsoletas o señalar expresiones más simples pero equivalentes que hacen que el código sea más fácil de leer. El análisis estático también es una herramienta integral en el proceso de desaprobación de la API, donde puede evitar retrocesos durante la migración del código base a una nueva API (ver [capítulo 22](#)). También hemos encontrado evidencia de que las verificaciones de análisis estático pueden educar a los desarrolladores y, de hecho, evitar que los antipatrones ingresen al código base.

2

1 Ver <http://errorprone.info/bugpatterns>.

2 Caitlin Sadowski et al. [Tricorder: construcción de un ecosistema de análisis de programas](#), Conferencia Internacional sobre Software Engineering (ICSE), mayo de 2015.

En este capítulo, veremos qué hace que el análisis estático sea efectivo, algunas de las lecciones que hemos aprendido en Google sobre cómo hacer que el análisis estático funcione y cómo implementamos estas mejores prácticas en nuestras herramientas y procesos de análisis estático.³

Características del análisis estático eficaz

Aunque ha habido décadas de investigación de análisis estático centrada en el desarrollo de nuevas técnicas de análisis y análisis específicos, un enfoque en enfoques para mejorar *escalabilidad y usabilidad* de herramientas de análisis estático ha sido un desarrollo relativamente reciente.

Escalabilidad

Debido a que el software moderno se ha vuelto más grande, las herramientas de análisis deben abordar explícitamente el escalado para producir resultados de manera oportuna, sin ralentizar el proceso de desarrollo de software. Las herramientas de análisis estático de Google deben adaptarse al tamaño del código base de miles de millones de líneas de Google. Para hacer esto, las herramientas de análisis son fragmentables e incrementales. En lugar de analizar proyectos grandes completos, enfocamos los análisis en los archivos afectados por un cambio de código pendiente y, por lo general, mostramos los resultados del análisis solo para archivos o líneas editados. El escalado también tiene beneficios: debido a que nuestra base de código es tan grande, hay muchas cosas al alcance de la mano en términos de errores para encontrar. Además de asegurarnos de que las herramientas de análisis puedan ejecutarse en una gran base de código, también debemos aumentar la cantidad y la variedad de análisis disponibles. Se solicitan contribuciones de análisis de toda la empresa.*procesos* escalable. Para hacer esto, la infraestructura de análisis estático de Google evita los resultados de análisis de cuello de botella mostrándolos directamente a los ingenieros relevantes.

usabilidad

Cuando se piensa en la usabilidad del análisis, es importante tener en cuenta la relación costo-beneficio para los usuarios de herramientas de análisis estático. Este "costo" podría ser en términos de tiempo del desarrollador o calidad del código. Arreglar una advertencia de análisis estático podría introducir un error. Para el código que no se modifica con frecuencia, ¿por qué "arreglar" el código que funciona bien en producción? Por ejemplo, arreglar una advertencia de código inactivo agregando una llamada al código previamente inactivo podría resultar en la ejecución repentina de un código no probado (posiblemente con errores). No hay un beneficio claro y un costo potencialmente alto. Por esta razón, generalmente nos enfocamos en las advertencias recién introducidas; Por lo general, solo vale la pena resaltar (y solucionar) los problemas existentes en el código que funciona de otro modo si son particularmente importantes (problemas de seguridad, correcciones de errores importantes, etc.). Centrarse en las advertencias recién introducidas (o advertencias en

³ Una buena referencia académica para la teoría del análisis estático es: Flemming Nielson et al. *Principios del análisis de programas* (Alemania: Springer, 2004).

líneas modificadas) también significa que los desarrolladores que ven las advertencias tienen el contexto más relevante.

Además, ¡el tiempo del desarrollador es valioso! El tiempo dedicado a clasificar los informes de análisis o solucionar los problemas destacados se compara con el beneficio proporcionado por un análisis en particular. Si el autor del análisis puede ahorrar tiempo (p. ej., proporcionando una solución que se puede aplicar automáticamente al código en cuestión), el costo de la compensación se reduce. Cualquier cosa que pueda arreglarse automáticamente debería arreglarse automáticamente. También tratamos de mostrar a los desarrolladores informes sobre problemas que realmente tienen un impacto negativo en la calidad del código para que no pierdan el tiempo buscando resultados irrelevantes.

Para reducir aún más el costo de revisar los resultados del análisis estático, nos enfocamos en una integración fluida del flujo de trabajo del desarrollador. Otro punto fuerte de homogeneizar todo en un solo flujo de trabajo es que un equipo de herramientas dedicado puede actualizar las herramientas junto con el flujo de trabajo y el código, lo que permite que las herramientas de análisis evolucionen con el código fuente en tandem.

Creemos que estas elecciones y compensaciones que hemos hecho al hacer que los análisis estáticos sean escalables y utilizables surgen orgánicamente de nuestro enfoque en tres principios básicos, que formulamos como lecciones en la siguiente sección.

Lecciones clave para hacer que el análisis estático funcione

Hay tres lecciones clave que hemos aprendido en Google sobre lo que hace que las herramientas de análisis estático funcionen bien. Echemos un vistazo a ellos en las siguientes subsecciones.

Centrarse en la felicidad del desarrollador

Mencionamos algunas de las formas en que tratamos de ahorrar tiempo a los desarrolladores y reducir el costo de interactuar con las herramientas de análisis estático antes mencionadas; también realizamos un seguimiento del rendimiento de las herramientas de análisis. Si no mides esto, no puedes solucionar los problemas. Solo implementamos herramientas de análisis con bajas tasas de falsos positivos (más sobre eso en un minuto). Nosotros también *Solicitar activamente y actuar sobre la retroalimentación* de desarrolladores que consumen resultados de análisis estáticos, en tiempo real. Alimentar este ciclo de retroalimentación entre los usuarios de herramientas de análisis estático y los desarrolladores de herramientas crea un círculo virtuoso que ha generado confianza en los usuarios y mejorado nuestras herramientas. La confianza del usuario es extremadamente importante para el éxito de las herramientas de análisis estático.

Para el análisis estático, un "falso negativo" es cuando una pieza de código contiene un problema que la herramienta de análisis fue diseñada para encontrar, pero la herramienta no lo detecta. Un "falso positivo" ocurre cuando una herramienta marca incorrectamente el código como si tuviera el problema. Investigación sobre herramientas de análisis estático tradicionalmente enfocadas a la reducción de falsos negativos; en la práctica, bajo falso positivo

las tasas a menudo son críticas para que los desarrolladores realmente quieran usar una herramienta: ¿quién quiere navegar a través de cientos de informes falsos en busca de algunos verdaderos?⁴

Es más, *percepciones* un aspecto clave de la tasa de falsos positivos. Si una herramienta de análisis estático produce advertencias que son técnicamente correctas pero que los usuarios malinterpretan como falsos positivos (por ejemplo, debido a mensajes confusos), los usuarios reaccionarán como si esas advertencias fueran en realidad falsos positivos. Del mismo modo, las advertencias que son técnicamente correctas pero sin importancia en el gran esquema de las cosas provocan la misma reacción. Llamamos a la tasa de falsos positivos percibida por el usuario la tasa de "falsos positivos efectivos". Un problema es un "falso positivo efectivo" si los desarrolladores no tomaron ninguna medida positiva después de ver el problema. Esto significa que si un análisis informa incorrectamente un problema, pero el desarrollador felizmente hace la solución de todos modos para mejorar la legibilidad o el mantenimiento del código, eso no es un falso positivo efectivo. Por ejemplo, contiene método en una tabla hash (que es equivalente a contieneValor) cuando en realidad querían llamar contiene clave— incluso si el desarrollador tuvo la intención correcta de verificar el valor, llamando contieneValoren cambio es más claro. De manera similar, si un análisis informa una falla real, pero el desarrollador no entendió la falla y, por lo tanto, no tomó ninguna medida, eso es un falso positivo efectivo.

Haga del análisis estático una parte del flujo de trabajo del desarrollador principal

En Google, integramos el análisis estático en el flujo de trabajo central a través de la integración con herramientas de revisión de código. Básicamente, todo el código confirmado en Google se revisa antes de confirmarlo; Debido a que los desarrolladores ya tienen una mentalidad de cambio cuando envían el código para su revisión, las mejoras sugeridas por las herramientas de análisis estático se pueden realizar sin demasiada interrupción. Hay otros beneficios para la integración de revisión de código. Los desarrolladores suelen cambiar de contexto después de enviar el código para su revisión y se bloquean en los revisores. Hay tiempo para que se ejecuten los análisis, incluso si tardan varios minutos en hacerlo. También existe la presión de los pares por parte de los revisores para abordar las advertencias de análisis estático. Además, el análisis estático puede ahorrar tiempo al revisor al resaltar problemas comunes automáticamente; Las herramientas de análisis estático ayudan a escalar el proceso de revisión de código (y los revisores).⁵

Empoderar a los usuarios para que contribuyan

Hay muchos expertos en dominios en Google cuyo conocimiento podría mejorar el código producido. El análisis estático es una oportunidad para aprovechar la experiencia y aplicarla a escala haciendo que los expertos del dominio escriban nuevas herramientas de análisis o comprobaciones individuales dentro de una herramienta.

4 Tenga en cuenta que hay algunos análisis específicos para los que los revisores podrían estar dispuestos a tolerar una tasa de falsos positivos: un ejemplo son los análisis de seguridad que identifican problemas críticos.

5 Consulte más adelante en este capítulo para obtener más información sobre puntos de integración adicionales al editar y examinar código.

Por ejemplo, los expertos que conocen el contexto de un tipo particular de archivo de configuración pueden escribir un analizador que verifique las propiedades de esos archivos. Además de los expertos del dominio, los análisis son aportados por los desarrolladores que descubren un error y les gustaría evitar que el mismo tipo de error vuelva a aparecer en cualquier otro lugar de la base de código. Nos enfocamos en construir un ecosistema de análisis estático que sea fácil de conectar en lugar de integrar un pequeño conjunto de herramientas existentes. Nos hemos centrado en desarrollar API simples que puedan usar los ingenieros de todo Google, no solo los expertos en análisis o lenguaje para crear análisis; por ejemplo, Refaster⁶ permite escribir un analizador especificando fragmentos de código previo y posterior que demuestran qué transformaciones espera ese analizador.

Tricorder: la plataforma de análisis estático de Google

Tricorder, nuestra plataforma de análisis estático, es una parte central del análisis estático en Google.⁷ Tricorder surgió de varios intentos fallidos de integrar el análisis estático con el flujo de trabajo del desarrollador en Google;⁸ la diferencia clave entre Tricorder y los intentos anteriores fue nuestro incansable enfoque en hacer que Tricorder solo entregue resultados valiosos a sus usuarios. Tricorder está integrado con la principal herramienta de revisión de código de Google, Critique. Las advertencias de Tricorder aparecen en el visor de diferencias de Critique como cuadros de comentarios grises, como se muestra en Figura 20-1.

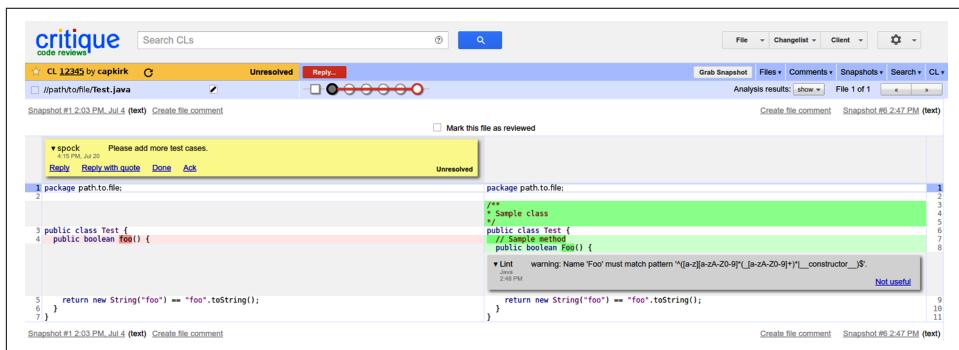


Figura 20-1. Visualización de diferencias de Critique, que muestra una advertencia de análisis estático de Tricorder en gris

⁶ Louis Wasserman, "Refactorizaciones escalables basadas en ejemplos con Refaster." Taller de Herramientas de Refactorización, 2013.

⁷ Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg y Collin Winter, **Tricorder: Construyendo un Ecosistema de análisis de programas**, Congreso Internacional de Ingeniería de Software (ICSE), mayo de 2015.

⁸ Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon y Ciera Jaspan, "Lecciones de Creación de herramientas de análisis estático en Google", *Comunicaciones de la ACM*, 61 núm. 4 (abril de 2018): 58–66, <https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext>.

Para escalar, Tricorder utiliza una arquitectura de microservicios. El sistema Tricorder envía solicitudes de análisis a servidores de análisis junto con metadatos sobre un cambio de código. Estos servidores pueden usar esos metadatos para leer las versiones de los archivos de código fuente en el cambio a través de un sistema de archivos basado en FUSE y pueden acceder a las entradas y salidas de compilación en caché. Luego, el servidor de análisis comienza a ejecutar cada analizador individual y escribe la salida en una capa de almacenamiento; los resultados más recientes para cada categoría se muestran en Crítica. Debido a que los análisis a veces tardan unos minutos en ejecutarse, los servidores de análisis también publican actualizaciones de estado para que los autores de cambios y los revisores sepan que los analizadores se están ejecutando y publican un estado completo cuando han terminado. Tricorder analiza más de 50 000 cambios de revisión de código por día y, a menudo, ejecuta varios análisis por segundo.

Los desarrolladores de todo Google escriben análisis Tricorder (llamados "analizadores") o contribuyen con "comprobaciones" individuales a los análisis existentes. Hay cuatro criterios para los nuevos controles Tricorder:

Sea comprensible

Sea fácil para cualquier ingeniero entender la salida.

Sea procesable y fácil de arreglar

La solución puede requerir más tiempo, pensamiento o esfuerzo que una verificación del compilador, y el resultado debe incluir una guía sobre cómo se podría solucionar el problema.

Produce menos del 10% de falsos positivos efectivos

Los desarrolladores deben sentir que la verificación señala un problema real **menos el 90% del tiempo**.

Tienen el potencial de tener un impacto significativo en la calidad del código.

Es posible que los problemas no afecten la corrección, pero los desarrolladores deben tomarlos en serio y elegir deliberadamente solucionarlos.

Los analizadores Tricorder informan resultados en más de 30 idiomas y admiten una variedad de tipos de análisis. Tricorder incluye más de 100 analizadores, y la mayoría son contribuciones externas al equipo de Tricorder. Siete de estos analizadores son en sí mismos sistemas complementarios que tienen cientos de comprobaciones adicionales, nuevamente aportadas por desarrolladores de Google. La tasa general efectiva de falsos positivos está justo por debajo del 5%.

Herramientas integradas

Hay muchos tipos diferentes de herramientas de análisis estático integradas con Tricorder.

Propenso a errores y **limpio y ordenado** extender el compilador para identificar antipatrones AST para Java y C++, respectivamente. Estos antipatrones podrían representar errores reales. Por ejemplo, considere el siguiente fragmento de código que codifica un campo F de tipo largo:

```
resultado = 31 * resultado + (int) (f ^ (f >>> 32));
```

Ahora considere el caso en el cual el tipo deFesEn t.El código aún se compilará, pero el desplazamiento a la derecha por 32 no es operativo, por lo queFes XORed consigo mismo y ya no afecta el valor producido.

Solucionamos 31 ocurrencias de este error en la base de código de Google mientras habilitamos la verificación como un error del compilador en Error Prone. Existen**muchos más ejemplos de este tipo**. Los antipatrones AST también pueden generar mejoras en la legibilidad del código, como eliminar una llamada redundante a .conseguir(en un puntero inteligente).

Otros analizadores muestran las relaciones entre archivos dispares en un corpus. El analizador de artefactos eliminados advierte si se elimina un archivo de origen al que se hace referencia en otros lugares que no son de código en la base de código (como dentro de la documentación registrada). IfThis-ThenThat permite a los desarrolladores especificar que partes de dos archivos diferentes deben cambiarse en tandem (y advierte si no es así). El analizador Finch de Chrome se ejecuta en archivos de configuración para experimentos A/B en Chrome, lo que destaca problemas comunes, como no tener las aprobaciones correctas para iniciar un experimento o interferencias con otros experimentos que se están ejecutando actualmente y que afectan a la misma población. El analizador Finch realiza llamadas de procedimiento remoto (RPC) a otros servicios para proporcionar esta información.

Además del propio código fuente, algunos analizadores se ejecutan en otros artefactos producidos por ese código fuente; muchos proyectos han habilitado un verificador de tamaño binario que advierte cuando los cambios afectan significativamente un tamaño binario.

Casi todos los analizadores son intraprocedimiento, lo que significa que los resultados del análisis se basan en el código dentro de un procedimiento (función). Las técnicas de análisis interprocesal composicional o incremental son técnicamente factibles, pero requerirían una inversión adicional en infraestructura (p. ej., analizar y almacenar resúmenes de métodos a medida que se ejecutan los analizadores).

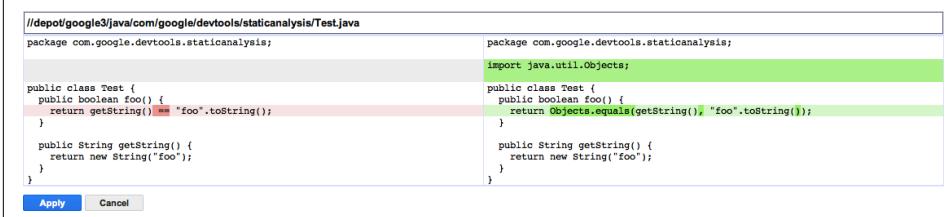
Canales de retroalimentación integrados

Como se mencionó anteriormente, establecer un circuito de retroalimentación entre los consumidores de análisis y los escritores de análisis es fundamental para rastrear y mantener la felicidad de los desarrolladores. Con Tricorder, mostramos la opción de hacer clic en el botón "No útil" en un resultado de análisis; este clic ofrece la opción de presentar un error *directamente contra el escritor del analizadores* sobre por qué el resultado no es útil con información sobre el resultado del análisis rellenada previamente. Los revisores de código también pueden pedir a los autores de cambios que aborden los resultados del análisis haciendo clic en el botón "Corrija". El equipo de Tricorder realiza un seguimiento de los analizadores con altas tasas de clics "No útiles", particularmente en relación con la frecuencia con la que los revisores solicitan corregir los resultados del análisis, y deshabilitará los analizadores si no funcionan para abordar los problemas y mejorar los "no útiles". Índice. Establecer y ajustar este circuito de retroalimentación requirió mucho trabajo, pero ha dado sus frutos muchas veces en mejores resultados de análisis y una mejor experiencia de usuario (UX): antes de que estableciésemos canales de retroalimentación claros, muchos desarrolladores simplemente ignoraban los resultados del análisis que no entendían..

Y a veces la solución es bastante simple, como actualizar el texto del mensaje que genera un analizador. Por ejemplo, una vez implementamos una verificación Propenso a errores que marcaba cuando se pasaban demasiados argumentos a unimprimir-como función en guayaba que aceptaba solo %s (y ningún otroimprimirespecificadores). El equipo propenso a errores recibió informes de errores semanales "No útiles" que afirmaban que el análisis era incorrecto porque la cantidad de especificadores de formato coincidía con la cantidad de argumentos, todo debido a que los usuarios intentaban pasar especificadores que no eran %s. Después de que el equipo cambiara el texto de diagnóstico para indicar directamente que la función acepta solo el %smarcador de posición, se detuvo la afluencia de informes de errores. Mejorar el mensaje producido por un análisis brinda una explicación de lo que está mal, por qué y cómo corregirlo exactamente en el punto donde es más relevante y puede marcar la diferencia para que los desarrolladores aprendan algo cuando lean el mensaje.

Correcciones sugeridas

Tricorder también comprueba, cuando es posible, *proporcionar correcciones*, como se muestra en Figura 20-2.



The screenshot shows a Java code editor window titled 'Test.java'. The code contains a class 'Test' with two methods: 'foo()' and 'getString()'. The 'foo()' method has a warning under 'return getString()' with the message 'Unnecessary toString()'. The 'getString()' method has a suggestion under 'return new String("foo")' with the message 'Use Objects.equals() instead'. At the bottom of the editor are 'Apply' and 'Cancel' buttons.

```
//depot/google3/java/com/google/devtools/staticanalysis/Test.java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }

    public String getString() {
        return new String("foo");
    }
}
```

Figura 20-2. Vista de un ejemplo de solución de análisis estático en Critique

Las correcciones automatizadas sirven como una fuente de documentación adicional cuando el mensaje no es claro y, como se mencionó anteriormente, reducen el costo de abordar los problemas de análisis estático. Las correcciones se pueden aplicar directamente desde Critique, o sobre un cambio de código completo a través de una herramienta de línea de comandos. Aunque no todos los analizadores proporcionan soluciones, muchos lo hacen. Tomamos el enfoque que *estilos* los problemas en particular deben solucionarse automáticamente; por ejemplo, por formateadores que reformatean automáticamente los archivos de código fuente. Google tiene guías de estilo para cada idioma que especifican problemas de formato; señalar errores de formato no es un buen uso del tiempo de un revisor humano. Los revisores hacen clic en "Corregir" miles de veces al día y los autores aplican las correcciones automáticas aproximadamente 3000 veces al día. Y los analizadores Tricorder recibieron clics de "No útil" 250 veces al día.

Personalización por proyecto

Una vez que construimos una base de confianza del usuario al mostrar solo resultados de análisis de alta confianza, agregamos la capacidad de ejecutar analizadores "opcionales" adicionales para proyectos específicos además de los predeterminados. El *Mejores prácticas de proto* analizador es un ejemplo de un analizador opcional. Este analizador destaca datos potencialmente peligrosos

cambios de formato **abúferes de protocolo**—Formato de serialización de datos independiente del idioma de Google. Estos cambios solo se interrumpen cuando los datos serializados se almacenan en algún lugar (por ejemplo, en los registros del servidor); Los búferes de protocolo para proyectos que no tienen datos serializados almacenados no necesitan habilitar la verificación. También hemos agregado la capacidad de personalizar los analizadores existentes, aunque por lo general esta personalización es limitada y muchas comprobaciones se aplican de manera uniforme por defecto en toda la base de código.

Algunos analizadores incluso comenzaron como opcionales, mejoraron en función de los comentarios de los usuarios, crearon una gran base de usuarios y luego pasaron al estado predeterminado tan pronto como pudimos capitalizar la confianza de los usuarios que habíamos creado. Por ejemplo, tenemos un analizador que sugiere mejoras en la legibilidad del código Java que normalmente no cambian el comportamiento del código. A los usuarios de Tricorder inicialmente les preocupaba que este análisis fuera demasiado "ruidoso", pero eventualmente querían tener más resultados de análisis disponibles.

La idea clave para que esta personalización tuviera éxito fue centrarse en *personalización a nivel de proyecto, no personalización a nivel de usuario*. La personalización a nivel de proyecto garantiza que todos los miembros del equipo tengan una vista coherente de los resultados del análisis de su proyecto y evita situaciones en las que un desarrollador intenta solucionar un problema mientras otro lo presenta.

Al principio del desarrollo de Tricorder, un conjunto de verificadores de estilo relativamente sencillo ("linters") mostraban los resultados en Critique, y Critique proporcionaba configuraciones de usuario para elegir el nivel de confianza de los resultados para mostrar y suprimir resultados de análisis específicos. Eliminamos toda esta personalización del usuario de Critique e inmediatamente comenzamos a recibir quejas de los usuarios sobre resultados de análisis molestos. En lugar de volver a habilitar la personalización, les preguntamos a los usuarios por qué estaban molestos y encontramos todo tipo de errores y falsos positivos con los linters. Por ejemplo, el linter de C++ también se ejecutó en archivos Objective-C pero produjo resultados incorrectos e inútiles. Arreglamos la infraestructura de pelusa para que esto ya no suceda. El HTML linter tenía una tasa extremadamente alta de falsos positivos con muy poca señal útil y, por lo general, los desarrolladores que escribían HTML lo ocultaban de la vista. Debido a que el linter rara vez fue útil, simplemente lo deshabilitamos. En resumen, la personalización del usuario resultó en errores ocultos y supresión de comentarios.

Prequerimientos

Además de la revisión del código, también hay otros puntos de integración del flujo de trabajo para el análisis estático en Google. Debido a que los desarrolladores pueden optar por ignorar las advertencias de análisis estático que se muestran en la revisión de código, Google también tiene la capacidad de agregar un análisis que bloquea la confirmación de un cambio de código pendiente, lo que llamamos *unpresuponer cheque*. Las comprobaciones previas al envío incluyen comprobaciones integradas muy sencillas y personalizables sobre el contenido o los metadatos de un cambio, como garantizar que el mensaje de confirmación no diga "NO ENVIAR" o que los archivos de prueba siempre se incluyan con los archivos de código correspondientes. Los equipos también pueden especificar un conjunto de pruebas que deben pasar o verificar que no haya Tricorder

problemas para una categoría en particular. Los envíos previos también verifican que el código esté bien formateado. Las comprobaciones previas al envío normalmente se ejecutan cuando un desarrollador envía un cambio por correo para su revisión y nuevamente durante el proceso de confirmación, pero se pueden activar ad hoc entre esos puntos. Ver [capítulo 23](#) para obtener más detalles sobre los presuntos envíos en Google.

Algunos equipos han escrito sus propios presupuestos personalizados. Estos son controles adicionales además del conjunto básico de envío previo que agregan la capacidad de hacer cumplir estándares de mejores prácticas más altos que la empresa en su conjunto y agregan análisis específicos del proyecto. Esto permite que los nuevos proyectos tengan pautas de mejores prácticas más estrictas que los proyectos con grandes cantidades de código heredado (por ejemplo). Los envíos previos específicos del equipo pueden hacer que el proceso de cambio a gran escala (LSC) ([consulte capítulo 22](#)) más difíciles, por lo que algunos se omiten por cambios con "CLEANUP=" en la descripción del cambio.

Integración del compilador

Si bien el bloqueo de confirmaciones con análisis estático es excelente, es aún mejor notificar a los desarrolladores sobre los problemas incluso antes en el flujo de trabajo. Cuando es posible, tratamos de introducir análisis estáticos en el compilador. Romper la compilación es una advertencia que no es posible ignorar, pero es inviable en muchos casos. Sin embargo, algunos análisis son altamente mecánicos y no tienen falsos positivos efectivos. Un ejemplo es [Comprobaciones de "ERROR" propensas a errores](#). Todas estas comprobaciones están habilitadas en el compilador de Java de Google, lo que evita que las instancias del error vuelvan a introducirse en nuestra base de código. Las comprobaciones del compilador deben ser rápidas para que no ralenticen la compilación. Además, aplicamos estos tres criterios (existen criterios similares para el compilador de C++):

- Procesable y fácil de corregir (siempre que sea posible, el error debe incluir una solución sugerida que se pueda aplicar mecánicamente)
- No producir falsos positivos efectivos (el análisis nunca debe detener la compilación del código correcto)
- Informar problemas que afecten solo a la corrección en lugar del estilo o las mejores prácticas

Para habilitar una nueva verificación, primero debemos limpiar todas las instancias de ese problema en el código base para que no rompamos la compilación de los proyectos existentes solo porque el compilador ha evolucionado. Esto también implica que el valor de implementar una nueva verificación basada en el compilador debe ser lo suficientemente alto como para garantizar la reparación de todas las instancias existentes. Google tiene una infraestructura para ejecutar varios compiladores (como clang y javac) en todo el código base en paralelo a través de un clúster, como una operación de MapReduce. Cuando los compiladores se ejecutan de esta manera MapReduce, la ejecución de las comprobaciones del análisis estático debe producir correcciones para automatizar la limpieza. Despues de preparar y probar un cambio de código pendiente que aplica las correcciones en todo el código base, confirmamos ese cambio y eliminamos todas las instancias existentes del problema.

construir. Las roturas de compilación se detectan después de la confirmación mediante nuestro sistema de integración continua (CI), o antes de la confirmación mediante comprobaciones previas al envío (consulte la discusión anterior).

Nuestro objetivo también es nunca emitir advertencias del compilador. Hemos encontrado repetidamente que los desarrolladores ignoran las advertencias del compilador. O habilitamos una verificación del compilador como un error (y rompemos la compilación) o no la mostramos en la salida del compilador. Debido a que se usan los mismos indicadores del compilador en todo el código base, esta decisión se toma globalmente. Las comprobaciones que no se pueden realizar para romper la compilación se suprimen o se muestran en la revisión de código (por ejemplo, a través de Tricorder). Aunque no todos los idiomas de Google tienen esta política, los que se usan con más frecuencia sí la tienen. Los compiladores de Java y C++ se configuraron para evitar mostrar advertencias del compilador. El compilador Go lleva esto al extremo; algunas cosas que otros idiomas considerarían advertencias (como variables no utilizadas o importaciones de paquetes) son errores en Go.

Análisis durante la edición y navegación de código

Otro punto de integración potencial para el análisis estático es un entorno de desarrollo integrado (IDE). Sin embargo, los análisis IDE requieren tiempos de análisis rápidos (normalmente menos de 1 segundo e idealmente menos de 100 ms), por lo que algunas herramientas no son adecuadas para integrar aquí. Además, existe el problema de asegurarse de que el mismo análisis se ejecute de forma idéntica en varios IDE. También notamos que los IDE pueden subir y bajar en popularidad (no exigimos un solo IDE); por lo tanto, la integración de IDE tiende a ser más complicada que conectarse al proceso de revisión. La revisión de código también tiene beneficios específicos para mostrar los resultados del análisis. Los análisis pueden tener en cuenta todo el contexto del cambio; algunos análisis pueden ser inexactos en código parcial (como un análisis de código muerto cuando se implementa una función antes de agregar sitios de llamada). Mostrar los resultados del análisis en la revisión del código también significa que los autores del código también tienen que convencer a los revisores si quieren ignorar los resultados del análisis. Dicho esto, la integración de IDE para análisis adecuados es otro excelente lugar para mostrar resultados de análisis estáticos.

Aunque nos enfocamos principalmente en mostrar las advertencias de análisis estático recién introducidas, o advertencias en el código editado, para algunos análisis, los desarrolladores en realidad desean tener la capacidad de ver los resultados del análisis en todo el código base durante la exploración del código. Un ejemplo de esto son algunos análisis de seguridad. Los equipos de seguridad específicos de Google quieren ver una vista holística de todas las instancias de un problema. A los desarrolladores también les gusta ver los resultados del análisis sobre el código base cuando planifican una limpieza. En otras palabras, hay momentos en los que mostrar resultados cuando la exploración de código es la elección correcta.

Conclusión

El análisis estático puede ser una gran herramienta para mejorar una base de código, encontrar errores temprano y permitir procesos más costosos (como revisión y prueba humana) para enfocarse en problemas que no son verificables mecánicamente. Al mejorar la escalabilidad y la facilidad de uso de nuestra infraestructura de análisis estático, hemos hecho del análisis estático un componente efectivo del desarrollo de software en Google.

TL; DR

- *Centrarse en la felicidad del desarrollador.* Hemos invertido un esfuerzo considerable en la creación de canales de retroalimentación entre los usuarios de análisis y los escritores de análisis en nuestras herramientas, y ajustamos agresivamente los análisis para reducir la cantidad de falsos positivos.
- *Haga que el análisis estático forme parte del flujo de trabajo principal del desarrollador.* El principal punto de integración para el análisis estático en Google es a través de la revisión de código, donde las herramientas de análisis brindan correcciones e involucran a los revisores. Sin embargo, también integramos análisis en puntos adicionales (a través de comprobaciones del compilador, confirmaciones de código de activación, en IDE y al navegar por el código).
- *Empoderar a los usuarios para que contribuyan.* Podemos escalar el trabajo que hacemos construyendo y manteniendo herramientas y plataformas de análisis aprovechando la experiencia de los expertos en el dominio. Los desarrolladores agregan continuamente nuevos análisis y comprobaciones que les facilitan la vida y mejoran nuestra base de código.

Gestión de dependencias

*Escrito por Titus Winters
Editado por Lisa Carey*

La gestión de dependencias, la gestión de redes de bibliotecas, paquetes y dependencias que no controlamos, es uno de los problemas menos entendidos y más desafiantes en la ingeniería de software. La administración de dependencias se enfoca en preguntas como: ¿cómo actualizamos entre versiones de dependencias externas? ¿Cómo describimos las versiones, para el caso? ¿Qué tipos de cambios se permiten o se esperan en nuestras dependencias? ¿Cómo decidimos cuándo es prudente depender del código producido por otras organizaciones?

A modo de comparación, el tema más estrechamente relacionado aquí es el control de fuente. Ambas áreas describen cómo trabajamos con el código fuente. El control de fuente cubre la parte más fácil: ¿dónde verificamos las cosas? ¿Cómo metemos las cosas en la compilación? Una vez que aceptamos el valor del desarrollo basado en troncales, la mayoría de las preguntas diarias de control de fuente para una organización son bastante mundanas: "Tengo algo nuevo, ¿a qué directorio lo agrego?"

La gestión de dependencias añade complejidad adicional tanto en tiempo como en escala. En un problema de control de fuente basado en troncales, es bastante claro cuando realiza un cambio que necesita para ejecutar las pruebas y no romper el código existente. Eso se basa en la idea de que está trabajando en una base de código compartida, tiene visibilidad de cómo se usan las cosas y puede desencadenar la compilación y ejecutar las pruebas. La gestión de dependencias se centra en los problemas que surgen cuando se realizan cambios fuera de su organización, sin acceso o visibilidad total. Debido a que sus dependencias ascendentes no pueden coordinarse con su código privado, es más probable que rompan su compilación y provoquen que sus pruebas fallen. ¿Cómo manejamos eso? ¿No deberíamos tomar dependencias externas? ¿Deberíamos pedir una mayor consistencia entre versiones de dependencias externas? ¿Cuándo actualizamos a una nueva versión?

La escala hace que todas estas preguntas sean más complejas, al darse cuenta de que en realidad no estamos hablando de importaciones de dependencias individuales y, en el caso general, dependemos de una red completa de dependencias externas. Cuando comenzamos a trabajar con una red, es fácil construir escenarios en los que el uso de dos dependencias por parte de su organización se vuelve insatisfactorio en algún momento. Generalmente, esto sucede porque una dependencia deja de funcionar sin algún requisito,¹ mientras que la otra es incompatible con el mismo requisito. Las soluciones simples sobre cómo administrar una única dependencia externa generalmente no tienen en cuenta las realidades de administrar una red grande. Pasaremos gran parte de este capítulo discutiendo varias formas de estos problemas de requisitos conflictivos.

El control de código fuente y la gestión de dependencias son cuestiones relacionadas separadas por la pregunta: "¿Nuestra organización controla el desarrollo/actualización/gestión de este subproyecto?" Por ejemplo, si cada equipo de su empresa tiene repositorios, objetivos y prácticas de desarrollo independientes, la interacción y la gestión del código producido por esos equipos tendrá más que ver con la gestión de dependencias que con el control de código fuente. Por otro lado, una gran organización con un único repositorio (*virtual?*) (monorepo) puede escalar significativamente más con políticas de control de código fuente: este es el enfoque de Google. Los proyectos de código abierto separados ciertamente cuentan como organizaciones separadas: las interdependencias entre proyectos desconocidos y proyectos que no necesariamente colaboran son un problema de gestión de dependencia. Quizás nuestro consejo más fuerte sobre este tema es este: *En igualdad de condiciones, prefiera los problemas de control de fuentes a los problemas de gestión de dependencias.* Si tiene la opción de redefinir la "organización" de manera más amplia (toda su empresa en lugar de solo un equipo), a menudo es una buena compensación. Los problemas de control de fuente son mucho más fáciles de pensar y mucho más baratos de manejar que los de gestión de dependencias.

A medida que el modelo de software de código abierto (OSS) continúa creciendo y expandiéndose a nuevos dominios, y el gráfico de dependencia para muchos proyectos populares continúa expandiéndose con el tiempo, la gestión de dependencias se está convirtiendo quizás en el problema más importante en la política de ingeniería de software. Ya no somos islas desconectadas construidas en una o dos capas fuera de una API. El software moderno se basa en imponentes pilares de dependencias; pero el hecho de que podamos construir esos pilares no significa que todavía hayamos descubierto cómo mantenerlos en pie y estables a lo largo del tiempo.

En este capítulo, veremos los desafíos particulares de la administración de dependencias, exploraremos soluciones (comunes y novedosas) y sus limitaciones, y veremos las realidades de trabajar con dependencias, incluida la forma en que hemos manejado las cosas en Google. Es importante comenzar todo esto con una admisión: hemos invertido mucho *pensamiento* en este problema y tengo una amplia experiencia con problemas de refactorización y mantenimiento

1 Puede ser cualquier versión de idioma, versión de una biblioteca de nivel inferior, versión de hardware, sistema operativo, indicador del compilador, versión del compilador, etc.

que muestran las deficiencias prácticas de los enfoques existentes. No tenemos evidencia de primera mano de soluciones que funcionen bien en organizaciones a escala. Hasta cierto punto, este capítulo es un resumen de lo que sabemos que no funciona (o al menos podría no funcionar a mayor escala) y donde creemos que existe el potencial para obtener mejores resultados. Definitivamente no podemos pretender tener todas las respuestas aquí; si pudieramos, no estaríamos llamando a este uno de los problemas más importantes en la ingeniería de software.

¿Por qué es tan difícil la gestión de dependencias?

Incluso definir el problema de la gestión de la dependencia presenta algunos desafíos inusuales. Muchas soluciones a medias en este espacio se centran en una formulación de problema demasiado limitada: "¿Cómo importamos un paquete del que pueda depender nuestro código desarrollado localmente?" Esta es una formulación necesaria pero no suficiente. El truco no es solo encontrar una manera de administrar una dependencia, el truco es cómo administrar *una red* de dependencias y sus cambios en el tiempo. Algun subconjunto de esta red es directamente necesario para su código de origen, parte de él solo es extraído por dependencias transitivas. Durante un período lo suficientemente largo, todos los nodos en esa red de dependencia tendrán nuevas versiones y algunas de esas actualizaciones serán importantes.² ¿Cómo gestionamos la cascada resultante de actualizaciones para el resto de la red de dependencia? O, específicamente, ¿cómo facilitamos la búsqueda de versiones mutuamente compatibles de todas nuestras dependencias dado que no controlamos esas dependencias? ¿Cómo analizamos nuestra red de dependencia? ¿Cómo gestionamos esa red, especialmente frente a un gráfico de dependencias en constante crecimiento?

Requisitos en conflicto y dependencias de diamantes

El problema central en la gestión de la dependencia destaca la importancia de pensar en términos de redes de dependencia, no de dependencias individuales. Gran parte de la dificultad surge de un problema: ¿qué sucede cuando dos nodos en la red de dependencia tienen requisitos en conflicto y su organización depende de ambos? Esto puede surgir por muchas razones, que van desde consideraciones de plataforma (sistema operativo [SO], versión de idioma, versión del compilador, etc.) hasta el problema mucho más mundano de la incompatibilidad de versión. El ejemplo canónico de incompatibilidad de versiones como un requisito de versión insatisfactorio es el *dependencia de diamantes* problema. Aunque generalmente no incluimos cosas como "¿qué versión del compilador" está usando en un gráfico de dependencia, la mayoría de estos problemas de requisitos en conflicto son isomorfos para "agregar un nodo (oculto) al gráfico de dependencia que representa esto?" requisito." Como tal, discutiremos principalmente los requisitos en conflicto en términos de dependencias de diamantes, pero tenga en cuenta que liberaría en realidad podría ser

² Por ejemplo, errores de seguridad, obsolescencias, estar en el conjunto de dependencias de una dependencia de nivel superior que tiene un error de seguridad, y así sucesivamente.

absolutamente cualquier pieza de software involucrada en la construcción de dos o más nodos en su red de dependencia.

El problema de la dependencia del diamante y otras formas de requisitos en conflicto requieren al menos tres capas de dependencia, como se demuestra en [Figura 21-1](#).

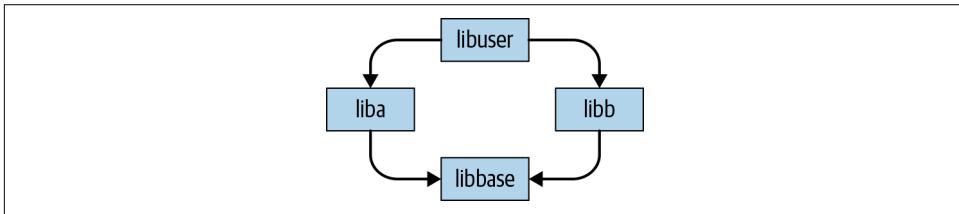


Figura 21-1. El problema de la dependencia del diamante

En este modelo simplificado, la librería es usada por ambos lib y libre, y lib y libre ambos son utilizados por un componente de nivel superior liberador. Si la librería alguna vez introduce un cambio incompatible, existe la posibilidad de que lib y libre, como productos de organizaciones separadas, no actualice simultáneamente. Si lib depende de la nueva librería y libre depende de la versión anterior, no hay una forma general del liberador (también conocido como su código) para poner todo junto. Este diamante puede formarse a cualquier escala: en toda la red de sus dependencias, si alguna vez hay un nodo de bajo nivel que debe estar en dos versiones incompatibles al mismo tiempo (en virtud de que hay dos caminos desde algún nivel superior nodo a esas dos versiones), habrá un problema.

Diferentes lenguajes de programación toleran el problema de la dependencia del diamante en diferentes grados. Para algunos lenguajes, es posible incrustar múltiples versiones (aisladas) de una dependencia dentro de una compilación: una llamada a la librería del lib podría llamar a una versión diferente de la misma API como una llamada a la librería del lib. Por ejemplo, Java proporciona mecanismos bastante bien establecidos para cambiar el nombre de los símbolos proporcionados por dicha dependencia.³ Mientras tanto, C++ tiene una tolerancia casi nula para las dependencias de diamantes en una compilación normal, y es muy probable que desencadenen errores arbitrarios y un comportamiento indefinido (UB) como resultado de una clara violación de las normas de C++. Una regla de definición. En el mejor de los casos, puede usar una idea similar al sombreado de Java para ocultar algunos símbolos en una biblioteca de vínculos dinámicos (DLL) o en los casos en los que está creando y vinculando por separado. Sin embargo, en todos los lenguajes de programación que conocemos, estas soluciones alternativas son, en el mejor de los casos, soluciones parciales: se puede hacer que funcionen varias versiones incrustadas ajustando los nombres de funciones, pero si hay tipos que se pasan entre dependencias, todas las apuestas están canceladas. Por ejemplo, simplemente no hay forma de que un mapa definido en la librería v1 para pasar a través de algunas bibliotecas a una API proporcionada por la librería v2 de forma semánticamente coherente. Trucos específicos del idioma para ocultar o renombrar entidades por separado

³ Esto se llama sombreado o versionado.

Las bibliotecas compiladas con frecuencia pueden proporcionar algo de protección para los problemas de dependencia de diamantes, pero no son una solución en el caso general.

Si encuentra un problema de requisitos conflictivos, la única respuesta fácil es saltar hacia adelante o hacia atrás en las versiones de esas dependencias para encontrar algo compatible. Cuando eso no es posible, debemos recurrir a parchear localmente las dependencias en cuestión, lo cual es particularmente desafiante porque la causa de la incompatibilidad tanto en el proveedor como en el consumidor probablemente no sea conocida por el ingeniero que primero descubre la incompatibilidad. Esto es inherente: liblos desarrolladores siguen trabajando de manera compatible conlibreríav1, y liblos desarrolladores ya se han actualizado a v2. Solo un desarrollador que esté incorporando ambos proyectos tiene la oportunidad de descubrir el problema, y ciertamente no está garantizado que esté lo suficientemente familiarizado conlibreríaylibapara trabajar a través de la actualización. La respuesta más fácil es degradarlibreríaylibre, aunque esa no es una opción si la actualización se forzó originalmente debido a problemas de seguridad.

Los sistemas de políticas y tecnología para la gestión de la dependencia se reducen en gran medida a la pregunta: "¿Cómo evitamos los requisitos conflictivos y al mismo tiempo permitimos el cambio entre los grupos que no coordinan?" Si tiene una solución para la forma general del problema de la dependencia del diamante que permite la realidad de los requisitos que cambian continuamente (tanto las dependencias como los requisitos de la plataforma) en todos los niveles de la red, ha descrito la parte interesante de una dependencia: solución de gestión.

Importación de dependencias

En términos de programación, claramente es mejor reutilizar parte de la infraestructura existente en lugar de construirla usted mismo. Esto es obvio y forma parte de la marcha fundamental de la tecnología: si todos los novatos tuvieran que volver a implementar su propio analizador JSON y motor de expresiones regulares, nunca llegaríamos a ningún lado. La reutilización es saludable, especialmente en comparación con el costo de volver a desarrollar software de calidad desde cero. Siempre que no descargue software troyano, si su dependencia externa cumple con los requisitos para su tarea de programación, debe usarlo.

Promesas de compatibilidad

Cuando comenzamos a considerar el tiempo, la situación gana algunas compensaciones complicadas. Solo porque puedes evitar *undesarrollo*El costo no significa que importar una dependencia sea la opción correcta. En una organización de ingeniería de software que es consciente del tiempo y el cambio, también debemos ser conscientes de sus costos de mantenimiento continuos. Incluso si importamos una dependencia sin la intención de actualizarla, las vulnerabilidades de seguridad descubiertas, las plataformas cambiantes y las redes de dependencia en evolución pueden conspirar para forzar esa actualización, independientemente de nuestra intención. Cuando llegue ese día, ¿qué tan caro va a ser? Algunas dependencias son más explícitas que otras sobre el costo de mantenimiento esperado por simplemente usar esa dependencia: cuánta compatibilidad es

¿ficticio? ¿Cuánta evolución se supone? ¿Cómo se manejan los cambios? ¿Durante cuánto tiempo se admiten las versiones?

Sugerimos que un proveedor de dependencia sea más claro acerca de las respuestas a estas preguntas. Considere el ejemplo establecido por grandes proyectos de infraestructura con millones de usuarios y sus promesas de compatibilidad.

C++

Para la biblioteca estándar de C++, el modelo es de compatibilidad con versiones anteriores casi indefinida. Se espera que los archivos binarios creados con una versión anterior de la biblioteca estándar se construyan y se vinculen con el estándar más nuevo: el estándar proporciona no solo compatibilidad API, sino compatibilidad continua con versiones anteriores para los artefactos binarios, conocidos como *Compatibilidad ABI*. La medida en que esto se ha mantenido varía de una plataforma a otra. Para los usuarios de gcc en Linux, es probable que la mayoría del código funcione bien durante un período de aproximadamente una década. El estándar no menciona explícitamente su compromiso con la compatibilidad de ABI: no hay documentos de política pública sobre ese punto. Sin embargo, la norma sí publica [Documento Permanente 8](#)(SD-8), que llama a un pequeño conjunto de tipos de cambio que la biblioteca estándar puede hacer entre versiones, definiendo implícitamente para qué tipo de cambios estar preparado. Java es similar: la fuente es compatible entre versiones de idioma, y los archivos JAR de versiones anteriores funcionarán fácilmente con versiones más nuevas.

Vamos

No todos los idiomas priorizan la misma cantidad de compatibilidad. El lenguaje de programación Go promete explícitamente compatibilidad de fuente entre la mayoría de las versiones, pero no compatibilidad binaria. No puede crear una biblioteca en Go con una versión del idioma y vincular esa biblioteca a un programa Go creado con una versión diferente del idioma.

Hacer rappel

El proyecto Abseil de Google es muy parecido a Go, con una importante advertencia sobre el tiempo. No estamos dispuestos a comprometernos con la compatibilidad.*indefinidamente*: Abseil se encuentra en la base de la mayoría de nuestros servicios internos más pesados desde el punto de vista computacional, que creemos que probablemente estarán en uso durante muchos años. Esto significa que nos reservamos el derecho de realizar cambios, especialmente en los detalles de implementación y ABI, para permitir un mejor rendimiento. Hemos experimentado demasiadas instancias de una API que resultó ser confusa y propensa a errores después del hecho; publicar fallas tan conocidas a decenas de miles de desarrolladores por un futuro indefinido se siente mal. Internamente, ya tenemos aproximadamente 250 millones de líneas de código C++ que dependen de esta biblioteca; no vamos a realizar cambios en la API a la ligera, pero debe ser posible. Con ese fin, Abseil explícitamente no promete compatibilidad con ABI, pero promete una forma ligeramente limitada de compatibilidad con API:

proporcionando una herramienta de refactorización automatizada que transformará el código de la antigua API a la nueva de forma transparente. Creemos que eso cambia significativamente el riesgo de costos inesperados a favor de los usuarios: no importa en qué versión se escribió una dependencia, un usuario de esa dependencia y Abseil deberían poder usar la versión más reciente. El costo más alto debería ser "ejecutar esta herramienta" y presumiblemente enviar el parche resultante para su revisión en la dependencia de nivel medio (libaolibre, continuando con nuestro ejemplo de antes). En la práctica, el proyecto es lo suficientemente nuevo como para que no hayamos tenido que hacer ningún cambio importante en la API. No podemos decir qué tan bien funcionará esto para el ecosistema en su conjunto, pero en teoría, parece un buen equilibrio entre la estabilidad y la facilidad de actualización.

Aumentar

En comparación, la biblioteca Boost C++ no promete **compatibilidad entre versiones**. La mayoría del código no cambia, por supuesto, pero "muchas de las bibliotecas de Boost se mantienen y mejoran activamente, por lo que la compatibilidad con versiones anteriores no siempre es posible". Se recomienda a los usuarios que actualicen solo en un período del ciclo de vida de su proyecto en el que algún cambio no cause problemas. El objetivo de Boost es fundamentalmente diferente al de la biblioteca estándar o Abseil: Boost es un campo de pruebas experimental. Una versión particular del flujo de Boost probablemente sea perfectamente estable y apropiada para su uso en muchos proyectos, pero los objetivos del proyecto de Boost no dan prioridad a la compatibilidad entre versiones; otros proyectos de larga duración pueden experimentar cierta fricción al mantenerse actualizados. Los desarrolladores de Boost son tan expertos como los desarrolladores de la biblioteca estándar⁴—Nada de esto tiene que ver con la experiencia técnica: se trata simplemente de lo que un proyecto promete o no promete y prioriza.

Mirando las bibliotecas en esta discusión, es importante reconocer que estos problemas de compatibilidad son *Ingeniería de software* problemas, no programación asuntos. Puede descargar algo como Boost sin promesa de compatibilidad e integrarlo profundamente en los sistemas más críticos y duraderos de su organización; va a *trabajamuy* bien. Todas las preocupaciones aquí son sobre cómo esas dependencias cambiarán con el tiempo, mantenerse al día con las actualizaciones y la dificultad de hacer que los desarrolladores se preocupen por el mantenimiento en lugar de solo hacer que las funciones funcionen. Dentro de Google, hay un flujo constante de orientación dirigida a nuestros ingenieros para ayudarlos a considerar esta diferencia entre "Lo hice funcionar" y "esto está funcionando de manera compatible". Eso no es sorprendente: es la aplicación básica de la Ley de Hyrum, después de todo.

En términos más generales: es importante darse cuenta de que la gestión de dependencias tiene una naturaleza completamente diferente en una tarea de programación frente a una tarea de ingeniería de software. Si se encuentra en un espacio problemático para el cual el mantenimiento a lo largo del tiempo es relevante, la administración de dependencias es difícil. Si solo está desarrollando una solución para hoy sin necesidad de actualizar nada, es perfectamente razonable tomar tantas soluciones disponibles

⁴ En muchos casos, existe una superposición significativa en esas poblaciones.

dependencias como desee sin pensar en cómo usarlas de manera responsable o planificar actualizaciones. Hacer que su programa funcione hoy violando todo en SD-8 y también confiando en la compatibilidad binaria de Boost y Abseil funciona bien... siempre y cuando nunca actualice la biblioteca estándar, Boost o Abseil, y tampoco lo haga nada que dependa de usted.

Consideraciones a la hora de importar

La importación de una dependencia para su uso en un proyecto de programación es casi gratuita: suponiendo que se haya tomado el tiempo para asegurarse de que hace lo que necesita y no es un agujero de seguridad en secreto, casi siempre es más barato reutilizar que volver a implementar la funcionalidad. Incluso si esa dependencia ha dado el paso de aclarar qué promesa de compatibilidad hará, siempre y cuando nunca actualicemos, cualquier cosa que construyas sobre esa instantánea de tu dependencia está bien, sin importar cuántas reglas violes en contra. - suponiendo esa API. Pero cuando pasamos de la programación a la ingeniería de software, esas dependencias se vuelven sutilmente más costosas y hay una gran cantidad de costos ocultos y preguntas que deben responderse. Con suerte, considerará estos costos antes de importar y, con suerte,

Cuando los ingenieros de Google intentan importar dependencias, los alentamos a que primero hagan esta lista (incompleta) de preguntas:

- ¿El proyecto tiene pruebas que puede ejecutar?
- ¿Pasan esas pruebas?
- ¿Quién proporciona esa dependencia? Incluso entre los proyectos OSS “sin garantía implícita”, existe una gama significativa de experiencia y conjunto de habilidades; es muy diferente depender de la compatibilidad de la biblioteca estándar de C++ o la biblioteca Guava de Java que seleccionar un proyecto aleatorio de GitHub o npm. La reputación no lo es todo, pero vale la pena investigar.
- ¿A qué tipo de compatibilidad aspira el proyecto?
- ¿El proyecto detalla qué tipo de uso se espera que sea compatible?
- ¿Qué tan popular es el proyecto?
- ¿Cuánto tiempo estaremos dependiendo de este proyecto?
- ¿Con qué frecuencia el proyecto hace cambios importantes?

Agregue a esto una breve selección de preguntas enfocadas internamente:

- ¿Qué tan complicado sería implementar esa funcionalidad dentro de Google?
- ¿Qué incentivos tendremos para mantener esta dependencia al día?
- ¿Quién realizará una actualización?

- ¿Qué tan difícil esperamos que sea realizar una actualización?

Nuestro propio Russ Cox tiene [escrito sobre esto más extensamente](#). No podemos dar una fórmula perfecta para decidir cuándo es más barato a largo plazo importar versus reimplementar; fallamos en esto nosotros mismos, la mayoría de las veces.

Cómo maneja Google la importación de dependencias

En resumen: podríamos hacerlo mejor.

La gran mayoría de las dependencias en cualquier proyecto de Google dado se desarrollan internamente. Esto significa que la gran mayoría de nuestra historia interna de administración de dependencias no es realmente administración de dependencias, es solo control de código fuente, por diseño. Como hemos mencionado, es mucho más fácil administrar y controlar las complejidades y los riesgos involucrados en agregar dependencias cuando los proveedores y los consumidores son parte de la misma organización y tienen la visibilidad adecuada y la Integración Continua (CI; ver [capítulo 23](#)) disponible. La mayoría de los problemas en la administración de dependencias dejan de serlo cuando puede ver exactamente cómo se usa su código y saber exactamente el impacto de cualquier cambio dado. El control de código fuente (cuando controlas los proyectos en cuestión) es mucho más fácil que la gestión de dependencias (cuando no lo haces).

Esa facilidad de uso comienza a fallar cuando se trata de nuestro manejo de proyectos externos. Para proyectos que estamos importando del ecosistema OSS o socios comerciales, esas dependencias se agregan en un directorio separado de nuestro monorepo, etiquetado *tercero*. Examinemos cómo se agrega un nuevo proyecto OSS a *tercero*.

Alice, ingeniera de software de Google, está trabajando en un proyecto y se da cuenta de que hay una solución de código abierto disponible. A ella realmente le gustaría tener este proyecto terminado y probado pronto, para quitarlo de en medio antes de irse de vacaciones. Entonces, la elección es volver a implementar esa funcionalidad desde cero o descargar el paquete OSS y agregarlo a *tercero*. Es muy probable que Alice decida que la solución de desarrollo más rápida tiene sentido: descarga el paquete y sigue algunos pasos en nuestro *terceropolíticas*. Esta es una lista de verificación bastante simple: asegúrese de que se construya con nuestro sistema de compilación, asegúrese de que no haya una versión existente de ese paquete y asegúrese de que al menos dos ingenieros estén registrados como PROPIETARIOS para mantener el paquete en caso de que cualquier mantenimiento es necesario. Alice hace que su compañero de equipo Bob diga: "Sí, te ayudaré". Ninguno de ellos necesita tener experiencia en el mantenimiento de un *tercero* paquete, y han evitado convenientemente la necesidad de entender nada sobre la *implementación* de este paquete. A lo sumo, han adquirido un poco de experiencia con su interfaz como parte de su uso para resolver el problema de la demostración previa.

A partir de este momento, el paquete suele estar disponible para que otros equipos de Google lo utilicen en sus propios proyectos. El acto de agregar dependencias adicionales es completamente transparente para Alice y Bob: es posible que no sepan que el paquete que descargaron y prometieron mantener se ha vuelto popular. Sutilmente, incluso si

están monitoreando el nuevo uso directo de su paquete, es posible que no noten necesariamente un crecimiento en el *transitivo* uso de su paquete. Si lo usan para una demostración, mientras que Charlie agrega una dependencia desde las entrañas de nuestra infraestructura de búsqueda, el paquete habrá pasado repentinamente de ser bastante inocuo a estar en la infraestructura crítica para los sistemas importantes de Google. Sin embargo, no tenemos ninguna señal en particular que le haya surgido a Charlie cuando está considerando agregar esta dependencia.

Ahora, es posible que este escenario esté perfectamente bien. Quizás esa dependencia esté bien escrita, no tenga errores de seguridad y no dependa de otros proyectos de OSS. Puede ser *posible* que pase bastantes años sin actualizarse. No es necesariamente *inteligente* para que eso suceda: los cambios externos podrían haberlo optimizado o agregado una nueva funcionalidad importante, o limpiado los agujeros de seguridad antes de CVEs fueron descubiertos. Cuanto más tiempo exista el paquete, es probable que se acumulen más dependencias (directas e indirectas). Cuanto más estable se mantenga el paquete, más probable será que acumulemos la Ley de Hyrum confiando en los detalles de la versión registrada.*tercer*.

Un día, a Alice y Bob se les informa que una actualización es crítica. Podría ser la revelación de una vulnerabilidad de seguridad en el paquete mismo o en un proyecto OSS que depende de él lo que fuerza una actualización. Bob ha hecho la transición a la gestión y no ha tocado la base de código en mucho tiempo. Alice se ha mudado a otro equipo desde la demostración y no ha vuelto a usar este paquete. Nadie cambió el archivo OWNERS. Miles de proyectos dependen de esto indirectamente; no podemos simplemente eliminarlo sin romper la compilación de Search y una docena de otros grandes equipos. Nadie tiene experiencia con los detalles de implementación de este paquete. Alice no está necesariamente en un equipo que tenga mucha experiencia en deshacer las sutilezas de la Ley de Hyrum que se han acumulado con el tiempo.

Todo lo cual quiere decir: Alice y los otros usuarios de este paquete se enfrentarán a una actualización costosa y difícil, y el equipo de seguridad ejercerá presión para resolver esto de inmediato. Nadie en este escenario tiene práctica en realizar la actualización, y la actualización es más difícil porque cubre muchos lanzamientos más pequeños que abarcan todo el período entre la introducción inicial del paquete *tercer* y la divulgación de seguridad.

Nuestro *tercer* las políticas no funcionan para estos escenarios lamentablemente comunes. Aproximadamente entendemos que necesitamos una barra más alta para la propiedad, necesitamos que sea más fácil (y más gratificante) actualizar regularmente y más difícil para *tercer* paquetes a ser huérfanos e importantes al mismo tiempo. La dificultad es que es difícil para los mantenedores de código base y *tercer* lleva a decir: "No, no puedes usar esta cosa que resuelve perfectamente tu problema de desarrollo porque no tenemos recursos para actualizar a todos con nuevas versiones constantemente". Proyectos que son populares y

5 vulnerabilidades y exposiciones comunes

no tienen promesa de compatibilidad (como Boost) son particularmente riesgosos: nuestros desarrolladores pueden estar muy familiarizados con el uso de esa dependencia para resolver problemas de programación fuera de Google, pero permitir que se arraigue en la estructura de nuestra base de código es un gran riesgo. Nuestro código base tiene una vida útil esperada de décadas en este punto: los proyectos upstream que no priorizan explícitamente la estabilidad son un riesgo.

Gestión de dependencias, en teoría

Habiendo visto las formas en que la administración de dependencias es difícil y cómo puede salir mal, analicemos más específicamente los problemas que estamos tratando de resolver y cómo podemos resolverlos. A lo largo de este capítulo, volvemos a la formulación, “¿Cómo gestionamos el código que viene de fuera de nuestra organización (o que no controlamos perfectamente): cómo lo actualizamos, cómo gestionamos las cosas de las que depende? con el tiempo? Debemos tener claro que cualquier buena solución aquí evita los requisitos conflictivos de cualquier forma, incluidos los conflictos de versión de dependencia de diamante, incluso en un ecosistema dinámico en el que se pueden agregar nuevas dependencias u otros requisitos (en cualquier punto de la red). También debemos ser conscientes del impacto del tiempo: todo el software tiene errores, algunos de ellos serán críticos para la seguridad, *críticos* para actualizar durante un período de tiempo suficientemente largo.

Por lo tanto, un esquema estable de administración de dependencias debe ser flexible con el tiempo y la escala: no podemos asumir la estabilidad indefinida de ningún nodo en particular en el gráfico de dependencia, ni podemos asumir que no se agregan nuevas dependencias (ya sea en el código que controlamos o en el código que controlamos). Dependemos. Si una solución para la gestión de dependencias evita problemas de requisitos conflictivos entre sus dependencias, es una buena solución. Si lo hace sin asumir la estabilidad en la versión de dependencia o distribución de dependencia, coordinación o visibilidad entre organizaciones o recursos informáticos significativos, es una gran solución.

Al proponer soluciones para la gestión de dependencias, hay cuatro opciones comunes que conocemos que exhiben al menos algunas de las propiedades apropiadas: nada cambia nunca, versiones semánticas, agrupar todo lo que necesita (coordinación no por proyecto, pero por distribución) o Live at Head.

Nada cambia (también conocido como el modelo de dependencia estática)

La forma más sencilla de garantizar dependencias estables es no cambiarlas nunca: sin cambios de API, sin cambios de comportamiento, nada. Las correcciones de errores solo se permiten si no se puede descifrar ningún código de usuario. Esto prioriza la compatibilidad y la estabilidad sobre todo lo demás. Claramente, tal esquema no es ideal debido a la suposición de estabilidad indefinida. Si, de alguna manera, llegamos a un mundo en el que los problemas de seguridad y las correcciones de errores no son un problema y las dependencias no cambian, el modelo Nothing Changes es muy atractivo: si comenzamos con restricciones satisfactorias, podremos mantener esa propiedad indefinidamente.

Aunque no es sostenible a largo plazo, en términos prácticos, aquí es donde comienza toda organización: hasta que haya demostrado que la vida útil esperada de su proyecto es lo suficientemente larga como para que el cambio sea necesario, es muy fácil vivir en un mundo en el que asumimos que nada cambia. También es importante tener en cuenta: este es probablemente el modelo correcto para la mayoría de las organizaciones nuevas. Es comparativamente raro saber que estás iniciando un proyecto que va a vivir por décadas y tendrá un *necesitar* para poder actualizar las dependencias sin problemas. Es mucho más razonable esperar que la estabilidad sea una opción real y pretender que las dependencias son perfectamente estables durante los primeros años de un proyecto.

La desventaja de este modelo es que, durante un período de tiempo lo suficientemente largo, es falso, y no hay una indicación clara de exactamente cuánto tiempo puede pretender que es legítimo. No contamos con sistemas de alerta temprana a largo plazo para errores de seguridad u otros problemas críticos que podrían obligarlo a actualizar una dependencia y, debido a las cadenas de dependencias, una sola actualización puede, en teoría, convertirse en una actualización forzada de toda su red de dependencia.

En este modelo, la selección de versiones es simple: no hay que tomar decisiones, porque no hay versiones.

Versionado semántico

El estándar de facto para "¿cómo gestionamos una red de dependencias hoy?" es el control de versiones semántico (SemVer).⁶ SemVer es la práctica casi omnipresente de representar un número de versión para alguna dependencia (especialmente bibliotecas) utilizando tres enteros separados por decimales, como 2.4.72 o 1.1.4. En la convención más común, los tres números de componentes representan versiones principales, secundarias y de parches, con la implicación de que un número principal cambiado indica un cambio en una API existente que puede interrumpir el uso existente, un número menor cambiado indica funcionalidad puramente agregada eso no debería interrumpir el uso existente, y una versión modificada del parche se reserva para los detalles de implementación que no afectan a la API y las correcciones de errores que se consideran de riesgo particularmente bajo.

Con la separación de SemVer de versiones principales/secundarias/de parches, se supone que un requisito de versión generalmente se puede expresar como "cualquier cosa más nueva que", salvo cambios incompatibles con la API (cambios de versión principales). Comúnmente, veremos "Requiere librería ≥ 1.5 ", ese requisito sería compatible con cualquier librería en 1.5, incluyendo 1.5.1, y cualquier cosa en 1.6 en adelante, pero no librería 1.4.9 (falta la API

⁶ Estrictamente hablando, SemVer se refiere solo a la práctica emergente de aplicar la semántica a mayor/menor/parche números de versión, no la aplicación de requisitos de versión compatibles entre dependencias numeradas de esa manera. Existen numerosas variaciones menores en esos requisitos entre diferentes ecosistemas, pero en general, el sistema de número de versión más restricciones descrito aquí como SemVer es representativo de la práctica en general.

introducido en 1.5) o 2.x (algunas API enlibreríase cambiaron de manera incompatible). Los cambios importantes de versión son una incompatibilidad significativa: debido a que una parte existente de la funcionalidad ha cambiado (o se ha eliminado), existen incompatibilidades potenciales para todos los dependientes. Los requisitos de versión existen (explícita o implícitamente) cada vez que una dependencia usa otra: podríamos ver "libarequierelibrería \geq 1,5" y "libre requierelibrería \geq 1.4.7."

Si formalizamos estos requisitos, podemos conceptualizar una red de dependencia como una colección de componentes de software (nodos) y los requisitos entre ellos (bordes). Las etiquetas de borde en esta red cambian en función de la versión del nodo de origen, ya sea a medida que se agregan (o eliminan) dependencias o cuando se actualiza el requisito de SemVer debido a un cambio en el nodo de origen (que requiere una función recién agregada en una dependencia). , por ejemplo. Debido a que toda esta red cambia de forma asíncrona con el tiempo, el proceso de encontrar un conjunto de dependencias compatibles entre sí que satisfagan todos los requisitos transitivos de su aplicación puede ser un desafío.⁷ Los solucionadores de satisfacción de versiones para SemVer son muy similares a los solucionadores de SAT en la investigación de lógica y algoritmos: dado un conjunto de restricciones (requisitos de versión en los bordes de dependencia), ¿podemos encontrar un conjunto de versiones para los nodos en cuestión que satisfaga todas las restricciones? La mayoría de los ecosistemas de administración de paquetes se construyen sobre este tipo de gráficos, gobernados por sus solucionadores SemVer SAT.

SemVer y sus solucionadores de SAT no prometen de ninguna manera que haya *existe* una solución a un conjunto dado de restricciones de dependencia. Constantemente se crean situaciones en las que no se pueden satisfacer las restricciones de dependencia, como ya hemos visto: si un componente de nivel inferior (librería) hace un aumento de número mayor, y algunas (pero no todas) de las bibliotecas que dependen de él (librepero nolibra) han actualizado, nos encontraremos con el problema de dependencia de diamantes.

Las soluciones de SemVer para la gestión de dependencias suelen estar basadas en SAT-solver. La selección de la versión consiste en ejecutar algún algoritmo para encontrar una asignación de versiones para dependencias en la red que satisfaga todas las restricciones de requisitos de versión. Cuando no existe una asignación de versiones tan satisfactoria, coloquialmente lo llamamos "infierno de la dependencia".

Veremos algunas de las limitaciones de SemVer con más detalle más adelante en este capítulo.

Modelos de distribución agrupados

Como industria, hemos visto la aplicación de un poderoso modelo de gestión de dependencias durante décadas: una organización reúne una colección de dependencias, encuentra un conjunto compatible entre sí y lanza la colección como una sola unidad. Esto es lo que sucede, por ejemplo, con las distribuciones de Linux: no hay garantía

7 De hecho, se ha demostrado que las restricciones de SemVer aplicadas a una red de dependencia son NP-completo.

que las diversas piezas que se incluyen en una distribución se cortan desde el mismo punto en el tiempo. De hecho, es algo más probable que las dependencias de nivel inferior sean algo más antiguas que las de nivel superior, solo para tener en cuenta el tiempo que lleva integrarlas.

Este modelo de "dibujar una caja más grande alrededor de todo y lanzar esa colección" presenta actores completamente nuevos: los distribuidores. Aunque los mantenedores de todas las dependencias individuales pueden tener poco o ningún conocimiento de las otras dependencias, estos de alto nivel *distribuidores* están involucrados en el proceso de encontrar, parchear y probar un conjunto de versiones mutuamente compatibles para incluir. Los distribuidores son los ingenieros responsables de proponer un conjunto de versiones para agrupar, probarlas para encontrar errores en ese árbol de dependencia y resolver cualquier problema.

Para un usuario externo, esto funciona muy bien, siempre que pueda confiar correctamente en solo una de estas distribuciones agrupadas. Esto es efectivamente lo mismo que cambiar una red de dependencia en una sola dependencia agregada y darle un número de versión. En lugar de decir: "Dependo de estas 72 bibliotecas en estas versiones", esto es, "Dependo de la versión N de RedHat" o "Dependo de las piezas en el gráfico NPM en el momento T".

En el enfoque de distribución por paquetes, la selección de la versión está a cargo de distribuidores dedicados.

vivir en la cabeza

El modelo que algunos de nosotros en Google⁸ han estado presionando es teóricamente sólido, pero impone cargas nuevas y costosas a los participantes en una red de dependencia. Es totalmente diferente a los modelos que existen en los ecosistemas de OSS en la actualidad, y no está claro cómo llegar de un punto a otro como industria. Dentro de los límites de una organización como Google, es costoso pero efectivo, y creemos que coloca la mayoría de los costos e incentivos en los lugares correctos. Llamamos a este modelo "Live at Head". Se puede ver como la extensión de administración de dependencias del desarrollo basado en enlaces troncales: donde el desarrollo basado en enlaces troncales habla de políticas de control de fuente, estamos extendiendo ese modelo para aplicarlo también a las dependencias ascendentes.

Live at Head presupone que podemos desanclar dependencias, eliminar SemVer y confiar en los proveedores de dependencias para probar los cambios en todo el ecosistema antes de comprometerse. Live at Head es un intento explícito de eliminar el tiempo y las opciones del tema de la gestión de la dependencia: dependa siempre de la versión actual de todo y nunca cambie nada de manera que sea difícil para sus dependientes adaptarse. Un cambio que (sin querer) altera la API o el comportamiento, en general, será detectado por CI en las dependencias posteriores y, por lo tanto, no debe confirmarse. Para casos en

⁸ Especialmente el autor y otros miembros de la comunidad de Google C++.

que tal cambio *debes* suceda (es decir, por razones de seguridad), dicha interrupción debe realizarse solo después de que se actualicen las dependencias posteriores o se proporcione una herramienta automatizada para realizar la actualización en el lugar. (Esta herramienta es esencial para los consumidores intermedios de código cerrado: el objetivo es permitir que cualquier usuario actualice el uso de una API cambiante sin un conocimiento experto del uso o la API. Esa propiedad mitiga significativamente los costos "en su mayoría transiéntes" de romper cambios). Este cambio filosófico en la responsabilidad en el ecosistema de código abierto es difícil de motivar inicialmente: poner la carga sobre un proveedor de API para probar y cambiar todos sus clientes intermedios es una revisión significativa de las responsabilidades de un proveedor de API.

Los cambios en un modelo Live at Head no se reducen a un SemVer "Creo que esto es seguro o no". En cambio, las pruebas y los sistemas de CI se utilizan para probar contra dependientes visibles para determinar experimentalmente qué tan seguro es un cambio. Por lo tanto, para un cambio que solo altera la eficiencia o los detalles de implementación, es probable que pasen todas las pruebas afectadas visibles, lo que demuestra que no hay formas obvias de que ese cambio afecte a los usuarios: es seguro comprometerse. Un cambio que modifica partes más obviamente observables de una API (sintáctica o semánticamente) a menudo generará cientos o incluso miles de fallas en las pruebas. Luego, depende del autor de ese cambio propuesto determinar si el trabajo involucrado para resolver esas fallas vale el valor resultante de confirmar el cambio. Bien hecho,

Las estructuras de incentivos y los supuestos tecnológicos aquí son materialmente diferentes a otros escenarios: asumimos que existen pruebas unitarias y CI, asumimos que los proveedores de API estarán sujetos a si las dependencias posteriores se romperán, y asumimos que los consumidores de API mantienen sus pruebas, pasando y confiando en su dependencia de manera compatible. Esto funciona significativamente mejor en un ecosistema de código abierto (en el que las correcciones se pueden distribuir con anticipación) que frente a dependencias ocultas/de código cerrado. Los proveedores de API reciben incentivos cuando realizan cambios para hacerlo de una manera que se pueda migrar sin problemas. Se incentiva a los consumidores de API a mantener sus pruebas en funcionamiento para no ser etiquetados como una prueba de baja señal y potencialmente omitida, lo que reduce la protección proporcionada por esa prueba.

En el enfoque Live at Head, la selección de la versión se maneja preguntando "¿Cuál es la versión estable más reciente de todo?" Si los proveedores han realizado cambios de manera responsable, todo funcionará en conjunto sin problemas.

Las limitaciones de SemVer

El enfoque Live at Head puede basarse en prácticas reconocidas para el control de versiones (desarrollo basado en troncales), pero en gran medida no se ha probado a escala. SemVer es el estándar de facto para la gestión de dependencias en la actualidad, pero como sugerimos, no está exento de

sus limitaciones. Debido a que es un enfoque tan popular, vale la pena analizarlo con más detalle y resaltar lo que creemos que son sus posibles escollos.

Hay mucho que desglosar en la definición de SemVer de lo que realmente significa un número de versión triple con puntos. ¿Es esto una promesa? ¿O el número de versión elegido para un lanzamiento es una estimación? Es decir, cuando los mantenedores deliberáramos cortar una nueva versión y elegir si se trata de una versión principal, secundaria o de parche, ¿qué dicen? ¿Es probable que una actualización de 1.1.4 a 1.2.0 sea segura y fácil, porque solo hubo adiciones de API y correcciones de errores? Por supuesto no. Hay un montón de cosas que los usuarios mal educados de la librería podrían haber hecho que podría causar interrupciones en la compilación o cambios de comportamiento frente a una adición de API "simple".⁹ Fundamentalmente, no puede demostrar que cualquier cosa sobre la compatibilidad cuando solo se considera la API de origen; tienes que saber *con la cual* cosas que usted está preguntando acerca de la compatibilidad.

Sin embargo, esta idea de "estimar" la compatibilidad comienza a debilitarse cuando hablamos de redes de dependencias y SAT-solvers aplicados a esas redes. El problema fundamental en esta formulación es la diferencia entre los valores de los nodos en el SAT tradicional y los valores de versión en un gráfico de dependencia de SemVer. Un nodo en un gráfico de tres SAT sea ya sea Verdadero o Falso. El mantenedor proporciona un valor de versión (1.1.14) en un gráfico de dependencia como *estimar* qué tan compatible es la nueva versión, dado el código que usó la versión anterior. Estamos construyendo toda nuestra lógica de satisfacción de la versión sobre una base inestable, tratando las estimaciones y la autocertificación como absolutas. Como veremos, incluso si eso funciona bien en casos limitados, en conjunto, no necesariamente tiene suficiente fidelidad para sustentar un ecosistema saludable.

Si reconocemos que SemVer es una estimación con pérdidas y representa solo un subconjunto del posible alcance de los cambios, podemos comenzar a verlo como un instrumento contundente. En teoría, funciona bien como la quíquidación. En la práctica, especialmente cuando construimos solucionadores de SAT encima de él, SemVer puede fallarnos (y lo hace) tanto al restringirnos en exceso como al protegernos insuficientemente.

SemVer podría sobrelimitarse

Considere lo que sucede cuando la librería reconoce que es más que un solo monolito: casi siempre hay interfaces independientes dentro de una biblioteca. Incluso si solo hay dos funciones, podemos ver situaciones en las que SemVer nos limita en exceso. Imagina eso la librería hecha se compone de sólo dos funciones, Foo y Bar. Nuestras dependencias de nivel mediolibre se usa solo Foo. Si el mantenedor de la librería hace un cambio radical en Bar, les corresponde a ellos actualizar la versión principal de la librería.

9 Por ejemplo: un polyfill mal implementado que agrega la nueva librería API antes de tiempo, causando un conflicto. O bien, el uso de API de reflexión del idioma para depender de la cantidad precisa de API proporcionadas por la librería, introduciendo bloqueos si ese número cambia. Estos no deberían suceder y ciertamente son raros incluso si suceden por accidente; el punto es que los proveedores no pueden demostrar la compatibilidad.

baseen un mundo SemVer.libaylibrese sabe que dependen delibrería1.x: los solucionadores de dependencias de SemVer no aceptarán una versión 2.x de esa dependencia. Sin embargo, en realidad, estas bibliotecas funcionarían juntas perfectamente: solo Bar cambió, y eso no se usó. La compresión inherente a "Hice un cambio radical; Debo cambiar el número de versión principal" tiene pérdida cuando no se aplica a la granularidad de una unidad API atómica individual. Aunque algunas dependencias pueden ser lo suficientemente detalladas como para que sean precisas,¹⁰esa no es la norma para un ecosistema SemVer.

Si SemVer se restringe en exceso, ya sea debido a un aumento de versión innecesariamente grave o a una aplicación insuficientemente detallada de los números de SemVer, los administradores de paquetes automatizados y los solucionadores de SAT informarán que sus dependencias no se pueden actualizar o instalar, incluso si todo funcionaría en conjunto sin problemas por ignorando los controles de SemVer. Cualquiera que alguna vez haya estado expuesto al infierno de la dependencia durante una actualización podría encontrar esto particularmente exasperante: una gran fracción de ese esfuerzo fue una completa pérdida de tiempo.

SemVer podría prometer demasiado

Por otro lado, la aplicación de SemVer hace la suposición explícita de que la estimación de compatibilidad de un proveedor de API puede ser completamente predictiva y que los cambios se dividen en tres grupos: interrupción (por modificación o eliminación), estrictamente aditivos o sin impacto de API. Si SemVer es una representación perfectamente fiel del riesgo de un cambio al clasificar los cambios sintácticos y semánticos, ¿cómo caracterizamos un cambio que agrega un retraso de un milisegundo a una API sensible al tiempo? O, más plausiblemente: ¿Cómo caracterizamos un cambio que altera el formato de nuestra salida de registro? ¿O que altera el orden en que importamos las dependencias externas? ¿O eso altera el orden en que se devuelven los resultados en una secuencia "desordenada"? ¿Es razonable suponer que esos cambios son "seguros" simplemente porque no son parte de la sintaxis o el contrato de la API en cuestión? ¿Qué pasa si la documentación dice "Esto puede cambiar en el futuro"? ¿O la API se llamó "ForInternalUseByLibBaseOnlyDoNotTouchThisIRReallyMeanIt?"¹¹

La idea de que las versiones de parches de SemVer, que en teoría solo cambian los detalles de implementación, son cambios "seguros" contradice absolutamente la experiencia de Google con la Ley de Hyrum: "Con una cantidad suficiente de usuarios, cada comportamiento observable de su sistema dependerá por alguien." Cambiar el orden en que se importan las dependencias, o cambiar el orden de salida para un productor "desordenado", a escala, invariablemente romperá las suposiciones en las que algún consumidor confiaba (quizás incorrectamente). El mismo término "cambio radical" es engañoso: hay cambios que son teóricamente disruptivos pero seguros en la práctica (eliminación de una API no utilizada). también hay

¹⁰ El ecosistema Node tiene ejemplos notables de dependencias que proporcionan exactamente una API.

¹¹ Vale la pena señalar: en nuestra experiencia, nombrar de esta manera no resuelve completamente el problema de los usuarios que buscan acceder a API privadas. Prefiera idiomas que tengan un buen control sobre el acceso público/privado a las API de todas las formas.

cambios que son teóricamente seguros pero que rompen el código del cliente en la práctica (cualquiera de nuestros ejemplos anteriores de la Ley de Hyrum). Podemos ver esto en cualquier SemVer/sistema de gestión de dependencias para el cual el sistema de requisitos de número de versión permite restricciones en el número de parche: si puede decir libbase> 1.1.14en vez delibra requiere libbase 1.1, eso es claramente una admisión de que hay diferencias observables en las versiones del parche.

Un cambio aislado no es romper o no romper—esa declaración puede evaluarse solo en el contexto de cómo se está utilizando. No hay una verdad absoluta en la noción de “Este es un cambio radical”; se puede ver que un cambio se está rompiendo solo para un conjunto (conocido o desconocido) de usuarios y casos de uso existentes. La realidad de cómo evaluamos un cambio se basa inherentemente en información que no está presente en la formulación de gestión de dependencias de SemVer: ¿cómo consumen esta dependencia los usuarios intermedios?

Debido a esto, un solucionador de restricciones de SemVer podría informar que sus dependencias funcionan juntas cuando no lo hacen, ya sea porque se aplicó un golpe incorrectamente o porque algo en su red de dependencia tenía una dependencia de la Ley de Hyrum en algo que no se consideraba parte de la superficie API observable. En estos casos, es posible que tenga errores de compilación o errores de tiempo de ejecución, sin límite superior teórico en su gravedad.

Motivaciones

Existe un argumento adicional de que SemVer no siempre incentiva la creación de código estable. Para quien mantiene una dependencia arbitraria, existe un incentivo sistémico variable para no realizar cambios importantes y mejorar las versiones principales. Algunos proyectos se preocupan mucho por la compatibilidad y harán todo lo posible para evitar un aumento de la versión principal. Otros son más agresivos, incluso actualizan intencionalmente las versiones principales en un horario fijo. El problema es que la mayoría de los usuarios de cualquier dependencia determinada son usuarios indirectos: no tendrían motivos significativos para estar al tanto de un cambio próximo. Incluso la mayoría de los usuarios directos no se suscriben a listas de correo u otras通知 de lanzamiento.

Todo lo cual se combina para sugerir que no importa cuántos usuarios se vean afectados por la adopción de un cambio incompatible en una API popular, los mantenedores corren con una pequeña fracción del costo del aumento de versión resultante. Para los mantenedores que también son usuarios, también puede haber un incentivo *hacer rompiendo*: siempre es más fácil diseñar una mejor interfaz en ausencia de restricciones heredadas. Esto es parte de por qué creemos que los proyectos deben publicar declaraciones de intención claras con respecto a la compatibilidad, el uso y los cambios importantes. Incluso si son de mejor esfuerzo, no vinculantes o ignorados por muchos usuarios, todavía nos da un punto de partida para razonar acerca de si un cambio de última hora/mejora de versión principal “vale la pena”, sin traer estas estructuras de incentivos en conflicto.

Vamosy clausura ambos manejan esto muy bien: en sus ecosistemas de administración de paquetes estándar, se espera que el equivalente a una actualización de versión principal sea un paquete completamente nuevo. Esto tiene cierto sentido de justicia: si está dispuesto a romper la compatibilidad con versiones anteriores de su paquete, ¿por qué pretendemos que este es el mismo conjunto de API? Volver a empaquetar y cambiar el nombre de todo parece una cantidad razonable de trabajo que esperar de un proveedor a cambio de que tomen la opción nuclear y desechen la compatibilidad con versiones anteriores.

Finalmente, está la falibilidad humana del proceso. En general, los aumentos de versión de SemVer deben aplicarse a semánticos cambios tanto como los sintácticos; cambiar el comportamiento de una API es tan importante como cambiar su estructura. Aunque es plausible que se puedan desarrollar herramientas para evaluar si una versión en particular implica cambios sintácticos en un conjunto de API públicas, discernir si hay cambios semánticos significativos e intencionales es computacionalmente inviable.¹² En términos prácticos, incluso las herramientas potenciales para identificar cambios sintácticos son limitadas. En casi todos los casos, depende del juicio humano del proveedor de la API si se deben actualizar las versiones principales, secundarias o de parches para cualquier cambio dado. Si confía solo en un puñado de dependencias mantenidas profesionalmente, su exposición esperada a esta forma de error administrativo de SemVer es probablemente baja.¹³ Si tiene una red de miles de dependencias debajo de su producto, debe estar preparado para cierta cantidad de caos simplemente por error humano.

Selección de versión mínima

En 2018, como parte de una serie de ensayos sobre la creación de un sistema de gestión de paquetes para el lenguaje de programación Go, el propio Russ Cox de Google describió una variación interesante de la gestión de dependencias de SemVer:[Selección de versión mínima](#)(MVS). Al actualizar la versión de algún nodo en la red de dependencia, es posible que sus dependencias deban actualizarse a versiones más nuevas para satisfacer un requisito de SemVer actualizado; esto puede desencadenar más cambios transitivamente. En la mayoría de las formulaciones de selección de versión/satisfacción de restricciones, se eligen las versiones más nuevas posibles de esas dependencias posteriores: después de todo, eventualmente necesitará actualizar a esas nuevas versiones, ¿no?

MVS toma la decisión opuesta: cuando libra's especificación requiere librería ≥ 1.7 , lo intentaremos librería 1.7 directamente, incluso si hay disponible un 1.8. Esto "produce compilaciones de alta fidelidad en las que las dependencias que crea un usuario son lo más parecidas posible a las que crea el usuario".

12 En un mundo de pruebas unitarias ubicuas, podríamos identificar cambios que requerían un cambio en el comportamiento de la prueba, pero aún sería difícil separar algorítmicamente "Este es un cambio de comportamiento" de "Esta es una corrección de errores para un comportamiento que no estaba previsto/prometido".

13 Entonces, cuando sea importante a largo plazo, elija dependencias bien mantenidas.

autor desarrollado en contra.”¹⁴ Hay una verdad críticamente importante revelada en este punto: cuando el desarrollador dice que requiere la librería ≥ 1.7 , eso significa casi con certeza que el desarrollador deliberadamente instaló la librería 1.7. Suponiendo que el mantenedor realizó incluso pruebas básicas antes de publicar,¹⁵ tenemos al menos evidencia anecdótica de pruebas de interoperabilidad para esa versión delibera y la versión 1.7 de la librería. No es CI o prueba de que todo se ha probado en conjunto, pero es algo.

En ausencia de restricciones de entrada precisas derivadas de una predicción 100% precisa del futuro, es mejor dar el menor salto posible. Así como suele ser más seguro dedicar una hora de trabajo a su proyecto en lugar de volcar un año de trabajo de una sola vez, los pasos más pequeños hacia adelante en las actualizaciones de dependencia son más seguros. MVS simplemente avanza cada dependencia afectada solo hasta donde sea necesario y dice: “Está bien, avancé lo suficiente para obtener lo que solicitó (y no más). ¿Por qué no haces algunas pruebas y ves si todo está bien?

Inherente a la idea de MVS está la admisión de que una versión más nueva podría introducir una incompatibilidad en la práctica, incluso si los números de versión *En teoría* decir lo contrario. Esto es reconocer la principal preocupación con SemVer, usando MVS o no: hay cierta pérdida de fidelidad en esta compresión de cambios de software en números de versión. MVS brinda una fidelidad práctica adicional, tratando de producir versiones seleccionadas más cercanas a aquellas que presumiblemente se han probado juntas. Esto podría ser un impulso suficiente para hacer que un conjunto más grande de redes de dependencia funcione correctamente. Desafortunadamente, no hemos encontrado una buena manera de verificar empíricamente esa idea. El jurado aún está deliberando sobre si MVS hace que SemVer sea “lo suficientemente bueno” sin solucionar los problemas básicos teóricos y de incentivos con el enfoque, pero aún creemos que representa una mejora manifiesta en la aplicación de las restricciones de SemVer tal como se usan hoy.

Entonces, ¿Funciona SemVer?

SemVer funciona bastante bien en escalas limitadas. Sin embargo, es muy importante reconocer lo que realmente está diciendo y lo que no. SemVer funcionará bien siempre que:

- Sus proveedores de dependencia son precisos y responsables (para evitar errores humanos en los golpes de SemVer)
- Sus dependencias son detalladas (para evitar restricciones excesivas falsas cuando se actualizan las API no utilizadas o no relacionadas en sus dependencias y el riesgo asociado de requisitos de SemVer insatisfactorios)

14 Russ Cox, “Selección de versión mínima”, 21 de febrero de 2018, <https://research.swtch.com/vgo-mvs>. 15 Si

esa suposición no se cumple, realmente debería dejar de depender de la librería.

- Todo el uso de todas las API está dentro del uso esperado (para evitar que se interrumpa de manera sorprendente por un cambio supuestamente compatible, ya sea directamente o en el código del que depende transitivamente)

Cuando solo tiene unas pocas dependencias cuidadosamente seleccionadas y bien mantenidas en su gráfico de dependencia, SemVer puede ser una solución perfectamente adecuada.

Sin embargo, nuestra experiencia en Google sugiere que es poco probable que pueda tener *alguna* de esas tres propiedades a escala y mantenerlas funcionando constantemente a lo largo del tiempo. La escala tiende a ser lo que muestra las debilidades en SemVer. A medida que su red de dependencia se amplía, tanto en el tamaño de cada dependencia como en el número de dependencias (así como los efectos monorepo de tener múltiples proyectos que dependen de la misma red de dependencias externas), comenzará la pérdida de fidelidad compuesta en SemVer. dominar. Estas fallas se manifiestan como falsos positivos (versiones prácticamente incompatibles que en teoría deberían haber funcionado) y falsos negativos (versiones compatibles rechazadas por los solucionadores de SAT y el infierno de dependencia resultante).

Gestión de dependencias con recursos infinitos

Aquí hay un experimento mental útil cuando se consideran soluciones de gestión de dependencias: ¿cómo sería la gestión de dependencias si todos tuviéramos acceso a recursos informáticos infinitos? Es decir, ¿qué es lo mejor que podemos esperar, si no tenemos recursos limitados sino que estamos limitados solo por la visibilidad y la coordinación débil entre las organizaciones? Tal como lo vemos actualmente, la industria confía en SemVer por tres razones:

- Solo requiere información local (un proveedor de API no *necesitará* conocer los datos de los usuarios intermedios)
- No asume la disponibilidad de pruebas (todavía no omnipresente en la industria, pero definitivamente se moverá de esa manera en la próxima década), recursos informáticos para ejecutar las pruebas o sistemas de CI para monitorear los resultados de la prueba.
- Es la práctica existente

El "requisito" de información local no es realmente necesario, específicamente porque las redes de dependencia tienden a formarse en solo dos entornos:

- Dentro de una sola organización
- Dentro del ecosistema OSS, donde la fuente es visible incluso si los proyectos no necesariamente colaboran

En cualquiera de esos casos, la información significativa sobre el uso posterior es *disponible*, incluso si no está siendo fácilmente expuesto o no se actúa en la actualidad. Es decir, parte del dominio efectivo de SemVer es que elegimos ignorar información que teóricamente es

disponible para nosotros. Si tuviéramos acceso a más recursos informáticos y esa información de dependencia apareciera fácilmente, la comunidad probablemente encontraría un uso para ella.

Aunque un paquete OSS puede tener innumerables dependientes de código cerrado, el caso común es que los paquetes OSS populares son populares tanto pública como privadamente. Las redes de dependencia no mezclan (no pueden) agresivamente dependencias públicas y privadas: generalmente, hay un subconjunto público y un subgrafo privado separado.¹⁶

A continuación, debemos recordar la *intención* de SemVer: "En mi opinión, este cambio será fácil (o no) de adoptar". ¿Hay una mejor manera de transmitir esa información? Sí, en forma de experiencia práctica que demuestre que el cambio es fácil de adoptar. ¿Cómo obtenemos tal experiencia? Si la mayoría (o al menos una muestra representativa) de nuestras dependencias son visibles públicamente, ejecutamos las pruebas para esas dependencias con cada cambio propuesto. Con un número suficientemente grande de tales pruebas, tenemos al menos un argumento estadístico de que el cambio es seguro en el sentido práctico de la Ley de Hyrum. Las pruebas aún pasan, el cambio es bueno, no importa si esto afecta la API, corrige errores o cualquier otra cosa; no hay necesidad de clasificar o estimar.

Imaginemos, entonces, que el ecosistema OSS se trasladara a un mundo en el que los cambios estuvieran acompañados de evidencia si son seguros. Si sacamos los costos de cómputo de la ecuación, el *verdad*¹⁷ de "qué tan seguro es esto" proviene de ejecutar pruebas afectadas en dependencias posteriores.

Incluso sin un CI formal aplicado a todo el ecosistema OSS, por supuesto que podemos usar un gráfico de dependencia de este tipo y otras señales secundarias para hacer un análisis previo más específico. Priorice las pruebas en las dependencias que se utilizan mucho. Priorice las pruebas en dependencias que estén bien mantenidas. Priorice las pruebas en dependencias que tengan un historial de proporcionar una buena señal y resultados de prueba de alta calidad. Más allá de simplemente priorizar las pruebas en función de los proyectos que probablemente nos brinden la mayor cantidad de información sobre la calidad del cambio experimental, podríamos usar la información de los autores del cambio para ayudar a estimar el riesgo y seleccionar una estrategia de prueba adecuada. En teoría, es necesario ejecutar pruebas de "todos los afectados" si el objetivo es que "nada en lo que alguien confíe sea un cambio de manera radical".

En Capítulo 12, identificamos cuatro variedades de cambio, que van desde refactorizaciones puras hasta la modificación de la funcionalidad existente. Dado un modelo basado en CI para la actualización de dependencias, podemos comenzar a mapear esas variedades de cambio en un modelo similar a SemVer para el cual el autor de un cambio estima el riesgo y aplica un nivel apropiado de prueba. Por ejemplo, un cambio de refactorización pura que modifica solo las API internas

16 Debido a que la red pública de dependencia de OSS generalmente no puede depender de un montón de nodos privados, gráficos a pesar del firmware.

17 O algo muy parecido.

podría suponerse que es de bajo riesgo y justificar la ejecución de pruebas solo en nuestro propio proyecto y quizás en una muestra de dependientes directos importantes. Por otro lado, un cambio que elimine una interfaz obsoleta o cambie los comportamientos observables puede requerir tantas pruebas como podamos permitirnos.

¿Qué cambios necesitaríamos en el ecosistema OSS para aplicar dicho modelo?

Desafortunadamente, bastantes:

- Todas las dependencias deben proporcionar pruebas unitarias. Aunque nos estamos moviendo inexorablemente hacia un mundo en el que las pruebas unitarias son bien aceptadas y omnipresentes, todavía no hemos llegado allí.
- Se comprende la red de dependencia para la mayoría del ecosistema OSS. No está claro si algún mecanismo está actualmente disponible para realizar algoritmos gráficos en esa red; la información *espúber y disponible*, pero en realidad generalmente no indexado o utilizable. Muchos sistemas de administración de paquetes/ecosistemas de administración de dependencias le permiten ver las dependencias de un proyecto, pero no los bordes inversos, los dependientes.
- La disponibilidad de recursos informáticos para ejecutar CI aún es muy limitada. La mayoría de los desarrolladores no tienen acceso a clústeres de cómputo de compilación y prueba.
- Las dependencias a menudo se expresan de forma anclada. Como mantenedor de librería, no podemos ejecutar experimentalmente un cambio a través de las pruebas paralibay libres; esas dependencias dependen explícitamente de una versión anclada específica delibrería.
- Podríamos querer incluir explícitamente el historial y la reputación en los cálculos de CI. Un cambio propuesto que interrumpe un proyecto que tiene un largo historial de pruebas que continúan superándose nos brinda una forma de evidencia diferente a la de un quiebre en un proyecto que se agregó recientemente y tiene un historial de quiebres por razones no relacionadas.

Inherente a esto hay una pregunta de escala: ¿contra qué versiones de cada dependencia en la red prueba los cambios previos al envío? Si probamos contra la combinación completa de todas las versiones históricas, quemaremos una cantidad verdaderamente asombrosa de recursos informáticos, incluso para los estándares de Google. La simplificación más obvia de esta estrategia de selección de versión parecería ser "probar la versión estable actual" (después de todo, el desarrollo basado en troncales es el objetivo). Y por lo tanto, el modelo de gestión de la dependencia con recursos infinitos es efectivamente el del modelo Live at Head. La pregunta pendiente es si ese modelo se puede aplicar de manera efectiva con una disponibilidad de recursos más práctica y si los proveedores de API están dispuestos a asumir una mayor responsabilidad para probar la seguridad práctica de sus cambios.

Exportación de dependencias

Hasta ahora, solo hemos hablado de asumir dependencias; es decir, dependiendo del software que hayan escrito otras personas. También vale la pena pensar en cómo construimos software que puede ser usó como dependencia. Esto va más allá de la mecánica de empaquetar software y cargarlo en un repositorio: debemos pensar en los beneficios, costos y riesgos de proporcionar software, tanto para nosotros como para nuestros dependientes potenciales.

Hay dos formas principales en las que un acto inocuo y, con suerte, caritativo como "abrir una biblioteca" puede convertirse en una posible pérdida para una organización. Primero, eventualmente puede convertirse en un lastre para la reputación de su organización si se implementa de manera deficiente o no se mantiene adecuadamente. Como dice el refrán de la comunidad Apache, debemos priorizar la "comunidad sobre el código". Si proporciona un código excelente pero es un miembro deficiente de la comunidad, aún puede ser perjudicial para su organización y la comunidad en general. En segundo lugar, un lanzamiento bien intencionado puede convertirse en un impuesto a la eficiencia de la ingeniería si no puede mantener las cosas sincronizadas. Con el tiempo, todas las horquillas se volverán caras.

Ejemplo: gflags de código abierto

Para la pérdida de reputación, considere el caso de algo así como la experiencia de Google alrededor de 2006 de fuente abierta de nuestras bibliotecas de banderas de línea de comandos de C++. Seguramente retribuir a la comunidad de código abierto es un acto puramente bueno que no volverá a atormentarnos, ¿verdad? Tristemente no. Una gran cantidad de razones conspiraron para convertir este buen acto en algo que ciertamente perjudicó nuestra reputación y posiblemente dañó también a la comunidad OSS:

- En ese momento, no teníamos la capacidad de ejecutar refactorizaciones a gran escala, por lo que todo lo que usaba esa biblioteca internamente tenía que permanecer exactamente igual: no podíamos mover el código a una nueva ubicación en la base de código.
- Separamos nuestro repositorio en "código desarrollado internamente" (que se puede copiar libremente si es necesario bifurcarlo, siempre que se le cambie el nombre correctamente) y "código que puede tener problemas legales/de licencia" (que puede tener más requisitos de uso matizados).
- Si un proyecto OSS acepta código de desarrolladores externos, generalmente es un problema legal: el creador del proyecto no propiamente aporte, sólo tienen derechos sobre él.

Como resultado, el proyecto gflags estaba condenado a ser un lanzamiento de "tirar por encima del muro" o una bifurcación desconectada. Los parches aportados al proyecto no se podían reincorporar a la fuente original dentro de Google, y no podíamos mover el proyecto dentro de nuestro monorepo porque aún no habíamos dominado esa forma de refactorización, ni podíamos hacer que todo dependiera internamente de la versión S.O.S.

Además, como la mayoría de las organizaciones, nuestras prioridades han cambiado y cambiado con el tiempo. En la época del lanzamiento original de esa biblioteca de banderas, estábamos interesados en

productos fuera de nuestro espacio tradicional (aplicaciones web, búsqueda), incluidas cosas como Google Earth, que tenía un mecanismo de distribución mucho más tradicional: binarios precompilados para una variedad de plataformas. A fines de la década de 2000, era inusual pero no inaudito que una biblioteca en nuestro monorepo, especialmente algo de bajo nivel como banderas, se usara en una variedad de plataformas. A medida que pasaba el tiempo y Google crecía, nuestro enfoque se redujo hasta el punto de que era extremadamente raro que se crearan bibliotecas con algo que no fuera nuestra cadena de herramientas configurada internamente y luego se implementaran en nuestra flota de producción. Las preocupaciones de "portabilidad" para respaldar adecuadamente un proyecto OSS como las banderas eran casi imposibles de mantener: nuestras herramientas internas simplemente no tenían soporte para esas plataformas y nuestro desarrollador promedio no tenía que interactuar con herramientas externas.

A medida que los autores originales y los partidarios de OSS pasaron a nuevas empresas o nuevos equipos, finalmente quedó claro que nadie internamente apoyaba realmente nuestro proyecto de banderas de OSS; nadie podía vincular ese apoyo con las prioridades de ningún equipo en particular. Dado que no era un trabajo específico del equipo, y nadie podía decir por qué era importante, no sorprende que básicamente dejamos que ese proyecto se pudra externamente.¹⁸Las versiones internas y externas divergieron lentamente con el tiempo y, finalmente, algunos desarrolladores externos tomaron la versión externa y la bifurcaron, prestándole la debida atención.

Aparte del inicial "Oh, mira, Google contribuyó con algo al mundo del código abierto", nada de eso nos hizo quedar bien y, sin embargo, cada pequeña parte tenía sentido dadas las prioridades de nuestra organización de ingeniería. Aquellos de nosotros que hemos estado cerca hemos aprendido: "No publique cosas sin un plan (y un mandato) para apoyarlo a largo plazo". Queda por ver si toda la ingeniería de Google ha aprendido eso o no. Es una gran organización.

18 Eso no quiere decir que sea *derechoo intelectual*, solo que, como organización, dejamos que algunas cosas se nos escapen.

Más allá del nebuloso "Nos vemos mal", también hay partes de esta historia que ilustran cómo podemos estar sujetos a problemas técnicos derivados de dependencias externas mal liberadas o mal mantenidas. Aunque la biblioteca de banderas se compartió pero se ignoró, todavía había algunos proyectos de código abierto respaldados por Google, o proyectos que debían poder compartirse fuera de nuestro ecosistema monorepo. Como era de esperar, los autores de esos otros proyectos pudieron identificar¹⁹ el subconjunto de API común entre las bifurcaciones internas y externas de esa biblioteca. Debido a que ese subconjunto común se mantuvo bastante estable entre las dos versiones durante un período prolongado, se convirtió silenciosamente en "la forma de hacer esto" para los equipos raros que tenían requisitos de portabilidad inusuales entre aproximadamente 2008 y 2017. Su código podría compilarse tanto en y ecosistemas externos, cambiando versiones bifurcadas de la biblioteca de banderas según el entorno.

Luego, por razones no relacionadas, los equipos de bibliotecas de C++ comenzaron a ajustar piezas observables pero no documentadas de la implementación de banderas internas. En ese momento, todos los que dependían de la estabilidad y la equivalencia de una bifurcación externa sin soporte comenzaron a gritar que sus compilaciones y lanzamientos se rompieron repentinamente. Una oportunidad de optimización por valor de varios miles de CPU agregadas en toda la flota de Google se retrasó significativamente, no porque fuera difícil actualizar la API de la que dependían 250 millones de líneas de código, sino porque un pequeño puñado de proyectos dependía de cosas no prometidas e inesperadas. Una vez más, la Ley de Hyrum afecta los cambios de software, en este caso incluso para API bifurcadas mantenidas por organizaciones separadas.

Estudio de caso: App Engine

Un ejemplo más serio de exponernos a un mayor riesgo de dependencia técnica inesperada proviene de la publicación del servicio AppEngine de Google. Este servicio permite a los usuarios escribir sus aplicaciones sobre un marco existente en uno de varios lenguajes de programación populares. Siempre que la aplicación esté escrita con un modelo adecuado de administración de estado/almacenamiento, el servicio AppEngine permite que esas aplicaciones se amplíen a niveles de uso enormes: la infraestructura de producción de Google administra y clona el almacenamiento de respaldo y la administración de frontend a pedido.

Originalmente, la compatibilidad de AppEngine con Python era una compilación de 32 bits que se ejecutaba con una versión anterior del intérprete de Python. El propio sistema AppEngine fue (por supuesto) implementado en nuestro monorepo y construido con el resto de nuestras herramientas comunes, en Python y en C++ para soporte de back-end. En 2014, comenzamos el proceso de realizar una actualización importante del tiempo de ejecución de Python junto con nuestro compilador de C++ y las instalaciones de la biblioteca estándar, con el resultado de que vinculamos efectivamente el "código que se compila con el compilador de C++ actual" al "código que usa el compilador de Python actualizado". versión": un proyecto que actualizó una de esas dependencias actualizó inherentemente la otra al mismo tiempo. Para la mayoría

¹⁹ A menudo a través de prueba y error.

proyectos, esto no era un problema. Para algunos proyectos, debido a los casos extremos y la Ley de Hyrum, nuestros expertos en plataformas de idiomas terminaron investigando y depurando para desbloquear la transición. En un caso aterrador en el que la Ley de Hyrum se encontró con aspectos prácticos comerciales, AppEngine descubrió que muchos de sus usuarios, nuestros clientes que pagan, no podían (o no querían) actualizar: o bien no querían tomar el cambio a la versión más nueva. versión de Python, o no podían permitirse los cambios en el consumo de recursos que implicaba pasar de Python de 32 bits a 64 bits. Debido a que había algunos clientes que estaban pagando una cantidad significativa de dinero por los servicios de AppEngine, AppEngine pudo presentar un caso comercial sólido de que se debe retrasar un cambio forzado al nuevo lenguaje y versiones del compilador. Esto significaba inherentemente que cada pieza de código C++ en el cierre transitivo de las dependencias de AppEngine tenía que ser compatible con las versiones anteriores del compilador y de la biblioteca estándar: cualquier corrección de errores u optimización de rendimiento que pudiera realizarse en esa infraestructura tenía que ser compatible entre versiones. Esa situación persistió durante casi tres años.

Con suficientes usuarios, cualquier "observable" de su sistema llegará a depender de alguien. En Google, restringimos a todos nuestros usuarios internos dentro de los límites de nuestra pila técnica y aseguramos la visibilidad de su uso con los sistemas de indexación de código y monorepo, por lo que es mucho más fácil garantizar que el cambio útil siga siendo posible. Cuando cambiamos del control de código fuente a la gestión de dependencias y perdemos visibilidad sobre cómo se usa el código o estamos sujetos a prioridades contrapuestas de grupos externos (especialmente los que le están pagando), se vuelve mucho más difícil hacer concesiones puramente de ingeniería. La liberación de API de cualquier tipo lo expone a la posibilidad de prioridades contrapuestas y restricciones imprevistas por parte de personas ajenas. Esto no quiere decir que no deba lanzar API; solo sirve para proporcionar el recordatorio:

Compartir código con el mundo exterior, ya sea como un lanzamiento de código abierto o como un lanzamiento de biblioteca de código cerrado, no es una simple cuestión de caridad (en el caso de OSS) o una oportunidad comercial (en el caso de código cerrado). Los usuarios dependientes que no puede monitorear, en diferentes organizaciones, con diferentes prioridades, eventualmente ejercerán alguna forma de inercia de la Ley de Hyrum en ese código. Especialmente si está trabajando con escalas de tiempo largas, es imposible predecir con precisión el conjunto de cambios necesarios o útiles que podrían volverse valiosos. Al evaluar si lanzar algo, tenga en cuenta los riesgos a largo plazo: las dependencias compartidas externamente suelen ser mucho más costosas de modificar con el tiempo.

Conclusión

La administración de dependencias es inherentemente desafiante: estamos buscando soluciones para la administración de superficies de API complejas y redes de dependencias, donde los mantenedores de esas dependencias generalmente asumen poca o ninguna coordinación. El estándar de facto para administrar una red de dependencias es el control de versiones semántico, o SemVer, que proporciona un resumen con pérdidas del riesgo percibido al adoptar cualquier cambio en particular. SemVer presupone que podemos predecir a priori la gravedad de un cambio, en ausencia de conocimiento de cómo se está consumiendo la API en cuestión: la Ley de Hyrum nos informa lo contrario. Sin embargo, SemVer funciona lo suficientemente bien a pequeña escala, e incluso mejor cuando incluimos el enfoque MVS. A medida que crece el tamaño de la red de dependencia,

Sin embargo, es posible que avancemos hacia un mundo en el que las estimaciones de compatibilidad proporcionadas por el mantenedor (números de versión de SemVer) se descarten en favor de la evidencia basada en la experiencia: ejecutar las pruebas de los paquetes posteriores afectados. Si los proveedores de API asumen una mayor responsabilidad de realizar pruebas con sus usuarios y anuncian claramente qué tipos de cambios se esperan, tenemos la posibilidad de redes de dependencia de mayor fidelidad a una escala aún mayor.

TL; DR

- Prefiera los problemas de control de fuente a los problemas de gestión de dependencias: si puede obtener más código de su organización para tener una mejor transparencia y coordinación, esas son simplificaciones importantes.
- Agregar una dependencia no es gratis para un proyecto de ingeniería de software, y la complejidad de establecer una relación de confianza “continua” es un desafío. La importación de dependencias a su organización debe hacerse con cuidado, con una comprensión de los costos de soporte continuos.
- Una dependencia es un contrato: hay un toma y daca, y tanto los proveedores como los consumidores tienen algunos derechos y responsabilidades en ese contrato. Los proveedores deben tener claro lo que están tratando de prometer a lo largo del tiempo.
- SemVer es una estimación abreviada de compresión con pérdida para “¿Qué tan arriesgado cree un ser humano que es este cambio?” SemVer con un solucionador SAT en un administrador de paquetes toma esas estimaciones y las escala para que funcionen como valores absolutos. Esto puede dar como resultado una restricción excesiva (infierno de dependencia) o una restricción insuficiente (versiones que deberían funcionar juntas pero que no lo hacen).
- En comparación, las pruebas y la CI brindan evidencia real de si un nuevo conjunto de versiones funciona en conjunto.

- Las estrategias de actualización de versión mínima en SemVer/administración de paquetes son de mayor fidelidad. Esto todavía depende de que los humanos puedan evaluar el riesgo de versión incremental con precisión, pero mejora claramente la posibilidad de que un experto haya probado el vínculo entre el proveedor de API y el consumidor.
- Las pruebas unitarias, la CI y los recursos informáticos (baratos) tienen el potencial de cambiar nuestra comprensión y enfoque de la gestión de dependencias. Ese cambio de fase requiere un cambio fundamental en la forma en que la industria considera el problema de la gestión de la dependencia y las responsabilidades tanto de los proveedores como de los consumidores.
- Proporcionar una dependencia no es gratis: “tíralo por la borda y olvídalos” puede costarle reputación y convertirse en un desafío para la compatibilidad. Apoyarlo con estabilidad puede limitar sus opciones y pesimizar el uso interno. Apoyar sin estabilidad puede costar buena voluntad o exponerlo al riesgo de que grupos externos importantes dependan de algo a través de la Ley de Hyrum y estropeen su plan de “falta de estabilidad”.

Cambios a gran escala

*Escrito por Hyrum Wright
Editado por Lisa Carey*

Piense por un momento en su propio código base. ¿Cuántos archivos puede actualizar de manera confiable en una sola confirmación simultánea? ¿Cuáles son los factores que limitan ese número? ¿Alguna vez has intentado cometer un cambio tan grande? ¿Sería capaz de hacerlo en un tiempo razonable en una emergencia? ¿Cómo se compara su mayor tamaño de compromiso con el tamaño real de su base de código? ¿Cómo probarías tal cambio? ¿Cuántas personas necesitarían revisar el cambio antes de que se confirme? ¿Sería capaz de revertir ese cambio si se comprometiera? Las respuestas a estas preguntas pueden sorprenderte (tanto lo que *pensar* cuáles son las respuestas y cuáles resultan ser realmente para su organización).

En Google, hace tiempo que abandonamos la idea de realizar cambios radicales en nuestra base de código en este tipo de grandes cambios atómicos. Nuestra observación ha sido que, a medida que crece el código base y el número de ingenieros que trabajan en él, el mayor cambio atómico posible contraintuitivamente *disminuye*—ejecutar todas las comprobaciones y pruebas previas al envío afectadas se vuelve difícil, por no hablar de incluso asegurarse de que todos los archivos del cambio estén actualizados antes del envío. Dado que se ha vuelto más difícil realizar cambios radicales en nuestro código base, dado nuestro deseo general de poder mejorar continuamente la infraestructura subyacente, hemos tenido que desarrollar nuevas formas de razonar sobre los cambios a gran escala y cómo implementarlos.

En este capítulo, hablaremos sobre las técnicas, tanto sociales como técnicas, que nos permiten mantener la gran base de código de Google flexible y receptiva a los cambios en la infraestructura subyacente. También proporcionaremos algunos ejemplos de la vida real de cómo y dónde hemos utilizado estos enfoques. Aunque es posible que su base de código no se parezca a la de Google, comprender estos principios y adaptarlos localmente ayudará a su organización de desarrollo a escalar y, al mismo tiempo, podrá realizar cambios amplios en su base de código.

¿Qué es un cambio a gran escala?

Antes de ir mucho más lejos, debemos profundizar en lo que califica como un cambio a gran escala (LSC). En nuestra experiencia, un LSC es cualquier conjunto de cambios que están lógicamente relacionados pero que prácticamente no pueden presentarse como una sola unidad atómica. Esto podría deberse a que toca tantos archivos que las herramientas subyacentes no pueden confirmarlos todos a la vez, o podría deberse a que el cambio es tan grande que siempre tendría conflictos de fusión. En muchos casos, la topología de su repositorio dicta un LSC: si su organización usa una colección de repositorios distribuidos o federados,¹ hacer cambios atómicos a través de ellos podría ni siquiera ser técnicamente posible.² Veremos las posibles barreras a los cambios atómicos con más detalle más adelante en este capítulo.

Los LSC en Google casi siempre se generan mediante herramientas automatizadas. Las razones para hacer un LSC varían, pero los cambios en sí generalmente se dividen en algunas categorías básicas:

- Limpieza de antipatrones comunes mediante el uso de herramientas de análisis de toda la base de código
- Sustitución de usos de funciones de biblioteca en desuso
- Permitir mejoras de infraestructura de bajo nivel, como actualizaciones del compilador
- Mover usuarios de un sistema antiguo a uno más nuevo³

La cantidad de ingenieros que trabajan en estas tareas específicas en una organización determinada puede ser baja, pero es útil para sus clientes conocer las herramientas y el proceso de LSC. Por su propia naturaleza, los LSC afectarán a una gran cantidad de clientes, y las herramientas de LSC se reducen fácilmente a equipos que realizan solo unas pocas docenas de cambios relacionados.

Puede haber causas motivadoras más amplias detrás de LSC específicos. Por ejemplo, un nuevo estándar de lenguaje podría introducir un idioma más eficiente para realizar una tarea determinada, una interfaz de biblioteca interna podría cambiar o una nueva versión del compilador podría requerir la reparación de problemas existentes que la nueva versión marcaría como errores. La mayoría de los LSC en Google en realidad tienen un impacto funcional cercano a cero: tienden a ser actualizaciones textuales generalizadas para mayor claridad, optimización o compatibilidad futura. Pero los LSC no se limitan teóricamente a esta clase de cambio de conservación/refactorización del comportamiento.

En todos estos casos, en una base de código del tamaño de Google, los equipos de infraestructura pueden necesitar cambiar de forma rutinaria cientos de miles de referencias individuales a la antigua

1 Para algunas ideas sobre por qué, consulte [capítulo 16](#).

2 Es posible en este mundo federado decir "simplemente nos comprometemos con cada repositorio lo más rápido posible para mantener la duración".

¡La construcción se rompe pequeña! Pero ese enfoque realmente no escala a medida que crece la cantidad de repositorios federados.

3 Para una discusión más detallada sobre esta práctica, véase [Capítulo 15](#).

patrón o símbolo. En los casos más grandes hasta ahora, hemos tocado millones de referencias y esperamos que el proceso continúe escalando bien. En general, hemos encontrado ventajoso invertir temprano y con frecuencia en herramientas para habilitar LSC para los muchos equipos que realizan trabajos de infraestructura. También hemos encontrado que las herramientas eficientes también ayudan a los ingenieros a realizar cambios más pequeños. Las mismas herramientas que hacen eficiente el cambio de miles de archivos también se reducen razonablemente a decenas de archivos.

¿Quién se ocupa de los LSC?

Como se acaba de indicar, los equipos de infraestructura que construyen y administran nuestros sistemas son responsables de gran parte del trabajo de realizar los LSC, pero las herramientas y los recursos están disponibles en toda la empresa. Si te saltaste [Capítulo 1](#), quizás se pregunte por qué los equipos de infraestructura son los responsables de este trabajo. ¿Por qué no podemos simplemente introducir una nueva clase, función o sistema y dictar que todos los que usan el anterior pasen al análogo actualizado? Aunque esto puede parecer más fácil en la práctica, resulta que no se escala muy bien por varias razones.

En primer lugar, los equipos de infraestructura que construyen y administran los sistemas subyacentes también son los que tienen el conocimiento del dominio necesario para corregir los cientos de miles de referencias a ellos. Es poco probable que los equipos que consumen la infraestructura tengan el contexto para manejar muchas de estas migraciones, y es globalmente ineficiente esperar que cada uno de ellos vuelva a aprender la experiencia que los equipos de infraestructura ya tienen. La centralización también permite una recuperación más rápida cuando se enfrentan a errores porque los errores generalmente se dividen en un pequeño conjunto de categorías, y el equipo que ejecuta la migración puede tener un libro de jugadas, formal o informal, para abordarlos.

Considere la cantidad de tiempo que lleva hacer el primero de una serie de cambios semimecánicos que no comprende. Probablemente dedique algún tiempo a leer sobre la motivación y la naturaleza del cambio, encuentre un ejemplo sencillo, intente seguir las sugerencias proporcionadas y luego intente aplicarlo a su código local. Repetir esto para todos los equipos de una organización aumenta considerablemente el costo total de ejecución. Al hacer que solo unos pocos equipos centralizados sean responsables de los LSC, Google internaliza esos costos y los reduce al hacer posible que el cambio suceda de manera más eficiente.

En segundo lugar, a nadie le gustan los mandatos sin financiación.⁴ Aunque un nuevo sistema puede ser categóricamente mejor que el que reemplaza, esos beneficios a menudo se difunden en una organización y, por lo tanto, es poco probable que importen lo suficiente como para que los equipos individuales quieran actualizar por su propia iniciativa. Si el nuevo sistema es lo suficientemente importante como para migrar,

4 Por "mandato no financiado" nos referimos a "requisitos adicionales impuestos por una entidad externa sin equilibrar compensación." Algo así como cuando el director ejecutivo dice que todo el mundo debe usar un traje de noche para los "viernes formales", pero no te da el aumento correspondiente para pagar tu ropa formal.

los costos de la migración correrán a cargo de algún lugar de la organización. Centralizar la migración y contabilizar sus costos es casi siempre más rápido y económico que depender de equipos individuales para migrar orgánicamente.

Además, tener equipos que sean dueños de los sistemas que requieren LSC ayuda a alinear los incentivos para garantizar que se lleve a cabo el cambio. Según nuestra experiencia, es poco probable que las migraciones orgánicas tengan un éxito total, en parte porque los ingenieros tienden a usar el código existente como ejemplo cuando escriben código nuevo. Contar con un equipo que tenga un interés personal en eliminar el antiguo sistema responsable del esfuerzo de migración ayuda a garantizar que realmente se lleve a cabo. Aunque financiar y dotar de personal a un equipo para ejecutar este tipo de migraciones puede parecer un costo adicional, en realidad es solo internalizar las externalidades que crea un mandato sin financiamiento, con los beneficios adicionales de las economías de escala.

Estudio de caso: Relleno de baches

Si bien los sistemas LSC de Google se usan para migraciones de alta prioridad, también descubrimos que el solo hecho de tenerlos disponibles abre oportunidades para varias correcciones pequeñas en nuestra base de código, que simplemente no habrían sido posibles sin ellas. Al igual que las tareas de infraestructura de transporte consisten en construir nuevas carreteras y reparar las antiguas, los grupos de infraestructura de Google dedican mucho tiempo a corregir el código existente, además de desarrollar nuevos sistemas y trasladar a los usuarios a ellos.

Por ejemplo, al principio de nuestra historia, surgió una biblioteca de plantillas para complementar la biblioteca de plantillas estándar de C++. Bien llamada Biblioteca de plantillas de Google, esta biblioteca constaba de la implementación de varios archivos de encabezado. Por razones perdidas en la noche de los tiempos, uno de estos archivos de encabezado se denominó *stl_util.h* y otro fue nombrado *mapa-util.h* (tenga en cuenta los diferentes separadores en los nombres de los archivos). Además de volver locos a los puristas de la consistencia, esta diferencia también condujo a una productividad reducida, y los ingenieros tuvieron que recordar qué archivo usaba qué separador, y solo descubrieron cuándo se equivocaron después de un ciclo de compilación potencialmente largo.

Aunque corregir este cambio de un solo carácter puede parecer inútil, especialmente en una base de código del tamaño de la de Google, la madurez de nuestro proceso y herramientas LSC nos permitió hacerlo con solo un par de semanas de trabajo en segundo plano. Los autores de la biblioteca pudieron encontrar y aplicar este cambio en masa sin tener que molestar a los usuarios finales de estos archivos, y pudimos reducir cuantitativamente la cantidad de fallas de compilación causadas por este problema específico. Los aumentos resultantes en la productividad (y la felicidad) compensaron con creces el tiempo necesario para realizar el cambio.

A medida que ha mejorado la capacidad de realizar cambios en todo nuestro código base, la diversidad de cambios también se ha ampliado y podemos tomar algunas decisiones de ingeniería sabiendo que no son inmutables en el futuro. A veces, vale la pena el esfuerzo de llenar algunos baches.

Barreras a los cambios atómicos

Antes de analizar el proceso que usa Google para efectuar los LSC, debemos hablar sobre por qué muchos tipos de cambios no se pueden realizar de forma atómica. En un mundo ideal, todos los cambios lógicos podrían empaquetarse en una sola confirmación atómica que podría probarse, revisarse y confirmarse independientemente de otros cambios. Desafortunadamente, a medida que crece el depósito y la cantidad de ingenieros que trabajan en él, ese ideal se vuelve menos factible. Puede ser completamente inviable incluso a pequeña escala cuando se utiliza un conjunto de repositorios distribuidos o federados.

Limitaciones técnicas

Para empezar, la mayoría de los sistemas de control de versiones (VCS) tienen operaciones que escalan linealmente con el tamaño de un cambio. Es posible que su sistema pueda manejar confirmaciones pequeñas (por ejemplo, del orden de decenas de archivos) sin problemas, pero es posible que no tenga suficiente memoria o capacidad de procesamiento para confirmar atómicamente miles de archivos a la vez. En los VCS centralizados, las confirmaciones pueden impedir que otros escritores (y, en sistemas más antiguos, los lectores) usen el sistema mientras procesan, lo que significa que las confirmaciones grandes paralizan a otros usuarios del sistema.

En resumen, puede que no sea solo "difícil" o "imprudente" hacer un gran cambio atómicamente: puede que simplemente sea imposible con una infraestructura dada. Dividir el cambio grande en partes más pequeñas e independientes evita estas limitaciones, aunque hace que la ejecución del cambio sea más compleja.⁵

Fusionar conflictos

A medida que crece el tamaño de un cambio, también aumenta el potencial de conflictos de fusión. Cada sistema de control de versiones que conocemos requiere actualización y fusión, potencialmente con resolución manual, si existe una versión más nueva de un archivo en el repositorio central. A medida que aumenta la cantidad de archivos en un cambio, la probabilidad de encontrar un conflicto de fusión también crece y se ve agravada por la cantidad de ingenieros que trabajan en el repositorio.

Si su empresa es pequeña, es posible que pueda introducir un cambio que afecte a todos los archivos del repositorio durante un fin de semana en el que nadie esté desarrollando. O puede tener un sistema informal para obtener el bloqueo del repositorio global al pasar un token virtual (¡o incluso físico!) a su equipo de desarrollo. En una gran empresa global como Google, estos enfoques simplemente no son factibles: alguien siempre está haciendo cambios en el repositorio.

5 Ver <https://ieeexplore.ieee.org/abstract/document/8443579>.

Con pocos archivos en un cambio, la probabilidad de conflictos de combinación se reduce, por lo que es más probable que se confirmen sin problemas. Esta propiedad también es válida para las siguientes áreas.

Sin cementerios embrujados

Las SRE que administran los servicios de producción de Google tienen un mantra: "No a los cementerios embrujados". Un cementerio embrujado en este sentido es un sistema tan antiguo, obtuso o complejo que nadie se atreve a entrar en él. Los cementerios embrujados suelen ser sistemas críticos para la empresa que se congelan en el tiempo porque cualquier intento de cambiarlos podría hacer que el sistema falle de manera incomprensible, lo que le costaría dinero real a la empresa. Plantean un riesgo existencial real y pueden consumir una cantidad desmesurada de recursos.

Sin embargo, los cementerios embrujados no solo existen en los sistemas de producción; se pueden encontrar en las bases de código. Muchas organizaciones tienen fragmentos de software que son antiguos y no reciben mantenimiento, escritos por alguien que hace mucho tiempo que no forma parte del equipo y que se encuentra en la ruta crítica de alguna funcionalidad importante que genera ingresos. Estos sistemas también están congelados en el tiempo, con capas de burocracia construidas para evitar cambios que puedan causar inestabilidad. ¡Nadie quiere ser el ingeniero de soporte de red II que se equivocó!

Estas partes de una base de código son un anatema para el proceso de LSC porque evitan la finalización de grandes migraciones, el desmantelamiento de otros sistemas de los que dependen o la actualización de compiladores o bibliotecas que utilizan. Desde la perspectiva de un LSC, los cementerios embrujados impiden todo tipo de progreso significativo.

En Google, hemos encontrado que el contraataque a esto es una buena prueba a la antigua. Cuando el software se prueba exhaustivamente, podemos hacerle cambios arbitrarios y saber con confianza si esos cambios son importantes, sin importar la antigüedad o la complejidad del sistema. Escribir esas pruebas requiere mucho esfuerzo, pero permite que un código base como el de Google evolucione durante largos períodos de tiempo, relegando la noción de cementerios de software embrujados a un cementerio propio.

Heterogeneidad

Los LSC realmente funcionan solo cuando la mayor parte del esfuerzo puede ser realizado por computadoras, no por humanos. Tan buenos como los humanos pueden ser con la ambigüedad, las computadoras dependen de entornos consistentes para aplicar las transformaciones de código adecuadas en los lugares correctos. Si su organización tiene muchos VCS diferentes, sistemas de integración continua (CI), herramientas específicas del proyecto o pautas de formato, es difícil realizar cambios radicales en toda su base de código. Simplificar el entorno para agregar más consistencia ayudará tanto a los humanos que necesitan moverse en él como a los robots que realizan transformaciones automatizadas.

Por ejemplo, muchos proyectos en Google tienen pruebas de envío previo configuradas para ejecutarse antes de que se realicen cambios en su base de código. Esas verificaciones pueden ser muy complejas, desde verificar nuevas dependencias contra una lista blanca hasta ejecutar pruebas y garantizar que el cambio tenga un error asociado. Muchas de estas comprobaciones son relevantes para los equipos que escriben nuevas funciones, pero para los LSC, solo agregan una complejidad irrelevante adicional.

Hemos decidido adoptar parte de esta complejidad, como ejecutar pruebas previas al envío, haciéndolo estándar en nuestra base de código. Para otras inconsistencias, recomendamos a los equipos que omitan sus controles especiales cuando partes de LSC toquen su código de proyecto. La mayoría de los equipos están felices de ayudar dado el beneficio que este tipo de cambios son para sus proyectos.



Muchos de los beneficios de la consistencia para los humanos mencionados en [Capítulo 8](#) también se aplican a las herramientas automatizadas.

Pruebas

Cada cambio debe probarse (un proceso del que hablaremos más en un momento), pero cuanto más grande sea el cambio, más difícil será probarlo correctamente. El sistema CI de Google ejecutará no solo las pruebas afectadas inmediatamente por un cambio, sino también cualquier prueba que dependa transitivamente de los archivos modificados.⁶ Esto significa que un cambio obtiene una amplia cobertura, pero también hemos observado que cuanto más lejos en el gráfico de dependencia se encuentra una prueba de los archivos afectados, más improbable es que el cambio en sí haya causado una falla.

Los cambios pequeños e independientes son más fáciles de validar, porque cada uno de ellos afecta a un conjunto más pequeño de pruebas, pero también porque las fallas de las pruebas son más fáciles de diagnosticar y corregir. Encontrar la causa raíz de una falla de prueba en un cambio de 25 archivos es bastante sencillo; encontrar 1 en un cambio de 10.000 archivos es como la proverbial aguja en un pajar.

La contrapartida de esta decisión es que los cambios más pequeños harán que las mismas pruebas se ejecuten varias veces, en particular las pruebas que dependen de gran parte de la base de código. Debido a que el tiempo de ingeniería dedicado a rastrear las fallas de las pruebas es mucho más costoso que el tiempo de cómputo requerido para ejecutar estas pruebas adicionales, hemos tomado la decisión consciente de que esta es una compensación que estamos dispuestos a hacer. Esa misma compensación puede no ser válida para todas las organizaciones, pero vale la pena examinar cuál es el equilibrio adecuado para la suya.

⁶ Esto probablemente suene a exageración, y probablemente lo sea. Estamos investigando activamente la mejor forma de determinar el conjunto "correcto" de pruebas para un cambio dado, equilibrando el costo del tiempo de cómputo para ejecutar las pruebas y el costo humano de tomar la decisión incorrecta.

Estudio de caso: prueba de LSC

Adán doblador

Hoy en día, es común que un porcentaje de dos dígitos (10% a 20%) de los cambios en un proyecto sea el resultado de LSC, lo que significa que una cantidad sustancial de código es modificada en proyectos por personas cuyo trabajo de tiempo completo no está relacionado con esos proyectos. Sin buenas pruebas, dicho trabajo sería imposible y la base de código de Google se atrofiaría rápidamente por su propio peso. Los LSC nos permiten migrar sistemáticamente todo nuestro código base a API más nuevas, desaprobar API más antiguas, cambiar versiones de idioma y eliminar prácticas populares pero peligrosas.

Incluso un simple cambio de firma de una línea se vuelve complicado cuando se realiza en miles de lugares diferentes en cientos de productos y servicios diferentes.⁷Después de escribir el cambio, debe coordinar las revisiones de código entre docenas de equipos. Por último, una vez que se aprueban las revisiones, debe realizar tantas pruebas como sea posible para asegurarse de que el cambio sea seguro. ⁸Decimos "tantas como puedas", porque un LSC de buen tamaño podría desencadenar una repetición de cada prueba en Google, y eso puede llevar un tiempo. De hecho, muchos LSC tienen que planificar el tiempo para atrapar a los clientes finales cuyo código retrocede mientras el LSC avanza en el proceso.

Probar un LSC puede ser un proceso lento y frustrante. Cuando un cambio es lo suficientemente grande, es casi seguro que su entorno local no estará sincronizado permanentemente con head, ya que la base de código se desplaza como la arena alrededor de su trabajo. En tales circunstancias, es fácil encontrarse ejecutando y volviendo a ejecutar pruebas solo para asegurarse de que sus cambios sigan siendo válidos. Cuando un proyecto tiene pruebas irregulares o falta cobertura de pruebas unitarias, puede requerir mucha intervención manual y ralentizar todo el proceso. Para ayudar a acelerar las cosas, utilizamos una estrategia llamada tren TAP (Plataforma de Automatización de Pruebas).

Viajar en el tren TAP

La idea central de los LSC es que rara vez interactúan entre sí, y la mayoría de las pruebas afectadas pasarán para la mayoría de los LSC. Como resultado, podemos probar más de un cambio a la vez y reducir el número total de pruebas ejecutadas. El modelo de tren ha demostrado ser muy efectivo para probar los LSC.

El tren TAP se beneficia de dos hechos:

- Los LSC tienden a ser refactorizaciones puras y, por lo tanto, tienen un alcance muy limitado, conservando la semántica local.

⁷ La mayor serie de LSC jamás ejecutada eliminó más de mil millones de líneas de código del repositorio durante el transcurso de tres días. Esto fue en gran parte para eliminar una parte obsoleta del repositorio que se había migrado a un nuevo hogar; pero aun así, ¿qué confianza hay que tener para eliminar mil millones de líneas de código?

⁸ Los LSC suelen estar respaldados por herramientas que hacen que encontrar, realizar y revisar cambios sea relativamente sencillo adelante.

- Los cambios individuales a menudo son más simples y están muy analizados, por lo que la mayoría de las veces son correctos.

El modelo de tren también tiene la ventaja de que funciona para múltiples cambios al mismo tiempo y no requiere que cada cambio individual viaje de forma aislada.⁹

El tren tiene cinco pasos y se inicia de nuevo cada tres horas:

1. Para cada cambio en el tren, ejecute una muestra de 1000 pruebas seleccionadas al azar.
2. Reúna todos los cambios que pasaron sus 1000 pruebas y cree un supercambio a partir de todos ellos: "el tren".
3. Ejecutar la unión de todas las pruebas directamente afectadas por el grupo de cambios. Dado un LSC lo suficientemente grande (o de bajo nivel), esto puede significar ejecutar todas las pruebas en el repositorio de Google. Este proceso puede tardar más de seis horas en completarse.
4. Para cada prueba que no falle, vuelve a ejecutarla individualmente contra cada cambio que se hizo en el tren para determinar qué cambios causaron que fallara.
5. TAP genera un informe por cada cambio que subió al tren. El informe describe todos los objetivos aprobados y fallidos y se puede utilizar como prueba de que es seguro enviar un LSC.

Revisión de código

Finalmente, como mencionamos en [Capítulo 9](#), todos los cambios deben revisarse antes de enviarlos, y esta política se aplica incluso a los LSC. Revisar confirmaciones grandes puede ser tedioso, oneroso e incluso propenso a errores, especialmente si los cambios se generan a mano (un proceso que desea evitar, como veremos en breve). En solo un momento, veremos cómo las herramientas a menudo pueden ayudar en este espacio, pero para algunas clases de cambios, aún queremos que los humanos verifiquen explícitamente que sean correctos. Dividir un LSC en fragmentos separados hace que esto sea mucho más fácil.

Estudio de caso: `scoped_ptr` a `std::unique_ptr`

Desde sus primeros días, la base de código C++ de Google ha tenido un puntero inteligente autodestructivo para envolver objetos C++ asignados al almacenamiento dinámico y garantizar que se destruyan cuando el puntero inteligente se sale del alcance. Este tipo se llamaba `alcance_ptr` se usó ampliamente en la base de código de Google para garantizar que la vida útil de los objetos se administrara adecuadamente. No era perfecto, pero dadas las limitaciones del estándar C++ vigente en ese momento (C++98) cuando se introdujo el tipo por primera vez, hizo que los programas fueran más seguros.

⁹ Es posible solicitar a TAP una ejecución "aislada" de un solo cambio, pero son muy costosas y se realizan solo durante las horas de menor actividad.

En C++ 11, el lenguaje introdujo un nuevo tipo:estándar::único_ptr. Cumplía la misma función que alcance_ptr, pero también evitó otras clases de errores que el lenguaje ahora podría detectar. estándar::único_ptr era estrictamente mejor que alcance_ptr, sin embargo, el código base de Google tenía más de 500.000 referencias a alcance_ptr dispersas entre millones de archivos fuente. Pasar al tipo más moderno requirió el intento de LSC más grande hasta ese momento dentro de Google.

En el transcurso de varios meses, varios ingenieros atacaron el problema en paralelo. Gracias a la infraestructura de migración a gran escala de Google, pudimos cambiar las referencias a alcance_ptr en referencias a estándar::único_ptr, así como adaptarse lentamente a alcance_ptr comportándose más cerca de estándar::único_ptr. En el punto álgido del proceso de migración, generamos, probamos y confirmamos constantemente más de 700 cambios independientes, tocando más de 15.000 archivos, *por día*. Hoy en día, a veces gestionamos 10 veces ese rendimiento, habiendo perfeccionado nuestras prácticas y mejorado nuestras herramientas.

Como casi todos los LSC, este tuvo una cola muy larga de rastrear varias dependencias de comportamiento matizadas (otra manifestación de la Ley de Hyrum), luchar contra las condiciones de carrera con otros ingenieros y usos en el código generado que no fueron detectables por nuestras herramientas automatizadas. . Continuamos trabajando en estos manualmente a medida que fueron descubiertos por la infraestructura de prueba.

alcance_ptr también se usó como un tipo de parámetro en algunas API ampliamente utilizadas, lo que dificultó los pequeños cambios independientes. Contemplamos escribir un sistema de análisis de gráficos de llamadas que pudiera cambiar una API y sus llamadores, transitivamente, en una confirmación, pero nos preocupaba que los cambios resultantes fueran demasiado grandes para confirmarlos atómicamente.

Al final, finalmente pudimos eliminar alcance_ptr primero convirtiéndolo en un tipo de alias de estándar::único_ptr y luego realizar la sustitución textual entre el alias antiguo y el nuevo, antes de eliminar finalmente el antiguo alcance_ptr alias. En la actualidad, la base de código de Google se beneficia del uso del mismo tipo estándar que el resto del ecosistema de C++, lo cual fue posible solo gracias a nuestra tecnología y herramientas para LSC.

Infraestructura LSC

Google ha invertido en una cantidad significativa de infraestructura para hacer posibles los LSC. Esta infraestructura incluye herramientas para la creación de cambios, la gestión de cambios, la revisión de cambios y las pruebas. Sin embargo, quizás el apoyo más importante para los LSC ha sido la evolución de las normas culturales en torno a los cambios a gran escala y la supervisión que se les ha dado. Aunque los conjuntos de herramientas técnicas y sociales pueden diferir para su organización, los principios generales deben ser los mismos.

Políticas y Cultura

Como hemos descrito en [capítulo 16](#), Google almacena la mayor parte de su código fuente en un único repositorio monolítico (monorepo), y cada ingeniero tiene visibilidad de casi todo este código. Este alto grado de apertura significa que cualquier ingeniero puede editar cualquier archivo y enviar esas ediciones para su revisión a quienes puedan aprobarlas. Sin embargo, cada una de esas ediciones tiene costos, tanto para generar como para revisar.¹⁰

Históricamente, estos costos han sido algo simétricos, lo que limitaba el alcance de los cambios que podía generar un solo ingeniero o equipo. A medida que mejoraba la herramienta LSC de Google, se volvió más fácil generar una gran cantidad de cambios de manera muy económica, y se volvió igualmente fácil para un solo ingeniero imponer una carga a una gran cantidad de revisores en toda la empresa. Aunque queremos fomentar mejoras generalizadas en nuestro código base, queremos asegurarnos de que haya cierta supervisión y consideración detrás de ellas, en lugar de ajustes indiscriminados.¹¹

El resultado final es un proceso de aprobación ligero para equipos e individuos que buscan hacer LSC en Google. Este proceso es supervisado por un grupo de ingenieros experimentados que están familiarizados con los matices de varios idiomas, así como por expertos de dominio invitados para el cambio particular en cuestión. El objetivo de este proceso no es prohibir los LSC, sino ayudar a los autores de cambios a producir los mejores cambios posibles, que aprovechen al máximo el capital técnico y humano de Google. Ocasionalmente, este grupo puede sugerir que una limpieza simplemente no vale la pena: por ejemplo, limpiar un error tipográfico común sin ninguna forma de prevenir que vuelva a ocurrir.

Relacionado con estas políticas hubo un cambio en las normas culturales que rodean a los LSC. Si bien es importante que los propietarios del código tengan un sentido de responsabilidad por su software, también debían aprender que los LSC eran una parte importante del esfuerzo de Google para escalar nuestras prácticas de ingeniería de software. Así como los equipos de productos son los que están más familiarizados con su propio software, los equipos de infraestructura bibliotecaria conocen los matices de la infraestructura, y hacer que los equipos de productos confíen en esa experiencia en el dominio es un paso importante hacia la aceptación social de los LSC. Como resultado de este cambio de cultura, los equipos de productos locales han llegado a confiar en los autores de LSC para realizar cambios relevantes para los dominios de esos autores.

Ocasionalmente, los propietarios locales cuestionan el propósito de una confirmación específica que se realiza como parte de un LSC más amplio, y los autores de cambios responden a estos comentarios tal como lo harían con otros comentarios de revisión. Desde el punto de vista social, es importante que los propietarios del código comprendan los cambios que se producen en su software, pero también se han dado cuenta de que no tienen derecho de veto sobre el LSC más amplio. Con el tiempo, hemos descubierto que una buena sección de preguntas frecuentes y

¹⁰ Hay costos técnicos obvios aquí en términos de cómputo y almacenamiento, pero los costos humanos en tiempo para revisar un cambio superan con creces los técnicos.

¹¹ Por ejemplo, no queremos que las herramientas resultantes se utilicen como un mecanismo para pelear por la ortografía correcta de "gris" o "gris" en los comentarios.

un sólido historial histórico de mejoras ha generado un respaldo generalizado de los LSC en todo Google.

Perspectiva de la base de código

Para hacer LSC, hemos encontrado que es invaluable poder hacer un análisis a gran escala de nuestra base de código, tanto a nivel textual usando herramientas tradicionales como a nivel semántico. Por ejemplo, el uso de Google de la herramienta de indexación semántica [Kythé](#) proporciona un mapa completo de los enlaces entre partes de nuestra base de código, lo que nos permite hacer preguntas como "¿Dónde están las personas que llaman a esta función?" o "¿Qué clases se derivan de ésta?" Kythe y herramientas similares también brindan acceso programático a sus datos para que puedan incorporarse a las herramientas de refactorización. (Para más ejemplos, véanse los capítulos [17](#) y [20](#).)

También usamos índices basados en compiladores para ejecutar transformaciones y análisis basados en árboles de sintaxis abstracta sobre nuestra base de código. Herramientas como [ClangMR](#), [Java Flume](#) o [Refrescador](#), que pueden realizar transformaciones de una manera altamente paralelizable, dependen de estos conocimientos como parte de su función. Para cambios más pequeños, los autores pueden usar herramientas especializadas y personalizadas, perlas o [sed](#), coincidencia de expresiones regulares, o incluso un simple script de shell.

Independientemente de la herramienta que utilice su organización para la creación de cambios, es importante que su esfuerzo humano escalé de forma sublineal con el código base; en otras palabras, debería llevar aproximadamente la misma cantidad de tiempo humano generar la recopilación de todos los cambios necesarios, sin importar el tamaño del repositorio. Las herramientas de creación de cambios también deben ser integrales en todo el código base, de modo que un autor pueda estar seguro de que su cambio cubre todos los casos que está tratando de solucionar.

Al igual que con otras áreas de este libro, una inversión temprana en herramientas suele dar sus frutos a corto o mediano plazo. Como regla general, hemos sostenido durante mucho tiempo que si un cambio requiere más de 500 ediciones, generalmente es más eficiente para un ingeniero aprender y ejecutar nuestras herramientas de generación de cambios en lugar de ejecutar manualmente esa edición. Para los "conserjes de código" experimentados, ese número suele ser mucho menor.

Gestión del cambio

Podría decirse que la pieza más importante de la infraestructura de cambio a gran escala es el conjunto de herramientas que fragmenta un cambio maestro en piezas más pequeñas y gestiona el proceso de prueba, envío, revisión y confirmación de forma independiente. En Google, esta herramienta se llama Rosie, y discutiremos su uso más completamente en unos momentos cuando examinamos nuestro proceso LSC. En muchos aspectos, Rosie no es solo una herramienta, sino una plataforma completa para hacer LSC a escala de Google. Brinda la capacidad de dividir los grandes conjuntos de cambios integrales producidos por las herramientas en fragmentos más pequeños, que se pueden probar, revisar y enviar de forma independiente.

Pruebas

Las pruebas son otra pieza importante de la infraestructura que permite cambios a gran escala. Como se discutió en [Capítulo 11](#), las pruebas son una de las formas importantes en que validamos que nuestro software se comporte como se espera. Esto es particularmente importante cuando se aplican cambios que no son creados por humanos. Una cultura e infraestructura de pruebas sólidas significa que otras herramientas pueden estar seguras de que estos cambios no tienen efectos no deseados.

La estrategia de prueba de Google para los LSC difiere ligeramente de la de los cambios normales y aún utiliza la misma infraestructura de CI subyacente. Probar los LSC significa no solo garantizar que el gran cambio maestro no cause fallas, sino que cada fragmento se pueda enviar de manera segura e independiente. Debido a que cada fragmento puede contener archivos arbitrarios, no usamos las pruebas estándar de envío previo basadas en proyectos. En cambio, ejecutamos cada fragmento sobre el cierre transitivo de cada prueba a la que podría afectar, lo cual discutimos anteriormente.

Ayuda de idioma

Los LSC en Google generalmente se realizan por idioma, y algunos idiomas los admiten mucho más fácilmente que otros. Descubrimos que las características del lenguaje, como el alias de tipo y las funciones de reenvío, son invaluables para permitir que los usuarios existentes continúen funcionando mientras presentamos nuevos sistemas y migramos usuarios a ellos de forma no atómica. Para los lenguajes que carecen de estas funciones, a menudo es difícil migrar sistemas de forma incremental.¹²

También hemos descubierto que es mucho más fácil realizar grandes cambios automáticos en los lenguajes escritos estáticamente que en los lenguajes escritos dinámicamente. Las herramientas basadas en compiladores junto con un sólido análisis estático brindan una cantidad significativa de información que podemos usar para crear herramientas que afecten a los LSC y rechacen las transformaciones no válidas incluso antes de que lleguen a la fase de prueba. El desafortunado resultado de esto es que los lenguajes como Python, Ruby y JavaScript que se escriben dinámicamente son más difíciles para los mantenedores. La elección del idioma está, en muchos aspectos, íntimamente ligada a la cuestión de la vida útil del código: los idiomas que tienden a verse como más enfocados en la productividad del desarrollador tienden a ser más difíciles de mantener. Aunque este no es un requisito de diseño intrínseco, es donde se encuentra el estado actual del arte.

Finalmente, vale la pena señalar que los formateadores automáticos de idiomas son una parte crucial de la infraestructura de LSC. Debido a que trabajamos para optimizar la legibilidad de nuestro código, queremos asegurarnos de que cualquier cambio producido por las herramientas automatizadas sea inteligible tanto para los revisores inmediatos como para los futuros lectores del código. Todas las herramientas de generación de LSC ejecutan el formateador automático apropiado para el idioma que se cambia como un paso separado para que las herramientas específicas del cambio no necesiten

12 De hecho, Go introdujo recientemente este tipo de funciones de lenguaje específicamente para admitir refactorizaciones a gran escala. (ver <https://talks.golang.org/2016/refactor.article>).

preocuparse por los detalles de formato. Aplicar formato automático, como [google-formato-javafomato clang](#), a nuestra base de código significa que los cambios producidos automáticamente "encajarán" con el código escrito por un ser humano, lo que reduce la fricción en el desarrollo futuro. Sin el formato automático, los cambios automáticos a gran escala nunca se habrían convertido en el statu quo aceptado en Google.

Estudio de caso: Operación RoseHub

Los LSC se han convertido en una gran parte de la cultura interna de Google, pero están comenzando a tener implicaciones en el mundo en general. Quizás el caso más conocido hasta ahora fue "[Operación RoseHub](#)".

A principios de 2017, una vulnerabilidad en la biblioteca Apache Commons permitió que cualquier aplicación Java con una versión vulnerable de la biblioteca en su classpath transitivo se volviera susceptible a la ejecución remota. Este error se conoció como Mad Gadget. Entre otras cosas, permitió que un hacker codicioso cifrara los sistemas de la Agencia de Transporte Municipal de San Francisco y cerrara sus operaciones. Debido a que el único requisito para la vulnerabilidad era tener la biblioteca incorrecta en algún lugar de su classpath, cualquier cosa que dependiera incluso de uno de los muchos proyectos de código abierto en Git-Hub era vulnerable.

Para resolver este problema, algunos Googlers emprendedores lanzaron su propia versión del proceso LSC. Mediante el uso de herramientas como [BigQuery](#), los voluntarios identificaron los proyectos afectados y enviaron más de 2600 parches para actualizar sus versiones de la biblioteca Commons a una que abordara Mad Gadget. En lugar de herramientas automatizadas que gestionan el proceso, más de 50 humanos hicieron que este LSC funcionara.

El proceso LSC

Con estas piezas de infraestructura en su lugar, ahora podemos hablar sobre el proceso para hacer un LSC. Esto se divide aproximadamente en cuatro fases (con límites muy nebulosos entre ellas):

1. Autorización
2. Cambio de creación
3. Gestión de fragmentos
4. Limpieza

Por lo general, estos pasos ocurren después de que se haya escrito un nuevo sistema, clase o función, pero es importante tenerlos en cuenta durante el diseño del nuevo sistema. En Google, nuestro objetivo es diseñar sistemas sucesores teniendo en cuenta una ruta de migración desde sistemas más antiguos, de modo que los encargados del mantenimiento del sistema puedan trasladar a sus usuarios al nuevo sistema automáticamente.

Autorización

Pedimos a los autores potenciales que completen un breve documento que explique el motivo de un cambio propuesto, su impacto estimado en la base de código (es decir, cuántos fragmentos más pequeños generaría el cambio grande) y las respuestas a cualquier pregunta que puedan tener los revisores potenciales. Este proceso también obliga a los autores a pensar en cómo describirán el cambio a un ingeniero que no esté familiarizado con él en forma de preguntas frecuentes y una descripción del cambio propuesto. Los autores también obtienen una "revisión de dominio" de los propietarios de la API que se está refactorizando.

Luego, esta propuesta se envía a una lista de correo electrónico con aproximadamente una docena de personas que supervisan todo el proceso. Después de la discusión, el comité da su opinión sobre cómo avanzar. Por ejemplo, uno de los cambios más comunes realizados por el comité es dirigir todas las revisiones de código para que un LSC vaya a un solo "aprobador global". Muchos autores de LSC primerizos tienden a suponer que los propietarios de proyectos locales deben revisar todo, pero para la mayoría de los LSC mecánicos, es más económico tener un solo experto que comprenda la naturaleza del cambio y construya la automatización para revisarlo correctamente.

Una vez que se aprueba el cambio, el autor puede continuar con la presentación del cambio. Históricamente, el comité ha sido muy liberal con su aprobación,¹³ y, a menudo, da su aprobación no solo para un cambio específico, sino también para un amplio conjunto de cambios relacionados. Los miembros del comité pueden, a su discreción, acelerar los cambios obvios sin necesidad de una deliberación completa.

La intención de este proceso es brindar supervisión y una ruta de escalada, sin ser demasiado oneroso para los autores del LSC. El comité también está facultado como órgano de escalamiento de inquietudes o conflictos sobre un LSC: los propietarios locales que no estén de acuerdo con el cambio pueden apelar a este grupo, que luego puede arbitrar cualquier conflicto. En la práctica, esto rara vez ha sido necesario.

Creación de cambios

Después de obtener la aprobación requerida, un autor de LSC comenzará a producir las ediciones de código reales. A veces, estos pueden generarse de manera integral en un solo gran cambio global que posteriormente se fragmentará en muchas piezas independientes más pequeñas. Por lo general, el tamaño del cambio es demasiado grande para caber en un solo cambio global, debido a las limitaciones técnicas del sistema de control de versiones subyacente.

13 Los únicos tipos de cambios que el comité ha rechazado rotundamente han sido aquellos que se consideran peligrosos. Pueden ser cambios que cambien la forma en que se manejan los errores, como convertir todos los NULO en instancias apunto nulo, o de valor extremadamente bajo, como cambiar la ortografía del inglés británico al inglés americano, o viceversa. A medida que nuestra experiencia con dichos cambios ha aumentado y el costo de los LSC ha disminuido, el umbral de aprobación también lo ha hecho.

El proceso de generación de cambios debe estar lo más automatizado posible para que el cambio principal pueda actualizarse a medida que los usuarios retroceden a usos anteriores.¹⁴o se producen conflictos de combinación textual en el código modificado. Ocasionalmente, para el raro caso en el que las herramientas técnicas no pueden generar el cambio global, hemos fragmentado la generación de cambios entre humanos (ver "[Estudio de caso: Operación RoseHub](#)" en la página 472). Aunque es mucho más laborioso que generar cambios automáticamente, esto permite que los cambios globales sucedan mucho más rápido para aplicaciones sensibles al tiempo.

Tenga en cuenta que optimizamos para la legibilidad humana de nuestra base de código, por lo que cualquiera que sea la herramienta que genere cambios, queremos que los cambios resultantes se parezcan lo más posible a los cambios generados por humanos. Este requisito conduce a la necesidad de guías de estilo y herramientas de formato automático (ver [Capítulo 8](#)).¹⁵

Fragmentación y envío

Después de generar un cambio global, el autor comienza a ejecutar Rosie. Rosie toma un gran cambio y lo fragmenta según los límites del proyecto y las reglas de propiedad en cambios que *puedes* ser presentado atómicamente. Luego, coloca cada cambio fragmentado individualmente a través de una canalización de envío de correo de prueba independiente. Rosie puede ser una gran usuaria de otras piezas de la infraestructura de desarrollo de Google, por lo que limita la cantidad de fragmentos pendientes para cualquier LSC determinado, se ejecuta con menor prioridad y se comunica con el resto de la infraestructura sobre la cantidad de carga aceptable para generar en nuestra infraestructura de prueba compartida.

Hablamos más sobre el proceso de envío de correo de prueba específico para cada fragmento a continuación.

Ganado versus mascotas

A menudo usamos la analogía de "ganado y mascotas" cuando nos referimos a máquinas individuales en un entorno informático distribuido, pero los mismos principios pueden aplicarse a los cambios dentro de una base de código.

En Google, como en la mayoría de las organizaciones, los cambios típicos en la base de código son hechos a mano por ingenieros individuales que trabajan en funciones específicas o correcciones de errores. Los ingenieros pueden pasar días o semanas trabajando en la creación, prueba y revisión de un solo cambio. Llegan a conocer íntimamente el cambio y se enorgullecen cuando finalmente se envía al repositorio principal. La creación de tal cambio es similar a tener y criar una mascota favorita.

14 Esto sucede por muchas razones: copiar y pegar de ejemplos existentes, confirmar cambios que se han implementado durante algún tiempo, o simplemente la confianza en los viejos hábitos.

15 En realidad, este es el razonamiento detrás del trabajo original sobre el formato clang para C++.

Por el contrario, el manejo eficaz de los LSC requiere un alto grado de automatización y produce una enorme cantidad de cambios individuales. En este entorno, hemos encontrado útil tratar los cambios específicos como ganado: confirmaciones sin nombre y sin rostro que pueden revertirse o rechazarse en cualquier momento con un costo mínimo a menos que toda la manada se vea afectada. A menudo, esto sucede debido a un problema imprevisto que no detectan las pruebas, o incluso a algo tan simple como un conflicto de combinación.

Con una confirmación "mascota", puede ser difícil no tomarse el rechazo como algo personal, pero cuando se trabaja con muchos cambios como parte de un cambio a gran escala, es simplemente la naturaleza del trabajo. Tener automatización significa que las herramientas se pueden actualizar y generar nuevos cambios a muy bajo costo, por lo que perder un poco de ganado de vez en cuando no es un problema.

Pruebas

Cada fragmento independiente se prueba ejecutándolo a través de TAP, el marco de CI de Google. Ejecutamos todas las pruebas que dependen de los archivos en un cambio determinado de manera transitiva, lo que a menudo crea una gran carga en nuestro sistema de CI.

Esto puede sonar computacionalmente costoso, pero en la práctica, la gran mayoría de los fragmentos afectan a menos de mil pruebas, de los millones en nuestra base de código. Para aquellos que afectan más, podemos agruparlos: primero ejecutando la unión de todas las pruebas afectadas para todos los fragmentos, y luego para cada fragmento individual ejecutando solo la intersección de sus pruebas afectadas con aquellas que fallaron la primera ejecución. La mayoría de estas uniones hacen que se ejecuten casi todas las pruebas en el código base, por lo que agregar cambios adicionales a ese lote de fragmentos es casi gratis.

Uno de los inconvenientes de ejecutar una cantidad tan grande de pruebas es que los eventos independientes de baja probabilidad son casi certezas a una escala lo suficientemente grande. Pruebas escamosas y quebradizas, como las discutidas en [Capítulo 11](#), que a menudo no dañan a los equipos que los escriben y mantienen, son particularmente difíciles para los autores de LSC. Aunque el impacto es bastante bajo para los equipos individuales, las pruebas irregulares pueden afectar seriamente el rendimiento de un sistema LSC. Los sistemas automáticos de detección y eliminación de escamas ayudan con este problema, pero puede ser un esfuerzo constante garantizar que los equipos que escriben pruebas escamosas sean los que cubran sus costos.

En nuestra experiencia con los LSC como cambios generados por máquinas que conservan la semántica, ahora tenemos mucha más confianza en la corrección de un solo cambio que en una prueba con un historial reciente de descamación, tanto que las pruebas de descamación recientes ahora se ignoran cuando se envían a través de nuestras herramientas automatizadas. En teoría, esto significa que un solo fragmento puede causar una regresión que solo se detecta mediante una prueba inestable que pasa de ser inestable a fallar. En la práctica, vemos esto tan raramente que es más fácil manejarlo a través de la comunicación humana en lugar de la automatización.

Para cualquier proceso LSC, los fragmentos individuales deben poder cometerse de forma independiente. Esto significa que no tienen ninguna interdependencia o que el mecanismo de fragmentación puede

agrupar cambios dependientes (como un archivo de encabezado y su implementación) juntos. Al igual que cualquier otro cambio, los fragmentos de cambios a gran escala también deben pasar controles específicos del proyecto antes de ser revisados y confirmados.

Revisores de correo

Una vez que Rosie ha validado que un cambio es seguro mediante pruebas, envía el cambio por correo a un revisor adecuado. En una empresa tan grande como Google, con miles de ingenieros, el descubrimiento del revisor en sí mismo es un problema desafiante. Recordar de [Capítulo 9](#) ese código en el repositorio está organizado con archivos OWNERS, que enumeran a los usuarios con privilegios de aprobación para un subárbol específico en el repositorio. Rosie utiliza un servicio de detección de propietarios que comprende estos archivos de PROPIETARIOS y pondera a cada propietario en función de su capacidad esperada para revisar el fragmento específico en cuestión. Si un propietario en particular no responde, Rosie agrega revisores adicionales automáticamente en un esfuerzo por revisar un cambio de manera oportuna.

Como parte del proceso de envío de correos, Rosie también ejecuta las herramientas de confirmación previa por proyecto, que pueden realizar comprobaciones adicionales. Para los LSC, deshabilitamos de forma selectiva ciertas comprobaciones, como las del formato de descripción de cambio no estándar. Si bien son útiles para cambios individuales en proyectos específicos, estas comprobaciones son una fuente de heterogeneidad en la base de código y pueden agregar una fricción significativa al proceso LSC. Esta heterogeneidad es una barrera para escalar nuestros procesos y sistemas, y no se puede esperar que las herramientas y los autores de LSC entiendan las políticas especiales para cada equipo.

También ignoramos agresivamente las fallas de verificación previas al envío que existen antes del cambio en cuestión. Cuando se trabaja en un proyecto individual, es fácil para un ingeniero corregirlos y continuar con su trabajo original, pero esa técnica no escala cuando se crean LSC en la base de código de Google. Los propietarios del código local son responsables de no tener fallas preexistentes en su base de código como parte del contrato social entre ellos y los equipos de infraestructura.

Revisando

Al igual que con otros cambios, se espera que los cambios generados por Rosie pasen por el proceso de revisión de código estándar. En la práctica, hemos descubierto que los propietarios locales no suelen tratar los LSC con el mismo rigor que los cambios regulares: confían demasiado en los ingenieros que generan los LSC. Idealmente, estos cambios se revisarían como cualquier otro, pero en la práctica, los propietarios de proyectos locales han llegado a confiar en los equipos de infraestructura hasta el punto en que estos cambios a menudo solo reciben una revisión superficial. Ahora solo enviamos cambios a los propietarios locales para los que se requiere su revisión por contexto, no solo permisos de aprobación. Todos los demás cambios pueden ir a un "aprobador global": alguien que tiene derechos de propiedad para aprobar *alguna* cambiar en todo el repositorio.

Cuando se usa un aprobador global, todos los fragmentos individuales se asignan a esa persona, en lugar de a propietarios individuales de diferentes proyectos. Aprobadores globales en general

tener un conocimiento específico del idioma y/o las bibliotecas que están revisando y trabajar con el autor del cambio a gran escala para saber qué tipo de cambios esperar. Saben cuáles son los detalles del cambio y qué posibles modos de falla podrían existir y pueden personalizar su flujo de trabajo en consecuencia.

En lugar de revisar cada cambio individualmente, los revisores globales usan un conjunto separado de herramientas basadas en patrones para revisar cada uno de los cambios y aprobar automáticamente los que cumplen con sus expectativas. Por lo tanto, necesitan examinar manualmente solo un pequeño subconjunto que es anómalo debido a conflictos de fusión o mal funcionamiento de las herramientas, lo que permite que el proceso se escale muy bien.

Sumisión

Finalmente, se confirman los cambios individuales. Al igual que con el paso de envío por correo, nos aseguramos de que el cambio pase las diversas comprobaciones previas al compromiso del proyecto antes de que finalmente se confirme en el repositorio.

Con Rosie, podemos crear, probar, revisar y enviar de manera efectiva miles de cambios por día en todo el código base de Google y hemos brindado a los equipos la capacidad de migrar a sus usuarios de manera efectiva. Las decisiones técnicas que solían ser definitivas, como el nombre de un símbolo ampliamente utilizado o la ubicación de una clase popular dentro de un código base, ya no necesitan ser definitivas.

Limpiar

Diferentes LSC tienen diferentes definiciones de "hecho", que pueden variar desde eliminar por completo un sistema antiguo hasta migrar solo referencias de alto valor y dejar que las antiguas desaparezcan orgánicamente.¹⁶ En casi todos los casos, es importante contar con un sistema que evite introducciones adicionales del símbolo o sistema que el cambio a gran escala trabajó duro para eliminar. En Google, usamos el marco Tricorder mencionado en los capítulos 20 y 19 para marcar en el momento de la revisión cuando un ingeniero introduce un nuevo uso de un objeto en desuso, y esto ha demostrado ser un método eficaz para evitar la reincidencia. Hablamos más sobre todo el proceso de desaprobación en [Capítulo 15](#).

Conclusión

Los LSC forman una parte importante del ecosistema de ingeniería de software de Google. En el momento del diseño, abren más posibilidades, sabiendo que algunas decisiones de diseño no necesitan ser tan fijas como antes. El proceso LSC también permite a los mantenedores de la infraestructura central la capacidad de migrar grandes franjas de la base de código de Google desde sistemas antiguos, versiones de idioma y modismos de biblioteca a otros nuevos, manteniendo la base de código constante.

16 Lamentablemente, los sistemas que más queremos descomponer orgánicamente son aquellos que son más resistentes para hacerlo.

Son los anillos de plástico del paquete de seis del ecosistema del código.

consistente, espacial y temporalmente. Y todo esto sucede con solo unas pocas docenas de ingenieros que brindan apoyo a decenas de miles de personas más.

No importa el tamaño de su organización, es razonable pensar en cómo haría este tipo de cambios radicales en su colección de código fuente. Ya sea por elección o por necesidad, tener esta capacidad permitirá una mayor flexibilidad a medida que su organización crece mientras mantiene su código fuente maleable con el tiempo.

TL; DR

- Un proceso LSC permite repensar la inmutabilidad de determinadas decisiones técnicas.
- Los modelos tradicionales de refactorización se rompen a gran escala.
- Hacer LSC significa hacer un hábito de hacer LSC.

Integración continua

*Escrito por Rachel Tannenbaum
Editado por Lisa Carey*

Integración continua, o CI, generalmente se define como “una práctica de desarrollo de software en la que los miembros de un equipo integran su trabajo con frecuencia [...] Cada integración se verifica mediante una compilación automatizada (incluida la prueba) para detectar errores de integración lo más rápido posible.”¹. En pocas palabras, el objetivo fundamental de CI es detectar automáticamente los cambios problemáticos lo antes posible.

En la práctica, ¿qué significa “integrar el trabajo con frecuencia” para la aplicación distribuida moderna? Los sistemas actuales tienen muchas piezas en movimiento más allá del código con la última versión en el repositorio. De hecho, con la tendencia reciente hacia los microservicios, es menos probable que los cambios que rompen una aplicación vivan dentro del código base inmediato del proyecto y es más probable que estén en microservicios poco acoplados al otro lado de una llamada de red. Mientras que una compilación continua tradicional prueba los cambios en su binario, una extensión de esto podría probar los cambios en los microservicios ascendentes. La dependencia simplemente se cambia de su pila de llamadas de funciones a una solicitud HTTP o llamadas a procedimientos remotos (RPC).

Aún más allá de las dependencias de código, una aplicación puede ingerir datos periódicamente o actualizar modelos de aprendizaje automático. Puede ejecutarse en sistemas operativos, tiempos de ejecución, servicios de alojamiento en la nube y dispositivos en evolución. Puede ser una característica que se asienta sobre una plataforma en crecimiento o ser la plataforma que debe adaptarse a una base de características en crecimiento. Todas estas cosas deben considerarse dependencias, y también debemos intentar “integrar continuamente” sus cambios. Para complicar aún más las cosas, estos componentes cambiantes a menudo son propiedad de desarrolladores fuera de nuestro equipo, organización o empresa y se implementan en sus propios horarios.

¹<https://www.martinfowler.com/articles/continuousIntegration.html>

Entonces, quizás una mejor definición para CI en el mundo de hoy, particularmente cuando se desarrolla a escala, es la siguiente:

Integración Continua (2): el ensamblaje y las pruebas continuas de todo nuestro ecosistema complejo y en rápida evolución.

Es natural conceptualizar la IC en términos de pruebas porque los dos están estrechamente relacionados, y lo haremos a lo largo de este capítulo. En capítulos anteriores, hemos discutido una amplia gama de pruebas, desde la unidad hasta la integración, hasta sistemas de mayor alcance.

Desde una perspectiva de prueba, CI es un paradigma para informar lo siguiente:

- *Cuáles* pruebas para ejecutar *cuándo*en el flujo de trabajo de desarrollo/lanzamiento, ya que los cambios de código (y otros) se integran continuamente en él
- *Cómo* para componer el sistema bajo prueba (SUT) en cada punto, equilibrando preocupaciones como la fidelidad y el costo de instalación

Por ejemplo, ¿qué pruebas ejecutamos en el envío previo, cuáles guardamos para el envío posterior y cuáles guardamos incluso más tarde hasta la implementación de la etapa? En consecuencia, ¿cómo representamos nuestro COU en cada uno de estos puntos? Como puede imaginar, los requisitos para un SUT de preenvío pueden diferir significativamente de los de un entorno de prueba bajo prueba. Por ejemplo, puede ser peligroso que una aplicación creada a partir de un código pendiente de revisión en el envío previo se comunique con los backends de producción reales (piense en las vulnerabilidades de seguridad y cuotas), mientras que esto suele ser aceptable para un entorno de prueba.

Y por qué Deberíamos tratar de optimizar este equilibrio a menudo delicado de probar "las cosas correctas" en "los momentos correctos" con CI? Numerosos trabajos anteriores ya han establecido los beneficios de CI para la organización de ingeniería y el negocio en general por igual.² Estos resultados están impulsados por una poderosa garantía: prueba verificable y oportuna de que la aplicación es buena para avanzar a la siguiente etapa. No necesitamos esperar que todos los colaboradores sean muy cuidadosos, responsables y minuciosos; en cambio, podemos garantizar el estado de funcionamiento de nuestra aplicación en varios puntos desde la creación hasta el lanzamiento, mejorando así la confianza y la calidad en nuestros productos y la productividad de nuestros equipos.

En el resto de este capítulo, presentaremos algunos conceptos clave, mejores prácticas y desafíos de CI antes de ver cómo administramos la CI en Google con una introducción a nuestra herramienta de desarrollo continuo, TAP, y un estudio en profundidad de una aplicación.

Transformación CI.

² Forsgren, Nicole, et al. (2018). Acelerar: la ciencia del software Lean y DevOps: construcción y escalado Organizaciones Tecnológicas de Alto Rendimiento. Revolución TI.

Conceptos de CI

Primero, comencemos analizando algunos conceptos básicos de CI.

Bucles de retroalimentación rápidos

Como se discutió en [Capítulo 11](#), el costo de un error crece casi exponencialmente cuanto más tarde se detecta. [Figura 23-1](#) muestra todos los lugares en los que se puede detectar un cambio de código problemático durante su vida útil.

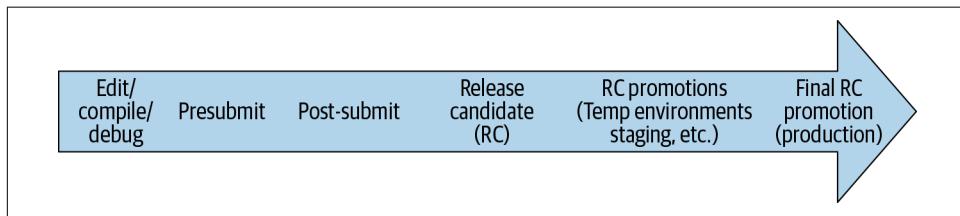


Figura 23-1. Vida de un cambio de código

En general, a medida que los problemas avanzan hacia la "derecha" en nuestro diagrama, se vuelven más costosos por las siguientes razones:

- Deben ser evaluados por un ingeniero que probablemente no esté familiarizado con el cambio de código problemático.
- Requieren más trabajo para que el autor del cambio de código recopile e investigue el cambio.
- Afectan negativamente a los demás, ya sean ingenieros en su trabajo o, en última instancia, al usuario final.

Para minimizar el costo de los errores, CI nos anima a usar *bucles de retroalimentación rápidos*.³ Cada vez que integramos un cambio de código (u otro) en un escenario de prueba y observamos los resultados, obtenemos un nuevo *Bucle de retroalimentación*. La retroalimentación puede tomar muchas formas; Los siguientes son algunos de los más comunes (en orden del más rápido al más lento):

- El bucle editar-compilar-depurar del desarrollo local
- Resultados de prueba automatizados a un autor de cambio de código en preenvío
- Un error de integración entre cambios en dos proyectos, detectado después de que ambos se enviaron y probaron juntos (es decir, después del envío)

³ Esto también se denomina a veces "desplazamiento a la izquierda en la prueba".

- Una incompatibilidad entre nuestro proyecto y una dependencia de microservicio ascendente, detectada por un evaluador de control de calidad en nuestro entorno de prueba, cuando el servicio ascendente implementa sus últimos cambios.
- Informes de errores de usuarios internos que están habilitados para una función antes que los usuarios externos
- Informes de errores o interrupciones por parte de usuarios externos o la prensa

Canarias—o implementar primero en un pequeño porcentaje de la producción— puede ayudar a minimizar los problemas que llegan a la producción, con un ciclo de retroalimentación inicial de subconjunto de producción que precede a toda la producción. Sin embargo, canarying también puede causar problemas, particularmente en relación con la compatibilidad entre implementaciones cuando se implementan varias versiones a la vez. Esto a veces se conoce como *versión sesgada*, un estado de un sistema distribuido en el que contiene múltiples versiones incompatibles de código, datos y/o configuración. Al igual que muchos problemas que analizamos en este libro, el sesgo de versión es otro ejemplo de un problema desafiante que puede surgir al intentar desarrollar y administrar software a lo largo del tiempo.

Experimentos y banderas de características son bucles de retroalimentación extremadamente poderosos. Reducen el riesgo de implementación al aislar los cambios dentro de los componentes modulares que se pueden alternar dinámicamente en la producción. Confiar en gran medida en la protección de banderas de características es un paradigma común para la entrega continua, que exploramos más a fondo en [capítulo 24](#).

Comentarios accesibles y procesables

También es importante que los comentarios de CI sean ampliamente accesibles. Además de nuestra cultura abierta en torno a la visibilidad del código, sentimos lo mismo acerca de nuestros informes de prueba. Tenemos un sistema de informes de prueba unificado en el que cualquiera puede buscar fácilmente una compilación o ejecución de prueba, incluidos todos los registros (excluyendo la información de identificación personal [PII] del usuario), ya sea para la ejecución local de un ingeniero individual o en una compilación de desarrollo o etapa automatizada.

Junto con los registros, nuestro sistema de informes de prueba proporciona un historial detallado de cuándo comenzaron a fallar los objetivos de compilación o prueba, incluidas auditorías de dónde se cortó la compilación en cada ejecución, dónde se ejecutó y por quién. También tenemos un sistema para la clasificación de escamas, que utiliza estadísticas para clasificar las escamas a nivel de todo Google, por lo que los ingenieros no necesitan resolver esto por sí mismos para determinar si su cambio rompió la prueba de otro proyecto (si la prueba es escamoso: probablemente no).

La visibilidad del historial de pruebas permite a los ingenieros compartir y colaborar en los comentarios, un requisito esencial para que equipos dispares diagnostiquen y aprendan de las fallas de integración entre sus sistemas. De manera similar, los errores (por ejemplo, tickets o problemas) en Google están abiertos con un historial de comentarios completo para que todos los vean y aprendan (con la excepción, nuevamente, de la PII del cliente).

Finalmente, cualquier retroalimentación de las pruebas de IC no solo debe ser accesible sino procesable, fácil de usar para encontrar y solucionar problemas. Veremos un ejemplo de cómo mejorar el usuario-

retroalimentación hostil en nuestro estudio de caso más adelante en este capítulo. Al mejorar la legibilidad de los resultados de las pruebas, automatiza la comprensión de los comentarios.

Automatización

Es bien sabido que la **automatización de tareas relacionadas con el desarrollo ahorra recursos de ingeniería** a la larga. Intuitivamente, debido a que automatizamos los procesos definiéndolos como código, la revisión por pares cuando se verifican los cambios reducirá la probabilidad de error. Por supuesto, los procesos automatizados, como cualquier otro software, tendrán errores; pero cuando se implementan de manera efectiva, aún son más rápidos, más fáciles y más confiables que si los ingenieros los intentaran manualmente.

CI, específicamente, automatiza la *construcción y lanzamiento* de procesos, con un Continuous Build y Continuous Delivery. Se aplican pruebas continuas en todo momento, que veremos en la siguiente sección.

Construcción continua

El *Construcción continua*(CB) integra los últimos cambios de código en la cabeza⁴ y ejecuta una compilación y prueba automatizadas. Debido a que el CB ejecuta pruebas además de compilar código, "romper la compilación" o "fallar en la compilación" incluye romper las pruebas así como romper la compilación.

Después de enviar un cambio, el OC debe ejecutar todas las pruebas pertinentes. Si un cambio pasa todas las pruebas, el CB lo marca como aprobado o "verde", como se muestra a menudo en las interfaces de usuario (IU). Este proceso introduce efectivamente dos versiones diferentes de head en el repositorio:*cabeza verdadera*, o el último cambio que se confirmó, y *cabeza verde*, o el último cambio que el OC haya verificado. Los ingenieros pueden sincronizar cualquier versión en su desarrollo local. Es común sincronizar con Green Head para trabajar con un entorno estable, verificado por el CB, mientras se codifica un cambio, pero tiene un proceso que requiere que los cambios se sincronicen con True Head antes del envío.

Entrega continua

El primer paso en Continuous Delivery (CD; discutido más completamente en [capítulo 24](#)) es *lanzamiento de automatización*, que ensambla continuamente el código y la configuración más recientes desde Head hasta los candidatos de versión. En Google, la mayoría de los equipos cortan estos en verde, en lugar de la cabeza real.

⁴Cabeza es el último código versionado en nuestro monorepo. En otros flujos de trabajo, esto también se conoce como *Maestro*, *línea principal*, *rama maestra*. En consecuencia, la integración en la cabeza también se conoce como *desarrollo basado en troncos*.

Candidato de lanzamiento(RC): una unidad cohesiva y desplegable creada por un proceso automatizado,⁵ ensamblado de código, configuración y otras dependencias que han pasado la compilación continua.

Tenga en cuenta que incluimos la configuración en las versiones candidatas; esto es extremadamente importante, aunque puede variar ligeramente entre entornos a medida que se promociona la candidata. No recomendamos necesariamente que compile la configuración en sus archivos binarios — en realidad, recomendaríamos una configuración dinámica, como experimentos o indicadores de características, para muchos escenarios.⁶

Más bien, estamos diciendo que cualquier configuración estática que *hace/have* debe promocionarse como parte de la versión candidata para que pueda someterse a pruebas junto con su código correspondiente. Recuerde, un gran porcentaje de los errores de producción son causados por problemas de configuración "tontos", por lo que es tan importante probar su configuración como lo es su código (y probarlo junto con él). *con el mismo código que lo usará*). El sesgo de la versión a menudo se ve atrapado en este proceso de promoción de candidatos de lanzamiento. Esto supone, por supuesto, que su configuración estática está en el control de versiones; en Google, la configuración estática está en el control de versiones junto con el código y, por lo tanto, pasa por el mismo proceso de revisión del código.

Entonces definimos CD de la siguiente manera:

Entrega continua(CD): un ensamblaje continuo de candidatos de lanzamiento, seguido de la promoción y prueba de esos candidatos a lo largo de una serie de entornos que a veces alcanzan la producción y otras veces no.

El proceso de promoción y despliegue a menudo depende del equipo. Mostraremos cómo nuestro estudio de caso navegó este proceso.

Para los equipos de Google que desean recibir comentarios continuos de los nuevos cambios en la producción (p. ej., Implementación continua), por lo general, no es factible impulsar de forma continua archivos binarios completos, que a menudo son bastante grandes, en verde. Por eso, haciendo *unselectivo*La implementación continua, a través de experimentos o indicadores de características, es una estrategia común.⁷

A medida que un RC avanza a través de los entornos, sus artefactos (p. ej., binarios, contenedores) idealmente no deberían volver a compilarse ni reconstruirse. El uso de contenedores como Docker ayuda a reforzar la coherencia de un RC entre entornos, desde el desarrollo local en adelante. De manera similar, usar herramientas de orquestación como Kubernetes (o en nuestro caso, generalmente **borg**), ayuda a reforzar la coherencia entre las implementaciones. Al hacer cumplir la consistencia de

⁵ En Google, la automatización de lanzamientos es administrada por un sistema separado de TAP. no nos centraremos en cómo la automatización ensambla los RC, pero si está interesado, lo remitimos a *Ingeniería de confiabilidad del sitio*(O'Reilly) en el que se analiza en detalle nuestra tecnología de automatización de lanzamientos (un sistema llamado Rapid).

⁶ CD con experimentos y banderas de características se discute más adelante en [capítulo 24](#).

⁷ Las llamamos "colisiones en el aire" porque la probabilidad de que ocurra es extremadamente baja; sin embargo, cuando esto sucede, los resultados pueden ser bastante sorprendentes.

nuestro lanzamiento e implementación entre entornos, logramos pruebas más tempranas de mayor fidelidad y menos sorpresas en la producción.

Pruebas continuas

Veamos cómo encajan CB y CD cuando aplicamos pruebas continuas (CT) a un cambio de código a lo largo de su vida útil, como se muestra [Figura 23-2](#).

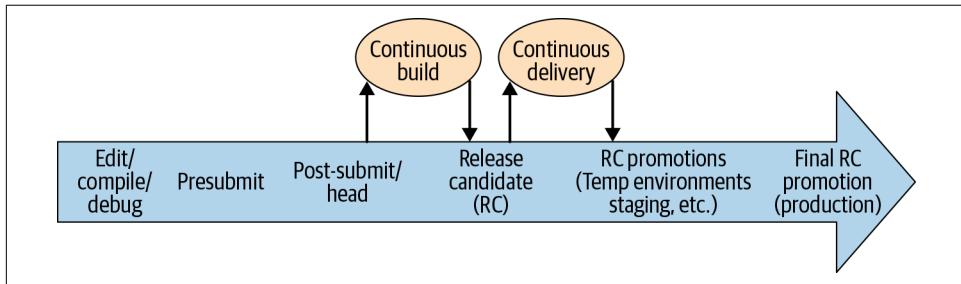


Figura 23-2. Vida de un cambio de código con CB y CD

La flecha hacia la derecha muestra la progresión de un solo cambio de código desde el desarrollo local hasta la producción. Una vez más, uno de nuestros objetivos clave en CI es determinar qué Probar cuándo en esta progresión. Más adelante en este capítulo, presentaremos las diferentes fases de prueba y brindaremos algunas consideraciones sobre qué probar en el envío previo versus el posterior al envío, y en el RC y más allá. Mostraremos que, a medida que nos desplazamos hacia la derecha, el cambio de código se somete a pruebas automatizadas progresivamente de mayor alcance.

¿Por qué presuponer no es suficiente?

Con el objetivo de detectar cambios problemáticos lo antes posible y la capacidad de ejecutar pruebas automatizadas en el envío previo, es posible que se pregunte: ¿por qué no ejecutar todas las pruebas en el envío previo?

La razón principal es que es demasiado caro. La productividad de los ingenieros es extremadamente valiosa, y esperar mucho tiempo para ejecutar cada prueba durante el envío del código puede ser muy perjudicial. Además, al eliminar la restricción de que los envíos previos sean exhaustivos, se pueden lograr muchas mejoras en la eficiencia si las pruebas se aprueban con mucha más frecuencia de lo que fallan. Por ejemplo, las pruebas que se ejecutan pueden restringirse a ciertos ámbitos o seleccionarse en función de un modelo que predice su probabilidad de detectar una falla.

Del mismo modo, es costoso para los ingenieros ser bloqueados en la presentación previa por fallas que surjan de inestabilidad o descamación que no tiene nada que ver con su cambio de código.

Otra razón es que durante el tiempo que ejecutamos las pruebas de envío previo para confirmar que un cambio es seguro, el repositorio subyacente podría haber cambiado de una manera incompatible con los cambios que se están probando. Es decir, es posible que dos cambios que toquen archivos completamente diferentes provoquen que una prueba falle. Llamamos a esto una colisión en el aire,

y aunque generalmente es raro, ocurre la mayoría de los días a nuestra escala. Los sistemas de CI para repositorios o proyectos más pequeños pueden evitar este problema al serializar los envíos para que no haya diferencia entre lo que está a punto de ingresar y lo que acaba de ingresar.

Preenvio versus postenvío

Entonces, ¿qué pruebas *deberían* ser ejecutado en presumit? Nuestra regla general es: solo los rápidos y confiables. Puede aceptar alguna pérdida de cobertura en el envío previo, pero eso significa que debe detectar cualquier problema que se le escape en el envío posterior y aceptar una cierta cantidad de reversiones. Después del envío, puede aceptar tiempos más largos y cierta inestabilidad, siempre que tenga los mecanismos adecuados para manejarlo.



Mostraremos cómo TAP y nuestro estudio de caso manejan la gestión de fallas en “CI en Google” en la página 493.

No queremos desperdiciar la valiosa productividad de los ingenieros esperando demasiado por pruebas lentas o por demasiadas pruebas; por lo general, limitamos las pruebas previas al envío solo a aquellas para el proyecto en el que se está produciendo el cambio. También realizamos pruebas al mismo tiempo, por lo que también hay que considerar una decisión de recursos. Finalmente, no queremos ejecutar pruebas poco confiables en el envío previo, porque el costo de tener muchos ingenieros afectados por ellas, depurando el mismo problema que no está relacionado con su cambio de código, es demasiado alto.

La mayoría de los equipos de Google ejecutan sus pequeñas pruebas (como las pruebas unitarias) en el envío previo⁸—estos son los más obvios para ejecutar, ya que tienden a ser los más rápidos y confiables. La pregunta más interesante es si y cómo ejecutar pruebas de mayor alcance en el envío previo, y esto varía según el equipo. Para los equipos que quieren ejecutarlos, las pruebas herméticas son un enfoque comprobado para reducir su inestabilidad inherente. Otra opción es permitir que las pruebas de gran alcance no sean confiables en el envío previo, pero deshabilitarlas agresivamente cuando comiencen a fallar.

Lanzamiento de pruebas de candidatos

Después de que un cambio de código haya pasado el CB (esto podría tomar varios ciclos si hubiera fallas), pronto encontrará el CD y se incluirá en un candidato de versión pendiente.

A medida que CD construye RC, ejecutará pruebas más grandes contra todo el candidato. Probamos una versión candidata promocionándola a través de una serie de entornos de prueba y probándola en cada implementación. Esto puede incluir una combinación de entornos temporales de espacio aislado.

⁸ Cada equipo de Google configura un subconjunto de las pruebas de su proyecto para que se ejecuten antes del envío (en lugar de después del envío). En realidad, nuestra compilación continua en realidad optimiza algunas pruebas previas al envío para que se guarden para el envío posterior, detrás de escena. Hablaremos más sobre esto más adelante en este capítulo.

ronments y entornos de prueba compartidos, como dev o staging. También es común incluir algunas pruebas de control de calidad manuales del RC en entornos compartidos.

Hay varias razones por las que es importante ejecutar un conjunto de pruebas completo y automatizado en un RC, incluso si es el mismo conjunto que CB acaba de ejecutar en el código posterior al envío (suponiendo que el CD se corte en verde):

Como un control de cordura

Verificamos dos veces que no sucedió nada extraño cuando se cortó el código y se volvió a compilar en el RC.

Para auditabilidad

Si un ingeniero desea verificar los resultados de las pruebas de un RC, están fácilmente disponibles y asociados con el RC, por lo que no necesitan buscar en los registros de CB para encontrarlos.

Para permitir picos de cereza

Si aplica una corrección selectiva a un RC, su código fuente ahora se ha desviado del último corte probado por el CB.

Para empujones de emergencia

En ese caso, el CD puede cortar desde la cabeza verdadera y ejecutar el conjunto mínimo de pruebas necesarias para sentirse seguro acerca de un empujón de emergencia, sin esperar a que pase el CB completo.

Pruebas de producción

Nuestro proceso de prueba continuo y automatizado llega hasta el entorno de implementación final: la producción. Deberíamos ejecutar el mismo conjunto de pruebas contra la producción (a veces llamado *sondeadores*) que hicimos contra la versión candidata anteriormente para verificar: 1) el estado de funcionamiento de la producción, de acuerdo con nuestras pruebas, y 2) la relevancia de nuestras pruebas, de acuerdo con la producción.

Las pruebas continuas en cada paso de la progresión de la aplicación, cada una con sus propias compensaciones, sirven como un recordatorio del valor de un enfoque de "defensa en profundidad" para detectar errores: no es solo un poco de tecnología o política. en el que confiamos para la calidad y la estabilidad, son muchos enfoques de prueba combinados.

CI está alertando

tito inviernos

Al igual que con los sistemas de producción que funcionan de manera responsable, el mantenimiento sostenible de los sistemas de software también requiere un monitoreo automatizado continuo. Así como usamos un sistema de monitoreo y alerta para comprender cómo los sistemas de producción responden al cambio, CI revela cómo nuestro software responde a los cambios en su entorno. Mientras que el monitoreo de la producción se basa en alertas pasivas y sondas activas de los sistemas en funcionamiento, CI

utiliza pruebas unitarias y de integración para detectar cambios en el software antes de implementarlo. Hacer comparaciones entre estos dos dominios nos permite aplicar el conocimiento de uno a otro.

Tanto la CI como las alertas tienen el mismo propósito general en el flujo de trabajo del desarrollador: identificar problemas tan rápido como sea razonablemente posible. CI enfatiza el lado inicial del flujo de trabajo del desarrollador y detecta los problemas al revelar fallas en las pruebas. Las alertas se centran en el final tardío del mismo flujo de trabajo y detectan problemas al monitorear métricas e informar cuando superan algún umbral. Ambas son formas de "identificar problemas automáticamente, tan pronto como sea posible".

Un sistema de alertas bien administrado ayuda a garantizar que se cumplan sus objetivos de nivel de servicio (SLO). Un buen sistema de CI ayuda a garantizar que su compilación esté en buen estado: el código se compila, las pruebas pasan y puede implementar una nueva versión si es necesario. Las políticas de mejores prácticas en ambos espacios se centran mucho en ideas de fidelidad y alertas procesables: las pruebas deben fallar solo cuando se viola el invariante subyacente importante, en lugar de porque la prueba es frágil o escamosa. Una prueba escamosa que falla cada pocas ejecuciones de CI es un problema tan grande como una alerta falsa que se dispara cada pocos minutos y genera una página para la guardia. Si no es procesable, no debería estar alertando. Si en realidad no está violando las invariantes del SUT, no debería ser una prueba fallida.

CI y alertas comparten un marco conceptual subyacente. Por ejemplo, existe una relación similar entre señales localizadas (pruebas unitarias, monitoreo de estadísticas aisladas/alertas basadas en causas) y señales de dependencia cruzada (pruebas de integración y liberación, sondeo de caja negra). Los indicadores de mayor fidelidad de si un sistema agregado está funcionando son las señales de un extremo a otro, pero esa fidelidad se paga con descamación, costos de recursos crecientes y dificultad para depurar las causas raíz.

De manera similar, vemos una conexión subyacente en los modos de falla para ambos dominios. Las alertas frágiles basadas en causas se activan al cruzar un umbral arbitrario (por ejemplo, reintentos en la última hora), sin que necesariamente exista una conexión fundamental entre ese umbral y el estado del sistema visto por un usuario final. Las pruebas frágiles fallan cuando se viola un requisito de prueba arbitrario o un invariante, sin que necesariamente haya una conexión fundamental entre ese invariante y la corrección del software que se está probando. En la mayoría de los casos, estos son fáciles de escribir y potencialmente útiles para depurar un problema mayor. En ambos casos, son indicadores aproximados de la salud/corrección general, y no logran captar el comportamiento holístico. Si no tiene una sonda fácil de extremo a extremo, pero facilita la recopilación de algunas estadísticas agregadas, los equipos escribirán alertas de umbral basadas en estadísticas arbitrarias. Si no tiene una manera de alto nivel de decir: "Falla la prueba si la imagen decodificada no es aproximadamente igual a esta imagen decodificada", los equipos crearán pruebas que afirman que los flujos de bytes son idénticos.

Las alertas basadas en causas y las pruebas frágiles aún pueden tener valor; simplemente no son la forma ideal de identificar problemas potenciales en un escenario de alerta. En el caso de una falla real, puede ser útil tener más detalles de depuración disponibles. Cuando los SRE están depurando una interrupción, puede ser útil tener información del tipo: "Hace una hora, los usuarios comenzaron a experimentar más solicitudes fallidas. Más o menos al mismo tiempo, el número de reintentos

comenzó a hacer tic tac. Comencemos a investigar allí". Del mismo modo, las pruebas frágiles aún pueden proporcionar información de depuración adicional: "La tubería de procesamiento de imágenes comenzó a escupir basura. Una de las pruebas unitarias sugiere que estamos obteniendo diferentes bytes del compresor JPEG. Comencemos a investigar allí".

Aunque el monitoreo y las alertas se consideran parte del dominio de gestión de la producción/SRE, donde se entiende bien la idea de los "presupuestos de errores",⁹ CI proviene de una perspectiva que todavía tiende a centrarse en absolutos. Enmarcar la CI como el "cambio a la izquierda" de las alertas comienza a sugerir formas de razonar sobre esas políticas y proponer mejores prácticas:

- Tener una tasa verde del 100 % en CI, al igual que tener un tiempo de actividad del 100 % para un servicio de producción, es terriblemente costoso. Si eso es *Realmente* su objetivo, uno de los mayores problemas va a ser una condición de carrera entre la prueba y la presentación.
- Tratar cada alerta como una causa igual de alarma generalmente no es el enfoque correcto. Si se activa una alerta en producción pero el servicio no se ve realmente afectado, silenciar la alerta es la opción correcta. Lo mismo es cierto para las pruebas fallidas: hasta que nuestros sistemas de CI aprendan a decir: "Se sabe que esta prueba está fallando por razones irrelevantes", probablemente deberíamos ser más liberales al aceptar cambios que deshabiliten una prueba fallida. No todas las fallas en las pruebas son indicativas de problemas de producción futuros.
- Las políticas que dicen: "Nadie puede comprometerse si nuestros últimos resultados de CI no son verdes" probablemente estén equivocadas. Si CI informa un problema, tales fallas definitivamente deben ser *investigadoras* de dejar que las personas cometan o agraven el problema. Pero si la causa raíz se comprende bien y claramente no afectaría la producción, bloquear las confirmaciones no es razonable.

Esta perspectiva de "CI está alertando" es nueva, y todavía estamos descubriendo cómo establecer paralelismos completos. Dado lo mucho que está en juego, no sorprende que SRE haya pensado mucho en las mejores prácticas relacionadas con el monitoreo y las alertas, mientras que CI se ha visto más como una característica de lujo.¹⁰ Durante los próximos años, la tarea en la ingeniería de software será ver dónde se puede reconceptualizar la práctica de SRE existente en un contexto de IC para ayudar a reformular el panorama de las pruebas y la IC, y quizás dónde las mejores prácticas en las pruebas pueden ayudar a aclarar los objetivos y las políticas sobre el monitoreo, y alertando.

⁹ Apuntar al 100 % de tiempo de actividad es el objetivo equivocado. Elija algo como 99,9% o 99,999% como negocio o producto compense, defina y controle su tiempo de actividad real, y utilice ese "presupuesto" como información sobre la agresividad con la que está dispuesto a impulsar lanzamientos riesgosos.

¹⁰ Creemos que la IC es realmente fundamental para el ecosistema de ingeniería de software: algo imprescindible, no un lujo. Pero eso es todavía no se entiende universalmente.

Desafíos de IC

Hemos discutido algunas de las mejores prácticas establecidas en CI y hemos presentado algunos de los desafíos involucrados, como la posible interrupción de la productividad del ingeniero de pruebas inestables, lentas, conflictivas o simplemente demasiadas en el momento de la presentación previa. Algunos desafíos adicionales comunes al implementar IC incluyen los siguientes:

- *Optimización previa al envío*, incluido *cualquier* prueba para que se ejecuten en el momento de preenviar datos los posibles problemas que ya hemos descrito, y cómo para ejecutarlos.
- *Hallazgo culpable y aislamiento de fallas*: ¿Qué código u otro cambio causó el problema y en qué sistema sucedió? “Integrar microservicios ascendentes” es un enfoque para el aislamiento de fallas en una arquitectura distribuida, cuando desea averiguar si un problema se originó en sus propios servidores o en un back-end. En este enfoque, organiza combinaciones de sus servidores estables junto con nuevos servidores de microservicios ascendentes. (Por lo tanto, está integrando los últimos cambios de los microservicios en sus pruebas). Este enfoque puede ser particularmente desafiante debido al sesgo de la versión: no solo estos entornos a menudo son incompatibles, sino que también es probable que encuentre falsos positivos: problemas que ocurren en una combinación por etapas particular que en realidad no se detectaría en la producción.
- *Restricciones de recursos*: Las pruebas necesitan recursos para ejecutarse y las pruebas grandes pueden ser muy costosas. Además, el costo de la infraestructura para insertar pruebas automatizadas a lo largo del proceso puede ser considerable.

También está el desafío de *gestión de fallas*—qué hacer cuando las pruebas fallan. Aunque los problemas más pequeños generalmente se pueden solucionar rápidamente, muchos de nuestros equipos encuentran que es extremadamente difícil tener un conjunto de pruebas consistentemente ecológico cuando se trata de grandes pruebas de un extremo a otro. Se vuelven inherentemente rotos o escamosos y son difíciles de depurar; debe haber un mecanismo para deshabilitarlos temporalmente y realizar un seguimiento de ellos para que la liberación pueda continuar. Una técnica común en Google es usar “listas calientes” de errores archivadas por un ingeniero de guardia o de lanzamiento y enviadas al equipo apropiado. Aún mejor es cuando estos errores se pueden generar y archivar automáticamente; algunos de nuestros productos más grandes, como Google Web Server (GWS) y Google Assistant, hacen esto. Estas listas calientes deben seleccionarse para asegurarse de que cualquier error que bloquee la versión se solucione de inmediato. Los bloqueadores de no liberación también deben corregirse; son menos urgentes, pero también debe priorizarse para que el conjunto de pruebas siga siendo útil y no sea simplemente una pila creciente de pruebas antiguas deshabilitadas. A menudo, los problemas detectados por las fallas de las pruebas de extremo a extremo son en realidad con las pruebas en lugar del código.

Las pruebas escamosas plantean otro problema para este proceso. Erosionan la confianza de manera similar a una prueba fallida, pero encontrar un cambio para revertir a menudo es más difícil porque la falla no sucederá todo el tiempo. Algunos equipos confían en una herramienta para eliminar tales pruebas irregulares de la presentación previa temporalmente mientras se investiga y corrige la irregularidad. Esto mantiene alta la confianza y permite más tiempo para solucionar el problema.

Prueba de inestabilidades otro desafío significativo que ya hemos visto en el contexto de las presunciones. Una táctica para lidiar con esto es permitir que se ejecuten múltiples intentos de la prueba. Esta es una opción de configuración de prueba común que usan los equipos. Además, dentro del código de prueba, se pueden introducir reintentos en varios puntos de especificidad.

Otro enfoque que ayuda con la inestabilidad de la prueba (y otros desafíos de IC) es la prueba hermética, que veremos en la siguiente sección.

Pruebas herméticas

Debido a que hablar con un backend en vivo no es confiable, a menudo usamos**backends herméticos** para pruebas de mayor alcance. Esto es particularmente útil cuando queremos ejecutar estas pruebas antes del envío, cuando la estabilidad es de suma importancia. En[Capítulo 11](#), introdujimos el concepto de pruebas herméticas:

Pruebas herméticas: las pruebas se ejecutan en un entorno de prueba (es decir, servidores de aplicaciones y recursos) que es completamente autónomo (es decir, sin dependencias externas como backends de producción).

Las pruebas herméticas tienen dos propiedades importantes: mayor determinismo (es decir, estabilidad) y aislamiento. Los servidores herméticos aún son propensos a algunas fuentes de no determinismo, como la hora del sistema, la generación de números aleatorios y las condiciones de carrera. Sin embargo, lo que se incluye en la prueba no cambia en función de las dependencias externas, por lo que cuando ejecuta una prueba dos veces con la misma aplicación y el mismo código de prueba, debe obtener los mismos resultados. Si falla una prueba hermética, sabe que se debe a un cambio en el código o las pruebas de su aplicación (con una advertencia menor: también pueden fallar debido a una reestructuración de su entorno de prueba hermético, pero esto no debería cambiar muy a menudo). Por esta razón, cuando los sistemas de CI vuelven a ejecutar las pruebas horas o días después para proporcionar señales adicionales, la hermeticidad hace que las fallas en las pruebas sean más fáciles de reducir.

La otra propiedad importante, el aislamiento, significa que los problemas en la producción no deberían afectar estas pruebas. Por lo general, también ejecutamos estas pruebas en la misma máquina, por lo que no tenemos que preocuparnos por los problemas de conectividad de la red. Lo contrario también es válido: los problemas causados por la ejecución de pruebas herméticas no deberían afectar la producción.

El éxito de la prueba hermética no debería depender del usuario que ejecuta la prueba. Esto permite que las personas reproduzcan las pruebas ejecutadas por el sistema CI y permite que las personas (p. ej., desarrolladores de bibliotecas) ejecuten pruebas propiedad de otros equipos.

Un tipo de respaldo hermético es falso. Como se discutió en[Capítulo 13](#), estos pueden ser más baratos que ejecutar un backend real, pero requieren trabajo para mantenerlos y tienen una fidelidad limitada.

La opción más limpia para lograr una prueba de integración digna de pre-envío es con una configuración totalmente hermética, es decir, iniciar toda la pila en un espacio aislado.¹¹—y Google proporciona configuraciones de sandbox listas para usar para componentes populares, como bases de datos, para que sea más fácil. Esto es más factible para aplicaciones más pequeñas con menos componentes, pero hay excepciones en Google, incluso una (de DisplayAds) que inicia alrededor de cuatrocientos servidores desde cero en cada envío previo y de forma continua en el envío posterior. Sin embargo, desde el momento en que se creó el sistema, la grabación/reproducción se ha convertido en un paradigma más popular para sistemas más grandes y tiende a ser más económico que iniciar una gran pila de espacio aislado.

Grabar/reproducir (ver[capítulo 14](#)) registran respuestas de back-end en vivo, las almacenan en caché y las reproducen en un entorno de prueba hermético. Grabar/reproducir es una herramienta poderosa para reducir la inestabilidad de las pruebas, pero una desventaja es que conduce a pruebas frágiles: es difícil lograr un equilibrio entre lo siguiente:

Falsos positivos

La prueba pasa cuando probablemente no debería haberlo hecho porque estamos presionando demasiado el caché y omitimos problemas que surgirían al capturar una nueva respuesta.

falsos negativos

La prueba falla cuando probablemente no debería haberlo hecho porque estamos accediendo muy poco a la memoria caché. Esto requiere que las respuestas se actualicen, lo que puede llevar mucho tiempo y provocar fallas en las pruebas que deben corregirse, muchas de las cuales pueden no ser problemas reales. Este proceso suele bloquear el envío, lo que no es lo ideal.

Idealmente, un sistema de grabación/reproducción debería detectar solo los cambios problemáticos y el cachemis solo cuando una solicitud ha cambiado de manera significativa. En el caso de que ese cambio cause un problema, el autor del cambio de código volvería a ejecutar la prueba con una respuesta actualizada, vería que la prueba sigue fallando y, por lo tanto, recibiría una alerta sobre el problema. En la práctica, saber cuándo una solicitud ha cambiado de manera significativa puede ser increíblemente difícil en un sistema grande y en constante cambio.

El asistente hermético de Google

El Asistente de Google proporciona un marco para que los ingenieros ejecuten pruebas de extremo a extremo, incluido un dispositivo de prueba con funcionalidad para configurar consultas, especificar si simular en un teléfono o un dispositivo doméstico inteligente y validar las respuestas a lo largo de un intercambio. con el Asistente de Google.

¹¹ En la práctica, a menudo es difícil hacer una completamente entorno de prueba de espacio aislado, pero la estabilidad deseada puede lograrse minimizando las dependencias externas.

Una de sus mayores historias de éxito fue hacer que su conjunto de pruebas fuera completamente hermético en el momento de la presentación previa. Cuando el equipo solía ejecutar pruebas no herméticas en el momento de la presentación previa, las pruebas fallaban de forma rutinaria. En algunos días, el equipo vería más de 50 cambios de código omitir e ignorar los resultados de la prueba. Al pasar a hermetic, el equipo redujo el tiempo de ejecución en un factor de 14, prácticamente sin descamación. Todavía ve fallas, pero esas fallas tienden a ser bastante fáciles de encontrar y revertir.

Ahora que las pruebas no herméticas se han enviado para enviarlas después, se acumulan fallas allí. La depuración de pruebas fallidas de un extremo a otro sigue siendo difícil, y algunos equipos ni siquiera tienen tiempo para intentarlo, por lo que simplemente las deshabilitan. Eso es mejor que detener todo el desarrollo para todos, pero puede provocar fallas en la producción.

Uno de los desafíos actuales del equipo es continuar perfeccionando sus mecanismos de almacenamiento en caché para que el envío previo pueda detectar más tipos de problemas que se han descubierto solo después del envío en el pasado, sin introducir demasiada fragilidad.

Otro es cómo realizar pruebas previas al envío para el Asistente descentralizado dado que los componentes se están trasladando a sus propios microservicios. Debido a que el Asistente tiene una pila grande y compleja, el costo de ejecutar una pila hermética en la presentación previa, en términos de trabajo de ingeniería, coordinación y recursos, sería muy alto.

Finalmente, el equipo está aprovechando esta descentralización en una nueva e inteligente estrategia de aislamiento de fallas posterior al envío. para cada uno de los *norte* microservicios dentro del Asistente, el equipo ejecutará un entorno posterior al envío que contenga el microservicio creado en cabeza, junto con las versiones de producción (o cercanas) del otrono_{re-1} servicios, para aislar los problemas del servidor recién construido. Esta configuración sería normalmente $\alpha(norte)$ costo de facilitar, pero el equipo aprovecha una característica genial llamada *intercambio en caliente* para reducir este costo a $\alpha(norte)$. Esencialmente, el intercambio en caliente permite una solicitud para indicar a un servidor que "intercambie" la dirección de un backend para llamar en lugar de la habitual. tan solo *norte* los servidores deben ejecutarse, uno para cada uno de los microservicios cortados en la cabeza, y pueden reutilizar el mismo conjunto de backends de producción intercambiados en cada uno de estos *norte* "ambientes."

Como hemos visto en esta sección, las pruebas herméticas pueden reducir la inestabilidad en las pruebas de mayor alcance y ayudar a aislar las fallas, abordando dos de los desafíos importantes de CI que identificamos en la sección anterior. Sin embargo, los backends herméticos también pueden ser más costosos porque usan más recursos y son más lentos de configurar. Muchos equipos usan combinaciones de backends herméticos y en vivo en sus entornos de prueba.

IC en Google

Ahora veamos con más detalle cómo se implementa la CI en Google. Primero, veremos nuestra compilación continua global, TAP, utilizada por la gran mayoría de los equipos de Google, y cómo habilita algunas de las prácticas y aborda algunos de los desafíos que analizamos en la sección anterior. También veremos una aplicación, Google Takeout, y cómo una transformación de CI la ayudó a escalar como plataforma y como servicio.

TAP: Construcción continua global de Google

Adán doblador

Ejecutamos una compilación continua masiva, llamada Plataforma de automatización de pruebas (TAP), de todo nuestro código base. Es responsable de ejecutar la mayoría de nuestras pruebas automatizadas. Como consecuencia directa de nuestro uso de un monorepo, TAP es la puerta de entrada para casi todos los cambios en Google. Cada día es responsable de manejar más de 50.000 cambios únicos ejecutando más de cuatro mil millones de casos de prueba individuales.

TAP es el corazón palpitante de la infraestructura de desarrollo de Google. Conceptualmente, el proceso es muy simple. Cuando un ingeniero intenta enviar código, TAP ejecuta las pruebas asociadas e informa sobre el éxito o el fracaso. Si las pruebas pasan, se permite el cambio en la base de código.

Optimización previa al envío

Para detectar problemas de manera rápida y consistente, es importante asegurarse de que las pruebas se ejecuten con cada cambio. Sin un CB, la ejecución de las pruebas generalmente se deja a discreción del ingeniero individual, y eso a menudo lleva a que unos pocos ingenieros motivados intenten ejecutar todas las pruebas y mantenerse al día con las fallas.

Como se discutió anteriormente, esperar mucho tiempo para ejecutar cada prueba en el envío previo puede ser muy perjudicial y, en algunos casos, llevar horas. Para minimizar el tiempo de espera, el enfoque CB de Google permite que los cambios potencialmente importantes aterricen en el repositorio (recuerde que se vuelven visibles de inmediato para el resto de la empresa). Todo lo que pedimos es que cada equipo cree un subconjunto rápido de pruebas, a menudo las pruebas unitarias de un proyecto, que se pueden ejecutar antes de que se envíe un cambio (por lo general, antes de que se envíe para la revisión del código): el envío previo. Empíricamente, un cambio que pasa el preenvío tiene una probabilidad muy alta (más del 95 %) de pasar el resto de las pruebas y, con optimismo, permitimos que se integre para que otros ingenieros puedan comenzar a usarlo.

Después de enviar un cambio, usamos TAP para ejecutar de forma asíncrona todas las pruebas potencialmente afectadas, incluidas las pruebas más grandes y más lentas.

Cuando un cambio hace que una prueba falle en TAP, es imperativo que el cambio se solucione rápidamente para evitar bloquear a otros ingenieros. Hemos establecido una norma cultural que desaconseja encarecidamente realizar cualquier trabajo nuevo además de las pruebas fallidas conocidas, aunque las pruebas irregulares lo dificultan. Por lo tanto, cuando se confirma un cambio que rompe la compilación de un equipo en TAP, ese cambio puede evitar que el equipo avance o cree una nueva versión. Como resultado, es imperativo tratar las roturas rápidamente.

Para hacer frente a tales roturas, cada equipo tiene un "Policía de construcción". La responsabilidad del Build Cop es hacer que todas las pruebas pasen en su proyecto en particular, sin importar quién las rompa. Cuando se notifica a un Build Cop de una prueba fallida en su proyecto, deja lo que sea que esté haciendo y corrige la compilación. Por lo general, esto se hace identificando el cambio ofensivo y determinando si es necesario revertirlo (la solución preferida) o si se puede corregir en el futuro (una propuesta más riesgosa).

En la práctica, la compensación de permitir que se realicen cambios antes de verificar todas las pruebas realmente ha valido la pena; el tiempo de espera promedio para enviar un cambio es de alrededor de 11 minutos, a menudo se ejecuta en segundo plano. Junto con la disciplina de Build Cop, podemos detectar y abordar de manera eficiente las roturas detectadas por pruebas de ejecución más largas con una cantidad mínima de interrupciones.

Hallazgo culpable

Uno de los problemas que enfrentamos con los grandes conjuntos de pruebas en Google es encontrar el cambio específico que rompió una prueba. Conceptualmente, esto debería ser realmente fácil: tome un cambio, ejecute las pruebas, si alguna prueba falla, marque el cambio como malo. Desafortunadamente, debido a la prevalencia de fallas y los problemas ocasionales con la infraestructura de prueba en sí, tener la confianza de que una falla es real no es fácil. Para complicar más las cosas, TAP debe evaluar tantos cambios por día (más de uno por segundo) que ya no puede ejecutar todas las pruebas en cada cambio. En su lugar, recurre a agrupar los cambios relacionados por lotes, lo que reduce el número total de pruebas únicas que se ejecutarán. Aunque este enfoque puede hacer que la ejecución de las pruebas sea más rápida, puede ocultar qué cambio en el lote provocó la interrupción de una prueba.

Para acelerar la identificación de fallas, utilizamos dos enfoques diferentes. En primer lugar, TAP divide automáticamente un lote fallido en cambios individuales y vuelve a ejecutar las pruebas contra cada cambio de forma aislada. A veces, este proceso puede tardar un tiempo en converger en una falla, por lo que, además, hemos creado herramientas de búsqueda de culpables que un desarrollador individual puede usar para realizar búsquedas binarias a través de un lote de cambios e identificar cuál es el probable culpable.

Gestión de fallas

Después de aislar un cambio importante, es importante corregirlo lo más rápido posible. La presencia de pruebas fallidas puede comenzar rápidamente a erosionar la confianza en el conjunto de pruebas. Como se mencionó anteriormente, reparar una compilación rota es responsabilidad del Build Cop. La herramienta más efectiva que tiene Build Cop es el *Retroceder*.

Revertir un cambio suele ser la ruta más rápida y segura para corregir una compilación porque restaura rápidamente el sistema a un buen estado conocido.¹² De hecho, TAP se ha actualizado recientemente para revertir automáticamente los cambios cuando tiene mucha confianza de que son los culpables.

Las reversiones rápidas funcionan de la mano con un conjunto de pruebas para garantizar una productividad continua. Las pruebas nos dan confianza para cambiar, las reversiones nos dan confianza para deshacer. Sin pruebas, las reversiones no se pueden realizar de manera segura. Sin reversiones, las pruebas rotas no se pueden reparar rápidamente, lo que reduce la confianza en el sistema.

12 ¡Cualquier cambio en el código base de Google se puede deshacer con dos clics!

Restricciones de recursos

Aunque los ingenieros pueden ejecutar las pruebas localmente, la mayoría de las ejecuciones de pruebas se realizan en un sistema distribuido de compilación y prueba llamado *Fragua*. Forge permite a los ingenieros ejecutar sus compilaciones y pruebas en nuestros centros de datos, lo que maximiza el paralelismo. A nuestra escala, los recursos necesarios para ejecutar todas las pruebas ejecutadas bajo demanda por los ingenieros y todas las pruebas que se ejecutan como parte del proceso de CB son enormes. Incluso dada la cantidad de recursos informáticos que tenemos, los sistemas como Forge y TAP tienen recursos limitados. Para sortear estas restricciones, los ingenieros que trabajan en TAP han ideado algunas formas inteligentes de determinar qué pruebas se deben ejecutar en qué momento para garantizar que se gaste la cantidad mínima de recursos para validar un cambio determinado.

El mecanismo principal para determinar qué pruebas deben ejecutarse es un análisis del gráfico de dependencia descendente para cada cambio. Las herramientas de creación distribuida de Google, Forge y Blaze, mantienen una versión casi en tiempo real del gráfico de dependencia global y la ponen a disposición de TAP. Como resultado, TAP puede determinar rápidamente qué pruebas son posteriores a cualquier cambio y ejecutar el conjunto mínimo para asegurarse de que el cambio sea seguro.

Otro factor que influye en el uso de TAP es la velocidad de ejecución de las pruebas. TAP a menudo puede ejecutar cambios con menos pruebas antes que aquellos con más pruebas. Este sesgo alienta a los ingenieros a escribir cambios pequeños y enfocados. La diferencia en el tiempo de espera entre un cambio que activa 100 pruebas y otro que activa 1000 puede ser de decenas de minutos en un día ajetreado. Los ingenieros que quieren pasar menos tiempo esperando terminan haciendo cambios más pequeños y específicos, lo que es una victoria para todos.

Estudio de caso de IC: Google Takeout

Google Takeout comenzó como un producto de copia de seguridad y descarga de datos en 2011. Sus fundadores fueron pioneros en la idea de la "liberación de datos": que los usuarios deberían poder llevar fácilmente sus datos con ellos, en un formato utilizable, dondequiera que vayan. Comenzaron integrando Takeout con un puñado de productos de Google, produciendo archivos de fotos de usuarios, listas de contactos, etc., para descargarlos a petición suya. Sin embargo, Take-out no se quedó pequeño por mucho tiempo y creció como plataforma y como servicio para una amplia variedad de productos de Google. Como veremos, la CI efectiva es fundamental para mantener saludable cualquier proyecto grande, pero es especialmente crítica cuando las aplicaciones crecen rápidamente.

Escenario n.º 1: implementaciones de desarrollo rotas continuamente

Problema: A medida que Takeout se ganó la reputación de ser una poderosa herramienta de descarga, archivo y obtención de datos en todo Google, otros equipos de la empresa comenzaron a recurrir a ella y solicitaron API para que sus propias aplicaciones también pudieran proporcionar funciones de copia de seguridad y descarga, incluido Google Drive. (las descargas de carpetas se realizan mediante Takeout) y Gmail (para vistas previas de archivos ZIP). En general, Takeout pasó de ser el backend solo del producto Google Takeout original a proporcionar API para al menos otros 10 productos de Google, ofreciendo una amplia gama de funciones.

El equipo decidió implementar cada una de las nuevas API como una instancia personalizada, utilizando los mismos archivos binarios originales de Takeout pero configurándolos para que funcionen de manera un poco diferente. Por ejemplo, el entorno para las descargas masivas de Drive tiene la flota más grande, la mayor cuota reservada para obtener archivos de la API de Drive y cierta lógica de autenticación personalizada para permitir que los usuarios que no hayan iniciado sesión descarguen carpetas públicas.

En poco tiempo, Takeout enfrentó "problemas de bandera". Los indicadores agregados para una de las instancias romperían las otras, y sus implementaciones se interrumpirían cuando los servidores no pudieran iniciarse debido a incompatibilidades de configuración. Más allá de la configuración de funciones, también había seguridad y configuración de ACL. Por ejemplo, el servicio de descarga de Drive del consumidor no debería tener acceso a las claves que cifran las exportaciones de Gmail empresarial. La configuración rápidamente se volvió complicada y condujo a roturas casi nocturnas.

Se hicieron algunos esfuerzos para desenredar y modularizar la configuración, pero el mayor problema que esto expuso fue que cuando un ingeniero de Takeout quería hacer un cambio de código, no era práctico probar manualmente que cada servidor se iniciaba en cada configuración. No se enteraron de los errores de configuración hasta la implementación del día siguiente. Hubo pruebas unitarias que se ejecutaron antes y después del envío (por TAP), pero no fueron suficientes para detectar este tipo de problemas.

Lo que hizo el equipo. El equipo creó minientornos temporales de espacio aislado para cada una de estas instancias que se ejecutaron en el envío previo y probaron que todos los servidores estaban en buen estado al inicio. Ejecutar los entornos temporales en preenvío evitó que el 95 % de los servidores dañados tuvieran una mala configuración y redujo las fallas de implementación nocturna en un 50 %.

Aunque estas nuevas pruebas de preenvío en espacio aislado redujeron drásticamente las fallas de implementación, no las eliminaron por completo. En particular, las pruebas de extremo a extremo de Takeout seguían interrumpiendo con frecuencia la implementación, y estas pruebas eran difíciles de ejecutar antes del envío (porque usan cuentas de prueba, que todavía se comportan como cuentas reales en algunos aspectos y están sujetas a las mismas garantías de seguridad y privacidad). Rediseñarlos para que fueran aptos para la presumisión habría sido una empresa demasiado grande.

Si el equipo no pudo ejecutar pruebas de un extremo a otro en el envío previo, ¿cuándo podría ejecutarlas? Quería obtener los resultados de las pruebas de un extremo a otro más rápidamente que la implementación de desarrollo del día siguiente y decidió que cada dos horas era un buen punto de partida. Pero el equipo no quería hacer una implementación de desarrollo completa con tanta frecuencia; esto generaría gastos generales e interrumpiría los procesos de ejecución prolongada que los ingenieros estaban probando en el desarrollo. La creación de un nuevo entorno de prueba compartido para estas pruebas también parecía demasiado sobrecarga para aprovisionar recursos, además de encontrar culpables (es decir, encontrar la implementación que condujo a una falla) podría implicar un trabajo manual no deseado.

Por lo tanto, el equipo reutilizó los entornos de prueba previos al envío y los extendió fácilmente a un nuevo entorno posterior al envío. A diferencia del envío previo, el envío posterior cumplió con las medidas de seguridad para usar las cuentas de prueba (por un lado, porque el código tiene

sido aprobado), por lo que las pruebas de extremo a extremo podrían ejecutarse allí. El CI posterior al envío se ejecuta cada dos horas, obtiene el código y la configuración más recientes de Green Head, crea un RC y ejecuta el mismo conjunto de pruebas de extremo a extremo que ya se ejecuta en desarrollo.

Lección aprendida. Los bucles de retroalimentación más rápidos evitan problemas en las implementaciones de desarrollo:

- Mover las pruebas para diferentes productos Takeout de "después de la implementación nocturna" a preenvío evitó que el 95 % de los servidores dañados tuvieran una mala configuración y redujo las fallas de implementación nocturna en un 50 %.
- Si bien las pruebas de un extremo a otro no se pudieron mover hasta el punto de envío previo, se movieron de "después del despliegue nocturno" a "después del envío dentro de las dos horas". Esto redujo efectivamente el "conjunto culpable" en 12 veces.

Escenario #2: Registros de prueba indescifrables

Problema: A medida que Takeout incorporaba más productos de Google, se convirtió en una plataforma madura que permitía a los equipos de productos insertar complementos, con código de obtención de datos específico del producto, directamente en el binario de Takeout. Por ejemplo, el complemento de Google Photos sabe cómo obtener fotos, metadatos de álbumes y similares. Takeout se expandió de su "puñado" original de productos para integrarse ahora con más de 90.

Las pruebas de extremo a extremo de Takeout volcaron sus fallas en un registro y este enfoque no se amplió a 90 complementos de productos. A medida que se integraban más productos, se introdujeron más fallas. A pesar de que el equipo estaba ejecutando las pruebas antes y con mayor frecuencia con la adición del CI posterior al envío, aún se acumulaban múltiples fallas en el interior y era fácil pasárselas por alto. Revisar estos registros se convirtió en una pérdida de tiempo frustrante, y las pruebas casi siempre fallaban.

Lo que hizo el equipo. El equipo refactorizó las pruebas en una suite dinámica basada en la configuración (usando un [corredor de prueba parametrizado](#)) que informó los resultados en una interfaz de usuario más amigable, mostrando claramente los resultados de las pruebas individuales en verde o rojo: no más excavaciones en los registros. También hicieron que las fallas fueran mucho más fáciles de depurar, sobre todo, al mostrar información de fallas, con enlaces a registros, directamente en el mensaje de error. Por ejemplo, si Takeout no pudo obtener un archivo de Gmail, la prueba construiría dinámicamente un enlace que buscaría el ID de ese archivo en los registros de Takeout y lo incluiría en el mensaje de falla de la prueba. Esto automatizó gran parte del proceso de depuración para los ingenieros de complementos de productos y requirió menos asistencia del equipo de Takeout para enviarles registros, como se demuestra en [Figura 23-3](#).

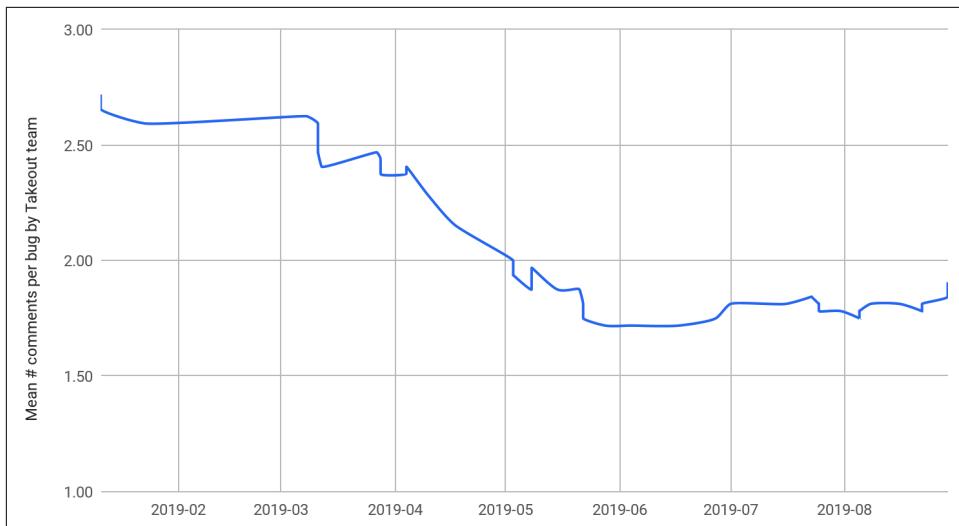


Figura 23-3. La participación del equipo en la depuración de fallas del cliente

Lección aprendida.La retroalimentación accesible y procesable de CI reduce las fallas en las pruebas y mejora la productividad. Estas iniciativas redujeron la participación del equipo de Takeout en la depuración de errores de prueba del cliente (complemento del producto) en un 35 %.

Escenario #3: Depuración de "todo Google"

Problema:Un efecto secundario interesante de Takeout CI que el equipo no anticipó fue que, debido a que verificó la salida de unos 90 productos extraños para el usuario final, en forma de archivo, básicamente estaban probando "todo Google". y detectar problemas que no tenían nada que ver con Takeout. Esto fue algo bueno: Takeout pudo ayudar a contribuir a la calidad de los productos de Google en general. Sin embargo, esto introdujo un problema para sus procesos de CI: necesitaban un mejor aislamiento de fallas para poder determinar qué problemas estaban en su compilación (cuáles eran la minoría) y cuáles se encontraban en microservicios débilmente acoplados detrás de las API del producto que llamaron.

Lo que hizo el equipo.La solución del equipo fue ejecutar exactamente el mismo conjunto de pruebas continuamente contra la producción como ya lo hizo en su CI posterior al envío. Esto fue barato de implementar y permitió al equipo aislar qué fallas eran nuevas en su construcción y cuáles estaban en producción; por ejemplo, el resultado del lanzamiento de un microservicio en otro lugar "en Google".

Lección aprendida.Ejecutar el mismo conjunto de pruebas contra prod y un CI posterior al envío (con binarios recién construidos, pero los mismos backends en vivo) es una forma económica de aislar fallas.

Desafío restante.En el futuro, la carga de probar "todo Google" (obviamente, esto es una exageración, ya que la mayoría de los problemas de los productos son detectados por sus respectivos equipos) crece a medida que Takeout se integra con más productos y estos se vuelven más complejos. Las comparaciones manuales entre este CI y prod son un uso costoso del tiempo del Build Cop.

Mejora futura.Esto presenta una oportunidad interesante para probar pruebas herméticas con grabación/reproducción en el CI posterior al envío de Takeout. En teoría, esto eliminaría las fallas de las API de productos back-end que aparecen en el CI de Takeout, lo que haría que la suite fuera más estable y eficaz para detectar fallas en las últimas dos horas de cambios de Takeout, que es su propósito previsto.

Escenario #4: Manteniéndolo verde

Problema:Como la plataforma admitía más complementos de productos, cada uno de los cuales incluía pruebas de un extremo a otro, estas pruebas fallaban y los conjuntos de pruebas de un extremo a otro casi siempre se rompián. No todas las fallas pudieron corregirse de inmediato. Muchos se debieron a errores en los complementos binarios del producto, sobre los cuales el equipo de Takeout no tenía control. Y algunas fallas importaron más que otras: los errores de baja prioridad y los errores en el código de prueba no necesitaban bloquear un lanzamiento, mientras que los errores de mayor prioridad sí lo hicieron. El equipo podría deshabilitar fácilmente las pruebas comentándolas, pero eso haría que las fallas fueran demasiado fáciles de olvidar.

Una fuente común de fallas: las pruebas fallaban cuando los complementos del producto implementaban una función. Por ejemplo, una función de obtención de listas de reproducción para el complemento de YouTube podría habilitarse para probarse en desarrollo durante algunos meses antes de habilitarse en producción. Las pruebas de Takeout solo conocían un resultado para verificar, por lo que a menudo resultaba en la necesidad de deshabilitar la prueba en entornos particulares y curarla manualmente a medida que se implementaba la función.

Lo que hizo el equipo.El equipo ideó una forma estratégica de deshabilitar las pruebas fallidas al etiquetarlas con un error asociado y enviarlo al equipo responsable (generalmente un equipo de complemento de producto). Cuando una prueba fallida se etiquetaba con un error, el marco de prueba del equipo suprimía su falla. Esto permitió que el conjunto de pruebas se mantuviera verde y aún brindara la confianza de que todo lo demás, además de los problemas conocidos, estaba pasando, como se ilustra enFigura 23-4.

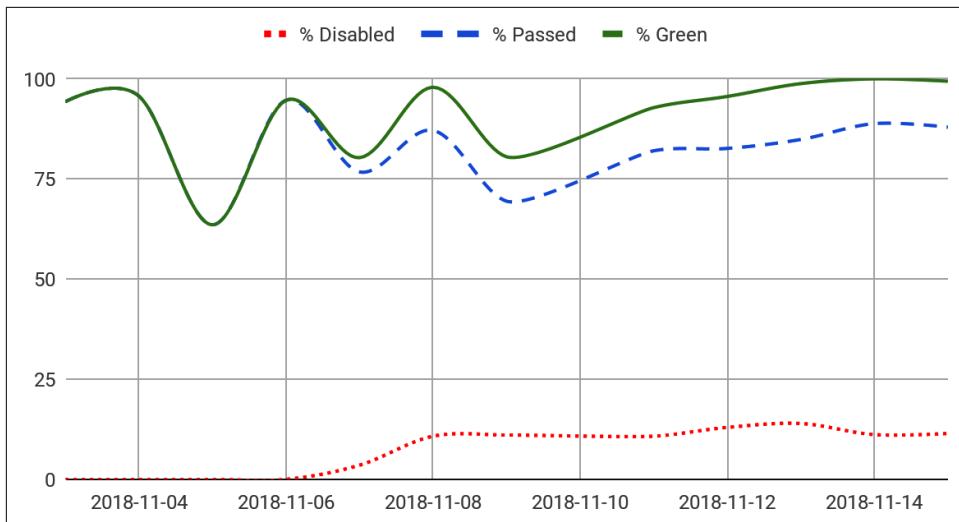


Figura 23-4. Lograr el verdor a través de la desactivación de pruebas (responsable)

Para el problema de implementación, el equipo agregó la capacidad para que los ingenieros de complementos especificuen el nombre de un indicador de función, o ID de un cambio de código, que habilitaba una función particular junto con la salida esperada con y sin la función. Las pruebas se equiparon para consultar el entorno de prueba para determinar si la función dada estaba habilitada allí y verificaron el resultado esperado en consecuencia.

Cuando las etiquetas de error de las pruebas deshabilitadas comenzaron a acumularse y no se actualizaron, el equipo automatizó su limpieza. Las pruebas ahora verificarían si un error se cerró consultando la API de nuestro sistema de errores. Si una prueba etiquetada que falla realmente pasó y estuvo pasando por más tiempo que el límite de tiempo configurado, la prueba solicitaría limpiar la etiqueta (y marcar el error como corregido, si aún no lo estaba). Hubo una excepción para esta estrategia: las pruebas escamosas. Para estos, el equipo permitiría que una prueba se etiquetara como escamosa, y el sistema no generaría una falla etiquetada como "escamosa" para la limpieza si pasaba.

Estos cambios crearon un conjunto de pruebas mayormente automantenable, como se ilustra en Figura 23-5.

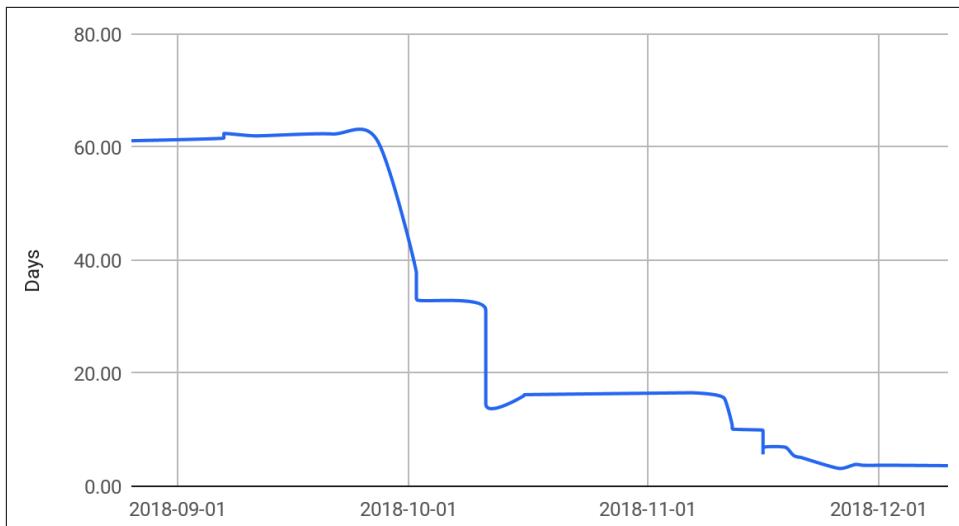


Figura 23-5. Tiempo medio para cerrar el error, después de enviar la corrección

Lecciones aprendidas. Deshabilitar las pruebas fallidas que no se pueden solucionar de inmediato es un enfoque práctico para mantener su suite verde, lo que le da la confianza de que está al tanto de todas las fallas de las pruebas. Además, la automatización del mantenimiento del conjunto de pruebas, incluida la gestión de la implementación y la actualización de errores de seguimiento para pruebas reparadas, mantiene el conjunto limpio y evita deudas técnicas. En el lenguaje de DevOps, podríamos llamar a la métrica en Figura 23-5 MTTCU: tiempo medio para limpiar.

Mejora futura. Automatizar el archivo y el etiquetado de errores sería un próximo paso útil. Este sigue siendo un proceso manual y engorroso. Como se mencionó anteriormente, algunos de nuestros equipos más grandes ya lo hacen.

Más desafíos. Los escenarios que hemos descrito están lejos de ser los únicos desafíos de CI que enfrenta Takeout, y aún hay más problemas por resolver. Por ejemplo, mencionamos la dificultad de aislar fallas de los servicios aguas arriba en “Desafíos de CI” en la página 490. Este es un problema que Takeout todavía enfrenta con fallas raras que se originan con servicios ascendentes, como cuando una actualización de seguridad en la infraestructura de transmisión utilizada por la API de “Descargas de carpetas de Drive” de Takeout rompió el descifrado de archivos cuando se implementó en producción. Los servicios upstream se preparan y prueban por sí mismos, pero no existe una forma simple de verificar automáticamente con CI si son compatibles con Takeout una vez que se lanzan a la producción. Una solución inicial involucró la creación de un entorno de CI de “prueba ascendente” para probar los binarios de producción de Takeout con las versiones preparadas de sus dependencias ascendentes. Sin embargo, esto resultó difícil de mantener, con problemas de compatibilidad adicionales entre las versiones de ensayo y producción.

Pero no puedo permitirme CI

Puede que estés pensando que todo está muy bien, pero no tienes ni el tiempo ni el dinero para construir nada de esto. Ciertamente reconocemos que Google podría tener más recursos para implementar CI que la típica startup. Sin embargo, muchos de nuestros productos han crecido tan rápido que tampoco tuvieron tiempo de desarrollar un sistema de CI (al menos no uno adecuado).

En sus propios productos y organizaciones, trate de pensar en el costo que ya está pagando por los problemas descubiertos y tratados en la producción. Estos afectan negativamente al usuario final o cliente, por supuesto, pero también afectan al equipo. La lucha contra incendios de producción frecuente es estresante y desmoralizadora. Si bien la construcción de sistemas de CI es costosa, no es necesariamente un costo nuevo sino un costo trasladado a la izquierda a una etapa anterior y más preferible, que reduce la incidencia y, por lo tanto, el costo de los problemas que ocurren demasiado a la derecha. CI conduce a un producto más estable y una cultura de desarrollo más feliz en la que los ingenieros se sienten más seguros de que "el sistema" detectará los problemas y pueden concentrarse más en las características y menos en solucionarlos.

Conclusión

Aunque hemos descrito nuestros procesos de CI y parte de cómo los hemos automatizado, nada de esto quiere decir que hayamos desarrollado sistemas de CI perfectos. Después de todo, un sistema de CI en sí mismo es solo software y nunca está completo y debe ajustarse para satisfacer las demandas cambiantes de la aplicación y los ingenieros a los que debe servir. Hemos tratado de ilustrar esto con la evolución del IC de Takeout y las futuras áreas de mejora que señalamos.

TL; DR

- Un sistema CI decide qué pruebas usar y cuándo.
- Los sistemas CI se vuelven progresivamente más necesarios a medida que su base de código envejece y crece en escala.
- CI debe optimizar pruebas más rápidas y confiables en el envío previo y pruebas más lentas y menos deterministas en el envío posterior.
- La retroalimentación accesible y accionable permite que un sistema de IC sea más eficiente.

Entrega continua

*Escrito por Radha Narayan, Bobbi Jones,
Sheri Shipe y David Owens
Editado por Lisa Carey*

Dada la rapidez y la imprevisibilidad con la que cambia el panorama tecnológico, la ventaja competitiva de cualquier producto radica en su capacidad para salir rápidamente al mercado. La velocidad de una organización es un factor crítico en su capacidad para competir con otros jugadores, mantener la calidad de los productos y servicios, o adaptarse a nuevas regulaciones. Esta velocidad se ve obstaculizada por el tiempo de implementación. La implementación no solo ocurre una vez en el lanzamiento inicial. Hay un dicho entre los educadores que ningún plan de lección sobrevive a su primer contacto con el alumnado. De la misma manera, ningún software es perfecto en el primer lanzamiento y la única garantía es que tendrá que actualizarlo. Rápidamente.

El ciclo de vida a largo plazo de un producto de software implica la exploración rápida de nuevas ideas, respuestas rápidas a cambios en el panorama o problemas de los usuarios, y permitir la velocidad del desarrollador a escala. De Eric Raymond *La catedral y el bazara* Eric Reis' *La puesta en marcha esbelta*, la clave del éxito a largo plazo de cualquier organización siempre ha estado en su capacidad para ejecutar las ideas y ponerlas en manos de los usuarios lo más rápido posible y reaccionar rápidamente a sus comentarios. Martin Fowler, en su libro *Entrega continua* (también conocido como CD), señala que "El mayor riesgo para cualquier esfuerzo de software es que terminas construyendo algo que no es útil. Cuanto antes y con mayor frecuencia obtenga el software en funcionamiento frente a usuarios reales, más rápido obtendrá comentarios para descubrir qué tan valioso es realmente".

El trabajo que permanece en progreso durante mucho tiempo antes de entregar valor al usuario es de alto riesgo y alto costo, e incluso puede ser una carga para la moral. En Google, nos esforzamos por lanzar pronto y con frecuencia, o "lanzar e iterar", para permitir que los equipos vean el impacto de su trabajo rápidamente y se adapten más rápido a un mercado cambiante. El valor del código no se percibe en el momento del envío, sino cuando las funciones están disponibles para los usuarios. Reduciendo el

El tiempo entre el "código completo" y la retroalimentación del usuario minimiza el costo del trabajo que está en progreso.

Obtienes resultados extraordinarios al darte cuenta de que el lanzamiento *nunca aterriza* pero que comienza un ciclo de aprendizaje en el que luego arreglas lo siguiente más importante, mides cómo fue, arreglas lo siguiente, etc., y es *nunca completo*.

— David Weekly, exgerente de productos de Google

En Google, las prácticas que describimos en este libro permiten que cientos (o en algunos casos miles) de ingenieros solucionen problemas rápidamente, trabajen de forma independiente en nuevas funciones sin preocuparse por el lanzamiento y comprendan la efectividad de las nuevas funciones a través de A/B. experimentación. Este capítulo se centra en las palancas clave de la innovación rápida, incluida la gestión del riesgo, la habilitación de la velocidad del desarrollador a escala y la comprensión de la compensación de costo y valor de cada función que lanza.

Modismos de entrega continua en Google

Un principio fundamental de la entrega continua (CD), así como de la metodología Agile, es que, con el tiempo, los lotes de cambios más pequeños dan como resultado una mayor calidad; en otras palabras, *más rápido es más seguro*. Esto puede parecer profundamente controvertido para los equipos a primera vista, especialmente si los requisitos previos para configurar el CD, por ejemplo, la integración continua (CI) y las pruebas, aún no se cumplen. Debido a que puede tomar un tiempo para que todos los equipos se den cuenta del ideal de CD, nos enfocamos en desarrollar varios aspectos que brinden valor de manera independiente en el camino hacia el objetivo final. Estos son algunos de estos:

Agilidad

Liberar con frecuencia y en lotes pequeños

Automatización

Reducir o elimine la sobrecarga repetitiva de lanzamientos frecuentes

Aislamiento

Esforzarse por una arquitectura modular para aislar los cambios y facilitar la resolución de problemas

Fiabilidad

Mida los indicadores clave de salud, como bloqueos o latencia, y siga mejorándolos.

Toma de decisiones basada en datos

Use pruebas A/B en métricas de salud para garantizar la calidad

Lanzamiento por fases

Implemente cambios para algunos usuarios antes de enviarlos a todos

Al principio, lanzar nuevas versiones de software con frecuencia puede parecer arriesgado. A medida que crece su base de usuarios, es posible que tema la reacción violenta de los usuarios enojados si hay errores que no detectó en las pruebas, y es posible que simplemente tenga demasiado código nuevo en

su producto para probarlo exhaustivamente. Pero aquí es precisamente donde el CD puede ayudar. Idealmente, hay tan pocos cambios entre una versión y la siguiente que la solución de problemas es trivial. En el límite, con CD, cada cambio pasa por la canalización de control de calidad y se implementa automáticamente en producción. A menudo, esta no es una realidad práctica para muchos equipos, por lo que a menudo hay un trabajo de cambio de cultura hacia CD como un paso intermedio, durante el cual los equipos pueden desarrollar su preparación para implementar en cualquier momento sin realmente hacerlo, aumentando su confianza para lanzar con mayor frecuencia en el futuro.

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

Cuando un equipo es pequeño, los cambios entran en un código base a un ritmo determinado. Hemos visto surgir un antipatrón a medida que un equipo crece con el tiempo o se divide en subequipos: un subequipo se bifurca en su código para evitar pisar los pies de alguien, pero luego lucha con la integración y la búsqueda del culpable. En Google, preferimos que los equipos continúen desarrollándose a la cabeza en la base de código compartida y configuren pruebas de CI, reverisiones automáticas y búsqueda de culpables para identificar problemas rápidamente. Esto se discute extensamente en [capítulo 23](#).

Una de nuestras bases de código, YouTube, es una gran aplicación de Python monolítica. El proceso de lanzamiento es laborioso, con Build Cops, gerentes de lanzamiento y otros voluntarios. Casi todos los lanzamientos tienen múltiples cambios y respins seleccionados cuidadosamente. También hay un ciclo de prueba de regresión manual de 50 horas ejecutado por un equipo de control de calidad remoto en cada versión. Cuando el costo operativo de una versión es tan alto, comienza a desarrollarse un ciclo en el que espera para lanzar su versión hasta que pueda probarla un poco más. Mientras tanto, alguien quiere agregar solo una característica más que está casi lista, y muy pronto tendrá un proceso de lanzamiento laborioso, propenso a errores y lento. Lo peor de todo es que los expertos que hicieron el lanzamiento la última vez están agotados y han dejado el equipo.

Si sus lanzamientos son costosos y, a veces, riesgosos, el *instinto* es ralentizar la cadencia de liberación y aumentar el período de estabilidad. Sin embargo, esto solo proporciona ganancias de estabilidad a corto plazo y, con el tiempo, reduce la velocidad y frustra a los equipos y usuarios. El *respuesta* es reducir costos, aumentar la disciplina y hacer que los riesgos sean más incrementales, pero es fundamental resistir las soluciones operativas obvias e invertir en cambios arquitectónicos a largo plazo. Las soluciones operativas obvias a este problema conducen a algunos enfoques tradicionales: volver a un modelo de planificación tradicional que deja poco espacio para el aprendizaje o la iteración, agregar más gobernanza y supervisión al proceso de desarrollo e implementar revisiones de riesgo o recompensar a los de bajo riesgo. (y a menudo de bajo valor) características.

Sin embargo, la inversión con el mejor rendimiento es migrar a una arquitectura de microservicios, que puede empoderar a un gran equipo de productos con la capacidad de seguir siendo ágil e innovador y, al mismo tiempo, reducir el riesgo. En algunos casos, en Google, la respuesta ha sido reescribir una aplicación desde cero en lugar de simplemente migrarla, estableciendo la modularidad deseada en la nueva arquitectura. Aunque cualquiera de estas opciones puede llevar meses y es probable que sea dolorosa a corto plazo, el valor obtenido en términos de costo operativo y simplicidad cognitiva se verá recompensado durante los años de vida útil de una aplicación.

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

Una clave para los lanzamientos continuos confiables es asegurarse de que los ingenieros "guarden la bandera" *"todos los cambios*. A medida que crece un producto, habrá múltiples funciones en varias etapas de desarrollo que coexistirán en un binario. La protección de banderas se puede utilizar para controlar la inclusión o la expresión del código de característica en el producto característica por característica y se puede expresar de manera diferente para las compilaciones de lanzamiento y desarrollo. Un indicador de función deshabilitado para una compilación debería permitir que las herramientas de compilación eliminen la función de la compilación si el idioma lo permite. Por ejemplo, una característica estable que ya se envió a los clientes podría estar habilitada para compilaciones de desarrollo y lanzamiento. Una función en desarrollo puede estar habilitada solo para desarrollo, protegiendo a los usuarios de una función sin terminar. El nuevo código de función vive en el binario junto con la antigua ruta de código; ambos pueden ejecutarse, pero el nuevo código está protegido por una bandera. Si el nuevo código funciona, puede eliminar la ruta de código anterior e iniciar la función por completo en una versión posterior. Si hay un problema, el valor de la bandera se puede actualizar independientemente de la versión binaria a través de una actualización de configuración dinámica.

En el viejo mundo de los lanzamientos binarios, teníamos que cronometrar los comunicados de prensa estrechamente con nuestros lanzamientos binarios. Teníamos que tener una implementación exitosa antes de que se pudiera emitir un comunicado de prensa sobre la nueva funcionalidad o una nueva característica. Esto significaba que la función estaría disponible antes de que se anunciara, y el riesgo de que se descubriera antes de tiempo era muy real.

Aquí es donde entra en juego la belleza de la guardia de la bandera. Si el nuevo código tiene una bandera, la bandera se puede actualizar para activar su función inmediatamente antes del comunicado de prensa, minimizando así el riesgo de filtrar una función. Tenga en cuenta que el código protegido por bandera no es un *Perfectored* de seguridad para características verdaderamente sensibles. El código aún se puede raspar y analizar si no está bien ofuscado, y no todas las funciones se pueden ocultar detrás de las banderas sin agregar mucha complejidad. Además, incluso los cambios en la configuración de las banderas deben implementarse con cuidado. Activar una marca para el 100 % de sus usuarios a la vez no es una buena idea, por lo que un servicio de configuración que gestione implementaciones de configuración seguras es una buena inversión. No obstante, el nivel de control y la capacidad de desvincular el destino de una característica particular del lanzamiento general del producto son palancas poderosas para la sustentabilidad a largo plazo de la aplicación.

Luchando por la agilidad: Configuración de un tren de lanzamiento

El binario de búsqueda de Google es el primero y el más antiguo. Grande y complicado, su base de código se puede vincular con el origen de Google: una búsqueda a través de nuestra base de código aún puede encontrar código escrito al menos desde 2003, a menudo antes. Cuando los teléfonos inteligentes comenzaron a despegar, función tras función móvil se calzaron en una bola de pelo de código escrito principalmente para la implementación del servidor. Aunque la experiencia de búsqueda se estaba volviendo más vibrante e interactiva, implementar una compilación viable se volvió cada vez más difícil. En un momento, publicamos el binario de búsqueda en producción solo una vez por semana, e incluso alcanzar ese objetivo era raro y, a menudo, se basaba en la suerte.

Cuando uno de nuestros autores colaboradores, Sheri Shipe, asumió el proyecto de aumentar nuestra velocidad de lanzamiento en la Búsqueda, cada ciclo de lanzamiento le tomaba días a un grupo de ingenieros para completarlo. Construyeron los datos binarios e integrados y luego comenzaron las pruebas. Cada error tuvo que clasificarse manualmente para asegurarse de que no afectara la calidad de la búsqueda, la experiencia del usuario (UX) y/o los ingresos. Este proceso era agotador y consumía mucho tiempo y no escalaba con el volumen o la tasa de cambio. Como resultado, un desarrollador nunca podría saber cuándo su función se lanzaría a producción. Esto hizo que los comunicados de prensa y los lanzamientos públicos fueran un desafío.

Los lanzamientos no ocurren en el vacío, y tener lanzamientos confiables hace que los factores dependientes sean más fáciles de sincronizar. En el transcurso de varios años, un grupo dedicado de ingenieros implementó un proceso de lanzamiento continuo, que simplificó todo lo relacionado con el envío de un binario de búsqueda al mundo. Automatizamos lo que pudimos, establecimos plazos para enviar funciones y simplificamos la integración de complementos y datos en el binario. Ahora podríamos lanzar constantemente un nuevo binario de búsqueda en producción cada dos días.

¿Cuáles fueron las compensaciones que hicimos para obtener previsibilidad en nuestro ciclo de lanzamiento? Se reducen a dos ideas principales que incorporamos al sistema.

Ningún binario es perfecto

la primera es que *ningún binario es perfecto*, especialmente para compilaciones que incorporan el trabajo de decenas o cientos de desarrolladores que desarrollan de forma independiente docenas de características principales. Si bien es imposible corregir todos los errores, constantemente debemos sopesar preguntas como: si una línea se ha movido dos píxeles hacia la izquierda, ¿afectará la visualización de un anuncio y los ingresos potenciales? ¿Qué pasa si la sombra de una caja se ha alterado ligeramente? ¿Será difícil para los usuarios con discapacidad visual leer el texto? Podría decirse que el resto de este libro trata de minimizar el conjunto de resultados no deseados para un lanzamiento, pero al final debemos admitir que el software es fundamentalmente complejo. No existe un binario perfecto: se deben tomar decisiones y compensaciones cada vez que se lanza un nuevo cambio a producción. Las métricas de indicadores clave de rendimiento con umbrales claros permiten

características para lanzar incluso si no son perfectas y también puede crear claridad en las decisiones de lanzamiento que de otro modo serían polémicas.

Un error involucró un dialecto raro que se habla en una sola isla de Filipinas. Si un usuario hiciera una pregunta de búsqueda en este dialecto, en lugar de una respuesta a su pregunta, obtendría una página web en blanco. Tuvimos que determinar si el costo de corregir este error valía la pena retrasar el lanzamiento de una nueva función importante.

Corrimos de oficina en oficina tratando de determinar cuántas personas realmente hablaban este idioma, si sucedía cada vez que un usuario buscaba en este idioma y si estas personas usaban Google de manera regular. Cada ingeniero de calidad con el que hablamos nos refirió a una persona de mayor rango. Finalmente, con los datos en la mano, le planteamos la pregunta al vicepresidente senior de Search. ¿Deberíamos retrasar un lanzamiento crítico para corregir un error que afectó solo a una isla filipina muy pequeña? Resulta que no importa cuán pequeña sea su isla, debe obtener resultados de búsqueda confiables y precisos: retrasamos el lanzamiento y solucionamos el error.

Cumpla con su fecha límite de publicación

La segunda idea es que *si llegas tarde al tren de liberación, se irá sin ti*. Hay algo que decir sobre el adagio, "los plazos son ciertos, la vida no". En algún momento de la línea de tiempo del lanzamiento, debe poner una apuesta en el suelo y rechazar a los desarrolladores y sus nuevas características. En términos generales, ninguna cantidad de súplicas o súplicas obtendrá una función en el lanzamiento de hoy después de que haya pasado la fecha límite.

Ahí está la excepción. La situación suele ser así. Es viernes por la noche y seis ingenieros de software irrumpen en pánico en el cubo del administrador de versiones. Tienen contrato con la NBA y terminaron la función hace unos momentos. Pero debe publicarse antes del gran partido de mañana. ¡El lanzamiento debe detenerse y debemos elegir la característica en el binario o estaremos incumpliendo el contrato! Un ingeniero de lanzamiento con ojos llorosos sacude la cabeza y dice que tomará cuatro horas cortar y probar un nuevo binario. Es el cumpleaños de su hijo y todavía tienen que recoger los globos.

Un mundo de lanzamientos regulares significa que si un desarrollador pierde el tren de lanzamiento, podrá tomar el próximo tren en cuestión de horas en lugar de días. Esto limita el pánico de los desarrolladores y mejora en gran medida el equilibrio entre el trabajo y la vida personal de los ingenieros de lanzamiento.

1 Recuerde la formulación de "presupuesto de error" de SRE: la perfección rara vez es la mejor meta. entender cuento el margen de error es aceptable y cuánto de ese presupuesto se ha gastado recientemente y utilícelo para ajustar la compensación entre velocidad y estabilidad.

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

Bloat es un efecto secundario desafortunado de la mayoría de los ciclos de vida de desarrollo de software, y cuanto más exitoso se vuelve un producto, más hinchado se vuelve su base de código. Una desventaja de un tren de lanzamiento rápido y eficiente es que este exceso a menudo se magnifica y puede manifestarse en desafíos para el equipo del producto e incluso para los usuarios. Especialmente si el software se entrega al cliente, como en el caso de las aplicaciones móviles, esto puede significar que el dispositivo del usuario paga el costo en términos de espacio, descarga y costos de datos, incluso para funciones que nunca usa, mientras que los desarrolladores pagan el costo de compilaciones más lentas, implementaciones complejas y errores raros. En esta sección, hablaremos sobre cómo las implementaciones dinámicas le permiten enviar solo lo que se usa, forzando las compensaciones necesarias entre el valor del usuario y el costo de la función.

Mientras que algunos productos están basados en la web y se ejecutan en la nube, muchos son aplicaciones de clientes que usan recursos compartidos en el dispositivo de un usuario: un teléfono o una tableta. Esta elección en sí misma muestra una compensación entre aplicaciones nativas que pueden tener un mayor rendimiento y resistencia a la conectividad irregular, pero también más difíciles de actualizar y más susceptibles a problemas a nivel de plataforma. Un argumento común en contra de la implementación continua y frecuente de aplicaciones nativas es que a los usuarios no les gustan las actualizaciones frecuentes y deben pagar el costo de los datos y la interrupción. Puede haber otros factores limitantes, como el acceso a una red o un límite en los reinicios necesarios para filtrar una actualización.

Aunque hay que hacer una compensación en términos de la frecuencia con la que se actualiza un producto, el objetivo es *hacer que estas elecciones sean intencionales*. Con un proceso de CD fluido y que funcione bien, ¿con qué frecuencia se realiza un lanzamiento viable? *creadose* puede separar de la frecuencia con la que un usuario *recibe* los cambios. Puede lograr el objetivo de poder realizar implementaciones semanales, diarias o por horas, sin hacerlo realmente, y debe elegir intencionalmente los procesos de publicación en el contexto de las necesidades específicas de sus usuarios y los objetivos organizacionales más amplios, y determinar la dotación de personal, y el modelo de herramientas que mejor respaldará la sostenibilidad a largo plazo de su producto.

Anteriormente en el capítulo, hablamos sobre mantener su código modular. Esto permite implementaciones dinámicas y configurables que permiten una mejor utilización de los recursos limitados, como el espacio en el dispositivo de un usuario. En ausencia de esta práctica, cada usuario debe recibir un código que nunca usará para respaldar traducciones que no necesita o arquitecturas destinadas a otros tipos de dispositivos. Las implementaciones dinámicas permiten que las aplicaciones mantengan tamaños pequeños mientras solo envían código a un dispositivo que aporta valor a sus usuarios, y los experimentos A/B permiten compensaciones intencionales entre el costo de una característica y su valor para los usuarios y su negocio.

Hay un costo inicial para configurar estos procesos, e identificar y eliminar las fricciones que mantienen la frecuencia de los lanzamientos por debajo de lo deseable es una tarea ardua.

proceso. Pero las ganancias a largo plazo en términos de gestión de riesgos, velocidad del desarrollador y permitir una innovación rápida son tan altas que estos costos iniciales valen la pena.

Desplazamiento a la izquierda: tomar decisiones basadas en datos antes

Si está creando para todos los usuarios, es posible que tenga clientes en pantallas inteligentes, parlantes o teléfonos y tabletas con Android e iOS, y su software puede ser lo suficientemente flexible como para permitir que los usuarios personalicen su experiencia. Incluso si está creando solo para dispositivos Android, la gran diversidad de los más de dos mil millones de dispositivos Android puede hacer que la posibilidad de calificar un lanzamiento sea abrumadora. Y con el ritmo de la innovación, para cuando alguien lea este capítulo, es posible que hayan florecido categorías completamente nuevas de dispositivos.

Uno de nuestros gerentes de lanzamiento compartió una sabiduría que cambió la situación cuando dijo que la diversidad de nuestro mercado de clientes no era un *problema*, pero un *hecho*. Después de aceptar eso, podríamos cambiar nuestro modelo de calificación de lanzamiento de las siguientes maneras:

- Si *integral* la prueba es prácticamente inviable, apunte a *representante* probando en su lugar.
- Los lanzamientos por etapas para aumentar lentamente los porcentajes de la base de usuarios permiten soluciones rápidas.
- Los lanzamientos A/B automatizados permiten obtener resultados estadísticamente significativos que prueban la calidad de un lanzamiento, sin que los humanos cansados tengan que mirar los paneles y tomar decisiones.

Cuando se trata de desarrollar para clientes de Android, las aplicaciones de Google usan pistas de prueba especializadas e implementaciones por etapas para un porcentaje cada vez mayor del tráfico de usuarios, monitoreando cuidadosamente los problemas en estos canales. Debido a que Play Store ofrece pistas de prueba ilimitadas, también podemos configurar un equipo de control de calidad en cada país en el que planeamos lanzar, lo que permite un cambio global de la noche a la mañana para probar funciones clave.

Un problema que notamos al realizar implementaciones en Android fue que podíamos esperar un cambio estadísticamente significativo en las métricas de los usuarios. *simplemente empujando una actualización*. Esto significaba que incluso si no realizábamos cambios en nuestro producto, impulsar una actualización podría afectar el comportamiento del dispositivo y del usuario de formas difíciles de predecir. Como resultado, aunque controlar la actualización de un pequeño porcentaje del tráfico de usuarios podría brindarnos buena información sobre fallas o problemas de estabilidad, nos dijo muy poco sobre si la versión más nueva de nuestra aplicación era mejor que la anterior.

Dan Siroker y Pete Koomen ya han discutido el valor de las pruebas A/B²sus características, pero en Google, algunas de nuestras aplicaciones más grandes también prueban A/B *su implementaciones*. Esto significa enviar dos versiones del producto: una que es la actualización deseada, siendo la línea de base un placebo (su versión anterior se envía nuevamente). A medida que las dos versiones se lanzan simultáneamente a una base lo suficientemente grande de usuarios similares, puede comparar una versión con la otra para ver si la última versión de su software es, de hecho, una mejora con respecto a la anterior. Con una base de usuarios lo suficientemente grande, debería poder obtener resultados estadísticamente significativos en días o incluso horas. Una canalización de métricas automatizadas puede permitir el lanzamiento más rápido posible al impulsar un lanzamiento a más tráfico tan pronto como haya suficientes datos para saber que las métricas de la medida de seguridad no se verán afectadas.

Obviamente, este método no se aplica a todas las aplicaciones y puede ser una gran sobrecarga cuando no tiene una base de usuarios lo suficientemente grande. En estos casos, la mejor práctica recomendada es apuntar a lanzamientos neutrales al cambio. Todas las funciones nuevas están protegidas por bandera, de modo que el único cambio que se prueba durante una implementación es la estabilidad de la implementación en sí.

Cambiando la Cultura del Equipo: Construyendo Disciplina en el Despliegue

Aunque "Always Be Deploying" ayuda a abordar varios problemas que afectan la velocidad del desarrollador, también existen ciertas prácticas que abordan problemas de escala. El equipo inicial que lanza un producto puede estar formado por menos de 10 personas, cada una de las cuales se turna en las responsabilidades de implementación y supervisión de la producción. Con el tiempo, su equipo puede crecer a cientos de personas, con subequipos responsables de funciones específicas. A medida que esto sucede y la organización se amplía, la cantidad de cambios en cada implementación y la cantidad de riesgo en cada intento de lanzamiento aumentan de forma superlineal. Cada lanzamiento contiene meses de sudor y lágrimas. Lograr que el lanzamiento sea exitoso se convierte en un esfuerzo intensivo de mano de obra. Los desarrolladores a menudo pueden verse atrapados tratando de decidir qué es peor: abandonar una versión que contiene una cuarta parte de nuevas funciones y correcciones de errores,

A escala, el aumento de la complejidad suele manifestarse como una mayor latencia de lanzamiento. Incluso si publica todos los días, un lanzamiento puede tardar una semana o más en implementarse completamente de manera segura, lo que lo deja una semana de retraso cuando intenta depurar cualquier problema. Aquí es donde "Always Be Deploying" puede devolver un proyecto de desarrollo a su forma efectiva. Los trenes de lanzamiento frecuentes permiten una divergencia mínima de una buena posición conocida, con la novedad de

² Dan Siroker y Pete Koomen, *Pruebas A/B: la forma más poderosa de convertir clics en clientes* (hoboken: Wiley, 2013).

cambios que ayuden a resolver problemas. Pero, ¿cómo puede un equipo garantizar que la complejidad inherente a un código base grande y en rápida expansión no obstaculice el progreso?

En Google Maps, adoptamos la perspectiva de que las características son muy importantes, pero muy pocas veces una característica es tan importante que se debe reservar un lanzamiento. Si los lanzamientos son frecuentes, el dolor que siente una función por perderse un lanzamiento es pequeño en comparación con el dolor que sienten todas las funciones nuevas en un lanzamiento por un retraso, y especialmente el dolor que los usuarios pueden sentir si se apresura una función que no está del todo lista ser incluido.

Una responsabilidad de la versión es proteger el producto de los desarrolladores.

Al hacer concesiones, la pasión y la urgencia que siente un desarrollador por lanzar una nueva función nunca pueden superar la experiencia del usuario con un producto existente. Esto significa que las nuevas funciones deben aislarse de otros componentes a través de interfaces con contratos sólidos, separación de preocupaciones, pruebas rigurosas, comunicación temprana y frecuente, y convenciones para la aceptación de nuevas funciones.

Conclusión

A lo largo de los años y en todos nuestros productos de software, hemos descubierto que, contrariamente a la intuición, más rápido es más seguro. La salud de su producto y la velocidad de desarrollo no se oponen entre sí, y los productos que se lanzan con más frecuencia y en lotes pequeños tienen mejores resultados de calidad. Se adaptan más rápido a los errores que encuentran en la naturaleza y a los cambios inesperados del mercado. No sólo eso, más rápido es *más económico*, porque tener un tren de lanzamiento frecuente y predecible lo obliga a reducir el costo de cada lanzamiento y hace que el costo de cualquier lanzamiento abandonado sea muy bajo.

Simplemente tener las estructuras en su lugar que *habilita* el despliegue continuo genera la mayor parte del valor, *incluso si en realidad no envía esos lanzamientos a los usuarios*. ¿Qué queremos decir? En realidad, no lanzamos una versión muy diferente de la Búsqueda, Maps o YouTube todos los días, pero para poder hacerlo se requiere un proceso de implementación continuo sólido y bien documentado, métricas precisas y en tiempo real sobre la satisfacción del usuario y el estado del producto, y un equipo coordinado con políticas claras sobre lo que entra o sale y por qué. En la práctica, hacer esto bien a menudo también requiere binarios que se puedan configurar en producción, configuración administrada como código (en el control de versiones) y una cadena de herramientas que permita medidas de seguridad como verificación de ejecución en seco, mecanismos de retroceso/retroceso y parches confiables. .

TL; DR

- *La velocidad es un deporte de equipo.*: El flujo de trabajo óptimo para un gran equipo que desarrolla código en colaboración requiere modularidad de arquitectura e integración casi continua.

- *Evaluar los cambios de forma aislada:* Señale cualquier característica para poder aislar los problemas antes de tiempo.
- *Haz de la realidad tu punto de referencia:* use un lanzamiento por etapas para abordar la diversidad de dispositivos y la amplitud de la base de usuarios. La calificación de versión en un entorno sintético que no es similar al entorno de producción puede dar lugar a sorpresas tardías.
- *Envíe solo lo que se usa:* Supervise el costo y el valor de cualquier función en la naturaleza para saber si sigue siendo relevante y ofrece suficiente valor para el usuario.
- *Desplazar a la izquierda:* Habilite una toma de decisiones más rápida y más basada en datos antes de todos los cambios a través de CI e implementación continua.
- *Más rápido es más seguro:* Realice envíos pronto y con frecuencia y en lotes pequeños para reducir el riesgo de cada lanzamiento y minimizar el tiempo de comercialización.

Cálculo como servicio

*Escrito por Onufry Wojtaszczyk
Editado por Lisa Carey*

No trato de entender las computadoras. Trato de entender los programas.

— Bárbara Liskov

Después de hacer el arduo trabajo de escribir código, necesita algo de hardware para ejecutarlo. Así, vas a comprar o alquilar ese hardware. Esto, en esencia, es *Cálculo como servicio* (CaaS), en el que "Computar" es la abreviatura de la potencia informática necesaria para ejecutar sus programas.

Este capítulo trata sobre cómo este concepto simple: solo dame el hardware para ejecutar mis cosas¹—se mapea en un sistema que sobrevivirá y escalará a medida que su organización evolucione y crezca. Es algo largo porque el tema es complejo, y se divide en cuatro secciones:

- “[“Domar el entorno informático” en la página 518](#) cubre cómo Google llegó a su solución para este problema y explica algunos de los conceptos clave de CaaS.
- “[“Escribir software para informática gestionada” en la página 523](#) muestra cómo una solución informática administrada afecta la forma en que los ingenieros escriben software. Creemos que el modelo de programación flexible/“ganado, no mascotas” ha sido fundamental para el éxito de Google en los últimos 15 años y es una herramienta importante en la caja de herramientas de un ingeniero de software.

1 Descargo de responsabilidad: para algunas aplicaciones, el “hardware para ejecutarlo” es el hardware de sus clientes (piense, por ejemplo, de un juego reactualizado que compró hace una década). Esto presenta desafíos muy diferentes que no cubrimos en este capítulo.

- “CaaS a lo largo del tiempo y la escala” en la página 530 profundiza en algunas lecciones que Google aprendió sobre cómo se desarrollan varias opciones sobre una arquitectura informática a medida que la organización crece y evoluciona.
- Finalmente, “Elección de un servicio informático” en la página 535 está dedicado principalmente a aquellos ingenieros que tomarán una decisión sobre qué servicio de cómputo utilizar en su organización.

Domar el entorno informático

Sistema Borg interno de Google² fue un precursor de muchas de las arquitecturas CaaS actuales (como Kubernetes o Mesos). Para comprender mejor cómo los aspectos particulares de dicho servicio responden a las necesidades de una organización en crecimiento y evolución, rastrearemos la evolución de Borg y los esfuerzos de los ingenieros de Google para domar el entorno informático.

Automatización del Trabajo

Imagina ser un estudiante en la universidad a principios de siglo. Si quisiera implementar un código nuevo y hermoso, usaría SFTP para el código en una de las máquinas en el laboratorio de computación de la universidad, SSH en la máquina, compilaría y ejecutaría el código. Esta es una solución tentadora por su simplicidad, pero se topa con problemas considerables con el tiempo y a escala. Sin embargo, debido a que es más o menos con lo que comienzan muchos proyectos, varias organizaciones terminan con procesos que son evoluciones un tanto optimizadas de este sistema, al menos para algunas tareas: la cantidad de máquinas crece (por lo que utiliza SFTP y SSH en muchas de ellas), pero la tecnología subyacente permanece. Por ejemplo, en 2002, Jeff Dean, uno de los ingenieros más importantes de Google, escribió lo siguiente sobre la ejecución de una tarea de procesamiento de datos automatizado como parte del proceso de lanzamiento:

[Ejecutar la tarea] es una pesadilla logística que consume mucho tiempo. Actualmente requiere obtener una lista de más de 50 máquinas, iniciar un proceso en cada una de estas más de 50 máquinas y monitorear su progreso en cada una de las más de 50 máquinas. No hay soporte para migrar automáticamente el cómputo a otra máquina si una de las máquinas muere, y el monitoreo del progreso de los trabajos se realiza de manera ad hoc [...] Además, dado que los procesos pueden interferir entre sí, hay un archivo de "registro" complicado e implementado por humanos para acelerar el uso de las máquinas, lo que da como resultado una programación menos que óptima y una mayor contienda por los escasos recursos de la máquina

Este fue un detonante temprano en los esfuerzos de Google para domar el entorno informático, lo que explica bien cómo la solución ingenua se vuelve inmantenible a mayor escala.

² Abhishek Verma, Luis Pedrosa, Madhukar R Korupolu, David Oppenheimer, Eric Tune y John Wilkes, “Gestión de clústeres a gran escala en Google con Borg” EuroSys, Artículo n.º: 18 (abril de 2015): 1-17.

Automatizaciones simples

Hay cosas simples que una organización puede hacer para mitigar parte del dolor. El proceso de implementar un binario en cada una de las más de 50 máquinas e iniciararlo allí se puede automatizar fácilmente a través de un script de shell y luego, si se trata de una solución reutilizable, a través de una pieza de código más robusta de una manera más sencilla. -para mantener el lenguaje que realizará la implementación en paralelo (especialmente porque es probable que el "50+" crezca con el tiempo).

Más interesante aún, el monitoreo de cada máquina también se puede automatizar.

Inicialmente, a la persona a cargo del proceso le gustaría saber (y poder intervenir) si algo salió mal con una de las réplicas. Esto significa exportar algunas métricas de monitoreo (como "el proceso está vivo" y "número de documentos procesados") del proceso, haciendo que escriba en un almacenamiento compartido o llamando a un servicio de monitoreo, donde pueden ver anomalías de un vistazo. Las soluciones de código abierto actuales en ese espacio son, por ejemplo, configurar un tablero en una herramienta de monitoreo como Graphana o Prometheus.

Si se detecta una anomalía, la estrategia habitual de mitigación es SSH en la máquina, eliminar el proceso (si aún está activo) y volver a iniciararlo. Esto es tedioso, posiblemente propenso a errores (asegúrese de conectarse a la máquina correcta y asegúrese de eliminar el proceso correcto), y podría automatizarse:

- En lugar de monitorear manualmente las fallas, se puede usar un agente en la máquina que detecta anomalías (como "el proceso no informó que está activo durante los últimos cinco minutos" o "el proceso no procesó ningún documento durante los últimos 10 minutos"), y mata el proceso si se detecta una anomalía.
- En lugar de iniciar sesión en la máquina para iniciar el proceso nuevamente después de la muerte, podría ser suficiente envolver toda la ejecución en un "mientras que cierto; corre y rompe; hecho" guión de concha.

El equivalente del mundo de la nube es establecer una política de reparación automática (para matar y volver a crear una máquina virtual o un contenedor después de que falle una verificación de estado).

Estas mejoras relativamente simples abordan una parte del problema de Jeff Dean descrito anteriormente, pero no todo; la regulación implementada por humanos y el cambio a una nueva máquina requieren soluciones más complejas.

Programación automatizada

El siguiente paso natural es automatizar la asignación de máquinas. Esto requiere el primer "servicio" real que eventualmente se convertirá en "Computación como servicio". Es decir, para automatizar la programación, necesitamos un servicio central que conozca la lista completa de máquinas disponibles y pueda, bajo demanda, elegir una cantidad de máquinas desocupadas e implementar automáticamente su binario en esas máquinas. Esto elimina la necesidad de una mano-

mantuvo el archivo de "registro", en lugar de delegar el mantenimiento de la lista de máquinas a las computadoras. Este sistema recuerda mucho a las arquitecturas anteriores de tiempo compartido.

Una extensión natural de esta idea es combinar esta programación con la reacción a la falla de la máquina. Al escanear los registros de la máquina en busca de expresiones que indiquen un mal estado (por ejemplo, errores de lectura masiva del disco), podemos identificar las máquinas que están rotas, señalar (a los humanos) la necesidad de reparar dichas máquinas y evitar programar cualquier trabajo en esas máquinas mientras tanto. . Extendiendo aún más la eliminación del trabajo duro, la automatización puede probar algunas soluciones primero antes de involucrar a un ser humano, como reiniciar la máquina, con la esperanza de que todo lo que estaba mal desaparezca, o ejecutar un escaneo de disco automatizado.

Una última queja de la cita de Jeff es la necesidad de que un humano migre el cómputo a otra máquina si la máquina en la que se está ejecutando falla. La solución aquí es simple: debido a que ya tenemos la automatización de la programación y la capacidad de detectar que una máquina está rota, simplemente podemos hacer que el programador asigne una nueva máquina y reinicie el trabajo en esta nueva máquina, abandonando la anterior. La señal para hacer esto puede provenir del demonio de introspección de la máquina o del monitoreo del proceso individual.

Todas estas mejoras abordan sistemáticamente la escala creciente de la organización. Cuando la flota era una sola máquina, SFTP y SSH eran soluciones perfectas, pero a la escala de cientos o miles de máquinas, la automatización debe hacerse cargo. La cita con la que comenzamos proviene de un documento de diseño de 2002 para la "Cola de trabajo global", una de las primeras soluciones internas de CaaS para algunas cargas de trabajo en Google.

Contenedорización y multiusuario

Hasta ahora, asumimos implícitamente un mapeo uno a uno entre las máquinas y los programas que se ejecutan en ellas. Esto es altamente ineficiente en términos de consumo de recursos informáticos (RAM, CPU), de muchas maneras:

- Es muy probable que haya muchos más tipos diferentes de trabajos (con diferentes requisitos de recursos) que tipos de máquinas (con diferente disponibilidad de recursos), por lo que muchos trabajos necesitarán usar el mismo tipo de máquina (que deberá aprovisionarse para el mayor de ellos).
- Las máquinas tardan mucho tiempo en implementarse, mientras que las necesidades de recursos del programa crecen con el tiempo. Si la obtención de máquinas nuevas y más grandes le toma meses a su organización, también debe hacerlas lo suficientemente grandes para acomodar el crecimiento esperado de

las necesidades de recursos durante el tiempo necesario para aprovisionar otras nuevas, lo que conduce al desperdicio, ya que las máquinas nuevas no se utilizan a su máxima capacidad.³

- Incluso cuando llegan las máquinas nuevas, todavía tiene las viejas (y probablemente sea un desperdicio tirarlas), por lo que debe administrar una flota heterogénea que no se adapta a sus necesidades.

La solución natural es especificar, para cada programa, sus requisitos de recursos (en términos de CPU, RAM, espacio en disco), y luego pedirle al programador que empaquete réplicas del programa en el conjunto disponible de máquinas.

El perro de mi vecino ladra en mi RAM

La solución antes mencionada funciona perfectamente si todos juegan bien. Sin embargo, si específico en mi configuración que cada réplica de mi tubería de procesamiento de datos consumirá una CPU y 200 MB de RAM, y luego, debido a un error o crecimiento orgánico, comienza a consumir más, las máquinas que obtiene programado en se quedará sin recursos. En el caso de la CPU, esto hará que los trabajos de servicio vecinos experimenten picos de latencia; en el caso de la RAM, provocará muertes por falta de memoria por parte del kernel o una latencia horrible debido al intercambio de disco.⁴

Dos programas en la misma computadora también pueden interactuar mal de otras maneras. Muchos programas querrán que sus dependencias estén instaladas en una máquina, en alguna versión específica, y esto podría chocar con los requisitos de versión de algún otro programa. Un programa puede esperar ciertos recursos de todo el sistema (piense en /tmp)estar disponible para su uso exclusivo. La seguridad es un problema: un programa puede estar manejando datos confidenciales y necesita asegurarse de que otros programas en la misma máquina no puedan acceder a ellos.

Por lo tanto, un servicio de cómputo multiinquilino debe proporcionar un grado de *aislamiento*, una garantía de algún tipo de que un proceso podrá continuar de manera segura sin ser perturbado por los otros inquilinos de la máquina.

Una solución clásica al aislamiento es el uso de máquinas virtuales (VM). Estos, sin embargo, vienen con una sobrecarga significativa en términos de uso de recursos (necesitan los recursos para ejecutar un sistema operativo completo en el interior) y tiempo de inicio (nuevamente, necesitan iniciar un sistema operativo completo). Esto los convierte en una solución menos que perfecta para trabajos por lotes.

³ Tenga en cuenta que este punto y el siguiente se aplican menos si su organización alquila máquinas de una nube pública proveedor.

⁴ Google ha elegido, hace mucho tiempo, que la degradación de la latencia debido al intercambio de disco es tan horrible que una la eliminación de memoria y una migración a una máquina diferente es universalmente preferible; por lo tanto, en el caso de Google, siempre se trata de una eliminación por falta de memoria.

⁵ Aunque se está realizando una cantidad considerable de investigación para reducir estos gastos generales, nunca será tan bajo como un proceso ejecutándose de forma nativa.

contenedorización para la que se esperan pequeñas huellas de recursos y tiempos de ejecución cortos. Esto llevó a los ingenieros de Google que diseñaron Borg en 2003 a buscar diferentes soluciones, y terminaron con *contenedores*—un mecanismo liviano basado en cgroups (aportado por los ingenieros de Google al kernel de Linux en 2007) y cárceles chroot, montajes de enlace y/o sistemas de archivos de unión/superposición para el aislamiento del sistema de archivos. Las implementaciones de contenedores de código abierto incluyen Docker y LMCTFY.

Con el tiempo y con la evolución de la organización, se descubren cada vez más fallas potenciales de aislamiento. Para dar un ejemplo específico, en 2011, los ingenieros que trabajaban en Borg descubrieron que el agotamiento del espacio de ID de proceso (que estaba configurado de manera predeterminada en 32 000 PID) se estaba convirtiendo en una falla de aislamiento y los límites en la cantidad total de procesos/subprocesos en un solo réplica puede generar tuvo que ser introducido. Veremos este ejemplo con más detalle más adelante en este capítulo.

Redimensionamiento y escalado automático

El Borg de 2006 programó el trabajo en función de los parámetros proporcionados por el ingeniero en la configuración, como el número de réplicas y los requisitos de recursos.

Mirando el problema desde la distancia, la idea de pedir a los humanos que determinen los números de requisitos de recursos es algo defectuosa: estos no son números con los que los humanos interactúan a diario. Y así, estos parámetros de configuración se convierten en sí mismos, con el tiempo, en una fuente de inefficiencia. Los ingenieros deben dedicar tiempo a determinarlos en el lanzamiento inicial del servicio y, a medida que su organización acumula más y más servicios, el costo para determinarlos aumenta. Además, a medida que pasa el tiempo, el programa evoluciona (probablemente crece), pero los parámetros de configuración no se mantienen. Esto termina en una interrupción, donde resulta que, con el tiempo, los nuevos lanzamientos tenían requisitos de recursos que consumían la holgura que quedaba para picos o interrupciones inesperados, y cuando tal pico o interrupción realmente ocurre, la holgura restante resulta ser insuficiente.

La solución natural es automatizar la configuración de estos parámetros. Desafortunadamente, esto resulta sorprendentemente difícil de hacer bien. Como ejemplo, Google ha llegado recientemente a un punto en el que más de la mitad del uso de recursos en toda la flota de Borg está determinado por la automatización del tamaño correcto. Dicho esto, aunque es solo la mitad del uso, es una fracción mayor de las configuraciones, lo que significa que la mayoría de los ingenieros no necesitan preocuparse por la carga tediosa y propensa a errores de dimensionar sus contenedores. Vemos esto como una aplicación exitosa de la idea de que "las cosas fáciles deberían ser fáciles y las cosas complejas deberían ser posibles". Un gran valor en el manejo de los casos fáciles.

Resumen

A medida que su organización crezca y sus productos se vuelvan más populares, crecerá en todos estos ejes:

- Número de aplicaciones diferentes a administrar
- Número de copias de una aplicación que debe ejecutarse
- El tamaño de la aplicación más grande

Para administrar la escala de manera efectiva, se necesita una automatización que le permita abordar todos estos ejes de crecimiento. Con el tiempo, debe esperar que la automatización en sí se involucre más, tanto para manejar nuevos tipos de requisitos (por ejemplo, la programación de GPU y TPU es un cambio importante en Borg que ocurrió en los últimos 10 años) como para aumentar la escala. Acciones que, a menor escala, podrían ser manuales, deberán automatizarse para evitar el colapso de la organización bajo la carga.

Un ejemplo, una transición que Google aún está en proceso de resolver, es la automatización de la gestión de nuestros *centros de datos*. Hace diez años, cada centro de datos era una entidad separada. Los gestionamos manualmente. Poner en marcha un centro de datos era un proceso manual complicado, que requería un conjunto de habilidades especializadas, que tomaba semanas (desde el momento en que todas las máquinas estaban listas) y era inherentemente riesgoso. Sin embargo, el crecimiento de la cantidad de centros de datos que administra Google hizo que nos moviéramos hacia un modelo en el que activar un centro de datos es un proceso automatizado que no requiere intervención humana.

Software de escritura para computación administrada

El paso de un mundo de listas de máquinas administradas manualmente a la programación y el dimensionamiento automatizados hizo que la administración de la flota fuera mucho más fácil para Google, pero también significó cambios profundos en la forma en que escribimos y pensamos sobre el software.

Arquitectura para el fracaso

Imagine que un ingeniero debe procesar un lote de un millón de documentos y validar su corrección. Si procesar un solo documento toma un segundo, el trabajo completo tomaría una máquina aproximadamente 12 días, lo que probablemente sea demasiado tiempo. Por lo tanto, dividimos el trabajo en 200 máquinas, lo que reduce el tiempo de ejecución a 100 minutos mucho más manejables.

Como se discutió en "Programación automatizada" en la página 519, en el mundo Borg, el programador puede matar unilateralmente a uno de los 200 trabajadores y moverlo a una máquina diferente.⁶ La parte "moverlo a una máquina diferente" implica que una nueva instancia de su trabajador puede eliminarse automáticamente, sin la necesidad de que un ser humano ingrese a la máquina mediante SSH y ajuste algunas variables de entorno o instale paquetes.

El paso de "el ingeniero tiene que monitorear manualmente cada una de las 100 tareas y atenderlas si se rompen" a "si algo sale mal con una de las tareas, el sistema está diseñado para que otros se hagan cargo de la carga, mientras que el programador automatizado lo mata y lo vuelve a instalar en una nueva máquina" ha sido descrito muchos años después a través de la analogía de "mascotas versus ganado".⁷

Si su servidor es una mascota, cuando se rompe, un humano viene a mirarlo (generalmente en estado de pánico), entiende qué salió mal y, con suerte, lo cuida hasta que se recupere. Es difícil de reemplazar. Si sus servidores son ganado, los nombra replica001 a replica100, y si uno falla, la automatización lo eliminará y aprovisionará uno nuevo en su lugar. La característica distintiva de "ganado" es que es fácil eliminar una nueva instancia del trabajo en cuestión: no requiere configuración manual y se puede realizar de forma totalmente automática. Esto permite la propiedad de autorreparación descrita anteriormente: en caso de falla, la automatización puede hacerse cargo y reemplazar el trabajo en mal estado por uno nuevo y saludable sin intervención humana. Tenga en cuenta que aunque la metáfora original hablaba de servidores (VM), lo mismo se aplica a los contenedores:

Si sus servidores son mascotas, su carga de mantenimiento crecerá linealmente, o incluso de forma superlativa, con el tamaño de su flota, y esa es una carga que ninguna organización debería aceptar a la ligera. Por otro lado, si sus servidores son ganado, su sistema podrá volver a un estado estable después de una falla, y no necesitará pasar el fin de semana cuidando un servidor o contenedor mascota para que recupere la salud.

Sin embargo, hacer que sus máquinas virtuales o contenedores sean ganado no es suficiente para garantizar que su sistema se comporte bien ante una falla. Con 200 máquinas, es muy probable que Borg elimine una de las réplicas, posiblemente más de una vez, y cada vez extiende la duración total en 50 minutos (o la cantidad de tiempo de procesamiento que se haya perdido). Para lidiar con esto con gracia, la arquitectura del procesamiento debe ser

⁶ El programador no hace esto arbitrariamente, sino por razones concretas (como la necesidad de actualizar el kernel o una el disco falla en la máquina, o una reorganización para mejorar la distribución general de las cargas de trabajo en el centro de datos). Sin embargo, el objetivo de tener un servicio informático es que, como autor de software, no debería saber ni preocuparme por las razones por las que esto podría suceder.

⁷La metáfora de "mascotas versus ganado" es atribuida a Bill Baker por Randy Biasy se ha vuelto extremadamente popular como una forma de describir el concepto de "unidad de software replicado". Como analogía, también se puede utilizar para describir conceptos distintos de servidores; por ejemplo, ver [capítulo 22](#).

diferente: en lugar de asignar el trabajo estáticamente, dividimos el conjunto completo de un millón de documentos en, digamos, 1000 fragmentos de 1000 documentos cada uno. Cada vez que un trabajador termina con un fragmento en particular, informa los resultados y toma otro. Esto significa que perdemos como máximo una porción de trabajo en caso de falla de un trabajador, en el caso de que el trabajador muera después de terminar la porción, pero antes de informarlo. Esto, afortunadamente, encaja muy bien con la arquitectura de procesamiento de datos que era el estándar de Google en ese momento: el trabajo no se asigna por igual al conjunto de trabajadores al comienzo del cómputo; se asigna dinámicamente durante el procesamiento general para dar cuenta de los trabajadores que fallan.

De manera similar, para los sistemas que atienden el tráfico de usuarios, lo ideal sería que se reprogramara un contenedor que no provoque errores para los usuarios. El programador Borg, cuando planea reprogramar un contenedor por motivos de mantenimiento, señala su intención al contenedor para avisarle con anticipación. El contenedor puede reaccionar a esto rechazando nuevas solicitudes mientras aún tiene tiempo para finalizar las solicitudes que tiene en curso. Esto, a su vez, requiere que el sistema del balanceador de carga comprenda la respuesta "No puedo aceptar nuevas solicitudes" (y redirija el tráfico a otras réplicas).

Para resumir: tratar sus contenedores o servidores como ganado significa que su servicio puede volver a un estado saludable automáticamente, pero se necesita un esfuerzo adicional para asegurarse de que pueda funcionar sin problemas mientras experimenta una tasa moderada de fallas.

Lote versus servir

Global WorkQueue (que describimos en la primera sección de este capítulo) abordó el problema de lo que los ingenieros de Google llaman "trabajos por lotes": programas que se espera que completen alguna tarea específica (como el procesamiento de datos) y que se ejecutan hasta su finalización. Los ejemplos canónicos de trabajos por lotes serían el análisis de registros o el aprendizaje de modelos de aprendizaje automático. Los trabajos por lotes contrastaban con los "trabajos de servicio", programas que se espera que se ejecuten indefinidamente y atiendan solicitudes entrantes, siendo el ejemplo canónico el trabajo que atendió consultas de búsqueda de usuarios reales del índice preconstruido.

Estos dos tipos de trabajos tienen (típicamente) características diferentes,⁸en particular:

- Los trabajos por lotes están interesados principalmente en el rendimiento del procesamiento. Los trabajos de entrega se preocupan por la latencia de atender una sola solicitud.
- Los trabajos por lotes son de corta duración (minutos o, como máximo, horas). Los trabajos de servicio suelen ser de larga duración (de forma predeterminada, solo se reinician con nuevas versiones).

⁸ Como todas las categorizaciones, esta no es perfecta; hay tipos de programas que no encajan perfectamente en ninguno de los categorías, o que posean características propias tanto de trabajos de servicio como por lotes. Sin embargo, como la mayoría de las categorizaciones útiles, todavía captura una distinción presente en muchos casos de la vida real.

- Debido a que son de larga duración, es más probable que los trabajos de servicio tengan tiempos de inicio más prolongados.

Hasta ahora, la mayoría de nuestros ejemplos eran sobre trabajos por lotes. Como hemos visto, para adaptar un trabajo por lotes para sobrevivir a las fallas, debemos asegurarnos de que el trabajo se distribuya en pequeños fragmentos y se asigne dinámicamente a los trabajadores. El marco canónico para hacer esto en Google fue MapReduce, luego reemplazado por Flume.¹⁰

Los trabajos de servicio son, en muchos sentidos, más adecuados para la resistencia a fallas que los trabajos por lotes. Su trabajo se divide naturalmente en pequeñas partes (solicitudes de usuarios individuales) que se asignan dinámicamente a los trabajadores: la estrategia de manejar un gran flujo de solicitudes a través del equilibrio de carga en un grupo de servidores se ha utilizado desde los primeros días de atender el tráfico de Internet.

Sin embargo, también hay múltiples aplicaciones de servicio que no se ajustan naturalmente a ese patrón. El ejemplo canónico sería cualquier servidor que describa intuitivamente como un "líder" de un sistema en particular. Dicho servidor normalmente mantendrá el estado del sistema (en la memoria o en su sistema de archivos local), y si la máquina en la que se ejecuta se cae, una instancia recién creada normalmente no podrá volver a crear el estado del sistema. Otro ejemplo es cuando tiene grandes cantidades de datos para servir (más de los que caben en una máquina) y decide dividir los datos entre, por ejemplo, 100 servidores, cada uno con el 1% de los datos y manejando solicitudes para esa parte. Esto es similar a la asignación estática de trabajo a los trabajadores del trabajo por lotes; si uno de los servidores se cae, usted (temporalmente) pierde la capacidad de servir una parte de sus datos. Un ejemplo final es si su servidor es conocido por otras partes de su sistema por su nombre de host. En ese caso, independientemente de cómo esté estructurado su servidor, si este host específico pierde la conectividad de la red, otras partes de su sistema no podrán comunicarse con él.¹¹

9 Véase Jeffrey Dean y Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", 6. Simposio sobre diseño e implementación de sistemas operativos (OSDI), 2004.

10 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw y Nathan Weizenbaum, "Flume-Java: Easy, Efficient Data-Parallel Pipelines", Conferencia ACM SIGPLAN sobre diseño e implementación de lenguajes de programación (PLDI), 2010.

11 Véase también Atul Adya et al. "Autofragmentación para aplicaciones de centros de datos", OSDI, 2019; y Atul Adya, Daniel Myers, Henry Qin y Robert Grandl, "Almacenes rápidos de valores clave: una idea cuyo tiempo llegó y se fue", HotOS XVII, 2019.

Estado administrador

Un tema común en la descripción anterior se centró en *Expresar* como una fuente de problemas cuando se trata de tratar trabajos como ganado.¹² Cada vez que reemplaza uno de sus trabajos de ganado, pierde todo el estado en proceso (así como todo lo que estaba en el almacenamiento local, si el trabajo se traslada a una máquina diferente). Esto significa que el estado en proceso debe tratarse como transitorio, mientras que el "almacenamiento real" debe ocurrir en otro lugar.

La forma más sencilla de lidiar con esto es extraer todo el almacenamiento a un sistema de almacenamiento externo. Esto significa que cualquier cosa que deba sobrevivir más allá del alcance de atender una sola solicitud (en el caso del trabajo de entrega) o procesar una parte de los datos (en el caso del lote) debe almacenarse fuera de la máquina, en un almacenamiento duradero y persistente. Si todo su estado local es inmutable, hacer que su aplicación sea resistente a fallas debería ser relativamente sencillo.

Desafortunadamente, la mayoría de las aplicaciones no son tan simples. Una pregunta natural que podría surgir es: "¿Cómo se implementan estas soluciones de almacenamiento persistentes y duraderas? *ellos* ganado?" La respuesta debería ser sí." El estado persistente puede ser manejado por el ganado a través de la replicación del estado. En un nivel diferente, las matrices RAID son un concepto análogo; tratamos los discos como transitorios (aceptamos el hecho de que uno de ellos puede desaparecer) mientras aún mantenemos el estado. En el mundo de los servidores, esto se puede realizar a través de múltiples réplicas que contienen una sola pieza de datos y se sincronizan para asegurarse de que cada pieza de datos se replique una cantidad suficiente de veces (generalmente de 3 a 5). Tenga en cuenta que configurar esto correctamente es difícil (se necesita alguna forma de manejo de consenso para lidiar con las escrituras), por lo que Google desarrolló una serie de soluciones de almacenamiento especializadas.¹³ que fueron habilitadores para la mayoría de las aplicaciones que adoptan un modelo donde todo el estado es transitorio.

Otros tipos de almacenamiento local que el ganado puede usar cubren los datos "recreables" que se guardan localmente para mejorar la latencia de servicio. El almacenamiento en caché es el ejemplo más obvio aquí: un caché no es más que un almacenamiento local transitorio que mantiene el estado en una ubicación transitoria, pero confía en que el estado no desaparece todo el tiempo, lo que permite mejores características de rendimiento en promedio. Una lección clave para la infraestructura de producción de Google ha sido aprovisionar el caché para cumplir con sus objetivos de latencia, pero aprovisionar la aplicación central para la carga total. Esto nos ha permitido evitar interrupciones cuando se perdió la capa de caché porque la ruta no almacenada en caché se aprovisionó para manejar la carga total.

12 Tenga en cuenta que, además del estado distribuido, existen otros requisitos para establecer un "servidores como ganado" efectivo solución, como sistemas de descubrimiento y balanceo de carga (para que su aplicación, que se mueve alrededor del centro de datos, pueda ser accedida de manera efectiva). Debido a que este libro trata menos sobre la construcción de una infraestructura CaaS completa y más sobre cómo dicha infraestructura se relaciona con el arte de la ingeniería de software, no entraremos en más detalles aquí.

13 Véase, por ejemplo, Sanjay Ghemawat, Howard Gobioff y Shun-Tak Leung, "The Google File System", Pro- actas del 19º Simposio de ACM sobre Sistemas Operativos, 2003; Fay Chang et al., "Bigtable: un sistema de almacenamiento distribuido para datos estructurados", 7º Simposio USENIX sobre diseño e implementación de sistemas operativos (OSDI); o James C. Corbett et al., "Spanner: base de datos distribuida globalmente de Google", OSDI, 2012.

(aunque con mayor latencia). Sin embargo, aquí hay una clara compensación: cuánto gastar en la redundancia para mitigar el riesgo de una interrupción cuando se pierde la capacidad de caché.

De manera similar al almacenamiento en caché, los datos pueden extraerse del almacenamiento externo al local en el calentamiento de una aplicación, para mejorar la latencia del servicio de solicitudes.

Un caso más de uso de almacenamiento local, esta vez en el caso de datos que se escriben más que leer, son las escrituras por lotes. Esta es una estrategia común para monitorear datos (piense, por ejemplo, en recopilar estadísticas de uso de CPU de la flota con el fin de guiar el sistema de escalado automático), pero se puede usar en cualquier lugar donde sea aceptable que perezca una fracción de datos, ya sea porque no necesitamos una cobertura de datos del 100% (este es el caso de monitoreo), o porque los datos que perecen se pueden volver a crear (este es el caso de un trabajo por lotes que procesa datos en fragmentos y escribe algunos resultados para cada uno). pedazo). Tenga en cuenta que, en muchos casos, incluso si un cálculo en particular debe llevar mucho tiempo, se puede dividir en ventanas de tiempo más pequeñas mediante la verificación periódica del estado en el almacenamiento persistente.

Conexión a un servicio

Como se mencionó anteriormente, si algo en el sistema tiene el nombre del host en el que se ejecuta su programa codificado (o incluso se proporciona como un parámetro de configuración al inicio), las réplicas de su programa no son ganado. Sin embargo, para conectarse a su aplicación, otra aplicación necesita obtener su dirección de alguna parte. ¿Dónde?

La respuesta es tener una capa adicional de direccionamiento indirecto; es decir, otras aplicaciones se refieren a su aplicación por algún identificador que es duradero en los reinicios de las instancias de "back-end" específicas. Ese identificador puede ser resuelto por otro sistema en el que el planificador escribe cuando coloca su aplicación en una máquina en particular. Ahora, para evitar búsquedas de almacenamiento distribuido en la ruta crítica de realizar una solicitud a su aplicación, es probable que los clientes busquen la dirección en la que se puede encontrar su aplicación, establezcan una conexión en el momento del inicio y la supervisen en segundo plano. . Esto generalmente se llama *descubrimiento de servicios* y muchas ofertas informáticas tienen soluciones integradas o modulares. La mayoría de estas soluciones también incluyen algún tipo de equilibrio de carga, lo que reduce aún más el acoplamiento a backends específicos.

Una repercusión de este modelo es que es probable que necesite repetir sus solicitudes en algunos casos, porque el servidor con el que está hablando podría desconectarse antes de que logre responder.¹⁴ Volver a intentar solicitudes es una práctica estándar para la comunicación de red (por ejemplo, aplicación móvil a un servidor) debido a problemas de red, pero puede ser menos intuitivo

¹⁴ Tenga en cuenta que los reintentos deben implementarse correctamente, con retroceso, degradación elegante y herramientas para evitar errores. coding fallas como jitter. Por tanto, esto probablemente debería ser parte de la biblioteca de llamadas a procedimientos remotos, en lugar de ser implementado a mano por cada desarrollador. Véase, por ejemplo, [Capítulo 22: Abordar fallas en cascada](#) en el libro de la SRE.

para cosas como un servidor comunicándose con su base de datos. Esto hace que sea importante diseñar la API de sus servidores de manera que maneje tales fallas con gracia. Para las solicitudes de mutación, tratar con solicitudes repetidas es complicado. La propiedad que desea garantizar es alguna variante de *idempotencia*—que el resultado de emitir una solicitud dos veces es el mismo que emitirla una vez. Una herramienta útil para ayudar con la idempotencia son los identificadores asignados por el cliente: si está creando algo (por ejemplo, un pedido para entregar una pizza a una dirección específica), el cliente asigna un identificador al pedido; y si ya se registró una orden con ese identificador, el servidor asume que se trata de una solicitud repetida y reporta éxito (también podría validar que los parámetros de la orden coincidan).

Una cosa más sorprendente que vimos que sucedió es que a veces el programador pierde contacto con una máquina en particular debido a algún problema de red. Luego decide que todo el trabajo allí se ha perdido y lo reprograma en otras máquinas, ¡y luego la máquina regresa! Ahora tenemos dos programas en dos máquinas diferentes, ambas pensando que son "replica072". La forma de eliminar la ambigüedad es verificar a cuál de ellos se refiere el sistema de resolución de direcciones (y el otro debe terminarse o terminarse); pero también es un caso más de idempotencia: dos réplicas que realizan el mismo trabajo y cumplen el mismo rol son otra fuente potencial de duplicación de solicitudes.

Código Único

La mayor parte de la discusión anterior se centró en los trabajos de calidad de producción, ya sea aquellos que atienden el tráfico de usuarios o las canalizaciones de procesamiento de datos que producen datos de producción. Sin embargo, la vida de un ingeniero de software también implica ejecutar análisis únicos, prototipos exploratorios, canalizaciones de procesamiento de datos personalizadas y más. Estos necesitan recursos informáticos.

A menudo, la estación de trabajo del ingeniero es una solución satisfactoria a la necesidad de recursos informáticos. Si uno quiere, por ejemplo, automatizar el escaneo a través de 1 GB de registros que produjo un servicio durante el último día para verificar si una línea A sospechosa siempre ocurre antes que la línea de error B, simplemente puede descargar los registros, escribir un breve Python y déjelo funcionar durante uno o dos minutos.

Pero si quieren automatizar el escaneo a través de 1 TB de registros que el servicio produjo durante el último año (para un propósito similar), esperar aproximadamente un día para que lleguen los resultados probablemente no sea aceptable. Un servicio de cómputo que le permite al ingeniero simplemente ejecutar el análisis en un entorno distribuido en varios minutos (utilizando unos pocos cientos de núcleos) significa la diferencia entre tener el análisis ahora y tenerlo mañana. Para las tareas que requieren iteración, por ejemplo, si necesito refinar la consulta después de ver los resultados, la diferencia puede estar entre hacerlo en un día y no hacerlo en absoluto.

Una preocupación que surge a veces con este enfoque es que permitir que los ingenieros ejecuten trabajos únicos en el entorno distribuido corre el riesgo de desperdiciar recursos. Esto

es, por supuesto, una compensación, pero que debe hacerse conscientemente. Es muy poco probable que el costo de procesamiento que ejecuta el ingeniero sea más costoso que el tiempo que el ingeniero dedica a escribir el código de procesamiento. Los valores de compensación exactos difieren según el entorno informático de una organización y cuánto paga a sus ingenieros, pero es poco probable que mil horas básicas cuesten algo parecido a un día de trabajo de ingeniería. Los recursos informáticos, en ese sentido, son similares a los marcadores, que discutimos al principio del libro; existe una pequeña oportunidad de ahorro para la empresa al instituir un proceso para adquirir más recursos informáticos, pero es probable que este proceso cueste mucho más en tiempo y oportunidad de ingeniería perdidos de lo que ahorra.

Dicho esto, los recursos de cómputo difieren de los marcadores en que es fácil tomar demasiados por accidente. Aunque es poco probable que alguien se lleve mil marcadores, es muy posible que alguien escriba accidentalmente un programa que ocupe mil máquinas sin darse cuenta.¹⁵ La solución natural a esto es instituir cuotas para el uso de recursos por ingenieros individuales. Una alternativa utilizada por Google es observar que debido a que estamos ejecutando cargas de trabajo por lotes de baja prioridad de manera efectiva y gratuita (consulte la sección sobre tenencia múltiple más adelante), podemos proporcionar a los ingenieros una cuota casi ilimitada para lotes de baja prioridad, que es lo suficientemente bueno para la mayoría de las tareas de ingeniería únicas.

CaaS a lo largo del tiempo y la escala

Hablamos anteriormente sobre cómo evolucionó CaaS en Google y las partes básicas necesarias para que esto suceda: cómo la simple misión de "simplemente dame recursos para ejecutar mis cosas" se traduce en una arquitectura real como Borg. Varios aspectos de cómo una arquitectura CaaS afecta la vida útil del software a lo largo del tiempo y la escala merecen una mirada más cercana.

Los contenedores como abstracción

Los contenedores, como los describimos anteriormente, se mostraron principalmente como un mecanismo de aislamiento, una forma de permitir la multitenencia, al tiempo que minimiza la interferencia entre diferentes tareas que comparten una sola máquina. Esa fue la motivación inicial, al menos en Google. Pero resultó que los contenedores también cumplen un papel muy importante en la abstracción del entorno informático.

Un contenedor proporciona un límite de abstracción entre el software implementado y la máquina real en la que se ejecuta. Esto significa que, con el tiempo, la máquina cambia, solo el software del contenedor (presuntamente administrado por un solo equipo) es el que tiene que cambiar.

15 Esto ha sucedido varias veces en Google; por ejemplo, debido a que alguien dejó la infraestructura de prueba de carga o porque un nuevo empleado estaba depurando un binario maestro en su estación de trabajo sin darse cuenta de que estaba generando 8000 trabajadores de máquina completa en segundo plano.

adaptarse, mientras que el software de la aplicación (administrado por cada equipo individual, a medida que crece la organización) puede permanecer sin cambios.

Analicemos dos ejemplos de cómo una abstracción en contenedores permite que una organización gestione el cambio.

Aabstracción del sistema de archivos proporciona una forma de incorporar software que no fue escrito en la empresa sin la necesidad de administrar configuraciones de máquina personalizadas. Esto podría ser software de código abierto que una organización ejecuta en su centro de datos o adquisiciones que desea incorporar a su CaaS. Sin una abstracción del sistema de archivos, incorporar un binario que espera un diseño de sistema de archivos diferente (por ejemplo, esperar un binario auxiliar en `/papelera/foo/bar`) requeriría modificar el diseño base de todas las máquinas de la flota, fragmentar la flota o modificar el software (lo que podría ser difícil o incluso imposible debido a consideraciones de licencia).

Aunque estas soluciones pueden ser factibles si la importación de una pieza de software externa es algo que sucede una vez en la vida, no es una solución sostenible si la importación de software se convierte en una práctica común (o incluso algo rara).

Una abstracción del sistema de archivos de algún tipo también ayuda con la gestión de dependencias porque permite que el software declare y empaquete previamente las dependencias (por ejemplo, versiones específicas de bibliotecas) que el software necesita para ejecutarse. Dependiendo del software instalado en la máquina, presenta una abstracción con fugas que obliga a todos a usar la misma versión de bibliotecas precompiladas y hace que la actualización de cualquier componente sea muy difícil, si no imposible.

Un contenedor también proporciona una forma sencilla de administrar *recursos con nombre* en la maquina. El ejemplo canónico son los puertos de red; otros recursos con nombre incluyen objetivos especializados; por ejemplo, GPU y otros aceleradores.

Inicialmente, Google no incluyó los puertos de red como parte de la abstracción del contenedor, por lo que los archivos binarios tuvieron que buscar ellos mismos los puertos no utilizados. Como resultado, el `pickUnusedPortOrDie()` función tiene más de 20 000 usos en el código base de Google C++. Docker, que se creó después de que se introdujeron los espacios de nombres de Linux, usa espacios de nombres para proporcionar contenedores con una NIC privada virtual, lo que significa que las aplicaciones pueden escuchar en cualquier puerto que deseen. Luego, la pila de red de Docker asigna un puerto en la máquina al puerto del contenedor. Kubernetes, que originalmente se construyó sobre Docker, va un paso más allá y requiere que la implementación de la red trate los contenedores ("pods" en el lenguaje de Kubernetes) como direcciones IP "reales", disponibles desde la red host. Ahora cada aplicación puede escuchar en cualquier puerto que desee sin temor a conflictos.

Estas mejoras son particularmente importantes cuando se trata de software que no está diseñado para ejecutarse en la pila de cómputo en particular. Aunque muchos programas populares de código abierto tienen parámetros de configuración para qué puerto usar, no hay coherencia entre ellos sobre cómo configurar esto.

Contenedores y dependencias implícitas

Como con cualquier abstracción, la Ley de dependencias implícitas de Hyrum se aplica a la abstracción del contenedor, probablemente se aplica *incluso más de lo habitual*, tanto por la gran cantidad de usuarios (en Google, todo el software de producción y mucho más se ejecutará en Borg) y porque los usuarios no sienten que están usando una API cuando usan cosas como el sistema de archivos (y es aún menos probable que lo hagan), pensar si esta API es estable, versionada, etc.).

Para ilustrar, volvamos al ejemplo del agotamiento del espacio de ID de proceso que experimentó Borg en 2011. Quizás se pregunte por qué los ID de proceso son agotables. ¿No son simplemente ID de números enteros que se pueden asignar desde el espacio de 32 bits o de 64 bits? En Linux, en la práctica se asignan en el rango [0,..., PID_MAX - 1], donde PID_MAX por defecto es 32.000. PID_MAX, sin embargo, se puede aumentar mediante un simple cambio de configuración (hasta un límite considerablemente más alto). ¿Problema resuelto?

Bueno no. Por la Ley de Hyrum, el hecho de que los PID que obtuvieron los procesos que se ejecutan en Borg estuvieran limitados al rango de 0...32,000 se convirtió en una garantía API implícita de la que la gente comenzó a depender; por ejemplo, los procesos de almacenamiento de registros dependían del hecho de que el PID se puede almacenar en cinco dígitos y fallaban para los PID de seis dígitos, porque los nombres de los registros excedían la longitud máxima permitida. Lidiar con el problema se convirtió en un largo proyecto de dos fases. Primero, un límite superior temporal en la cantidad de PID que puede usar un solo contenedor (de modo que un solo trabajo de fuga de subprocessos no pueda inutilizar toda la máquina). En segundo lugar, dividir el espacio PID para subprocessos y procesos. (Porque resultó que muy pocos usuarios dependían de la garantía de 32 000 para los PID asignados a subprocessos, a diferencia de los procesos. Entonces, podríamos aumentar el límite para subprocessos y mantenerlo en 32, 000 para procesos). La fase tres sería introducir espacios de nombres PID en Borg, dando a cada contenedor su propio espacio PID completo. Como era de esperar (nuevamente la Ley de Hyrum), una multitud de sistemas terminaron asumiendo que el triple {nombre de host, marca de tiempo, pid} identifica de manera única un proceso, que se rompería si se introdujeran espacios de nombres PID. El esfuerzo por identificar todos estos lugares y corregirlos (y respaldar cualquier dato relevante) aún continúa ocho años después.

El punto aquí no es que deba ejecutar sus contenedores en espacios de nombres PID. Aunque es una buena idea, no es la lección interesante aquí. Cuando se construyeron los contenedores de Borg, los espacios de nombres PID no existían; e incluso si lo hicieran, no es razonable esperar que los ingenieros que diseñaron Borg en 2003 reconozcan el valor de presentarlos. Incluso ahora, ciertamente hay recursos en una máquina que no están suficientemente aislados, lo que probablemente cause problemas algún día. Esto subraya los desafíos de diseñar un sistema de contenedores que se pueda mantener con el tiempo y, por lo tanto, el valor de usar un sistema de contenedores desarrollado y utilizado por una comunidad más amplia, donde este tipo de problemas ya han ocurrido para otros y se han incorporado las lecciones aprendidas.

Un servicio para gobernarlos a todos

Como se discutió anteriormente, el diseño original de WorkQueue estaba dirigido solo a algunos trabajos por lotes, que terminaron compartiendo un grupo de máquinas administradas por WorkQueue, y se usó una arquitectura diferente para servir trabajos, con cada trabajo de servicio particular ejecutándose en su propio, grupo dedicado de máquinas. El equivalente de código abierto sería ejecutar un clúster de Kubernetes separado para cada tipo de carga de trabajo (más un grupo para todos los trabajos por lotes).

En 2003, se inició el proyecto Borg, con el objetivo (y eventualmente logrando) construir un servicio de cómputo que asimila estos grupos dispares en un gran grupo. El grupo de Borg cubría tanto trabajos de servicio como por lotes y se convirtió en el único grupo en cualquier centro de datos (el equivalente sería ejecutar un solo clúster grande de Kubernetes para todas las cargas de trabajo en cada ubicación geográfica). Hay dos ganancias de eficiencia significativas aquí que vale la pena discutir.

La primera es que las máquinas de servir se convirtieron en ganado (tal como lo expresó el documento de diseño de Borg: “*Las máquinas son anónimas*:a los programas no les importa en qué máquina se ejecutan siempre que tenga las características adecuadas”). Si cada equipo que administra un trabajo de servicio debe administrar su propio grupo de máquinas (su propio clúster), la misma sobrecarga organizacional de mantenimiento y administración de ese grupo se aplica a cada uno de estos equipos. A medida que pasa el tiempo, las prácticas de gestión de estos grupos divergirán con el tiempo, lo que hará que los cambios en toda la empresa (como pasar a una nueva arquitectura de servidor o cambiar de centro de datos) sean cada vez más complejos. Una infraestructura de gestión unificada, es decir, una *común*servicio informático para todas las cargas de trabajo de la organización: permite a Google evitar este factor de escala lineal; no hay *norte*diferentes prácticas de gestión para las máquinas físicas en la flota, solo está Borg.¹⁶

El segundo es más sutil y puede que no sea aplicable a todas las organizaciones, pero fue muy relevante para Google. Las distintas necesidades de los trabajos por lotes y de servicio resultan ser complementarias. Por lo general, los trabajos de servicio deben aprovisionarse en exceso porque deben tener capacidad para atender el tráfico de usuarios sin disminuciones significativas de latencia, incluso en el caso de un pico de uso o una interrupción parcial de la infraestructura. Esto significa que una máquina que solo ejecuta trabajos de servicio estará infroutilizada. Es tentador tratar de aprovechar esa holgura comprometiendo en exceso la máquina, pero eso anula el propósito de la holgura en primer lugar, porque si ocurre el pico o la interrupción, los recursos que necesitamos no estarán disponibles.

Sin embargo, ¡este razonamiento se aplica solo a los trabajos de servicio! Si tenemos una cantidad de trabajos de servicio en una máquina y estos trabajos solicitan RAM y CPU que suman el

16 Como en todo sistema complejo, hay excepciones. No todas las máquinas propiedad de Google están gestionadas por Borg, y no todos los centros de datos están cubiertos por una sola celda Borg. Pero la mayoría de los ingenieros trabajan en un entorno en el que no tocan máquinas que no sean Borg o celdas no estándar.

tamaño total de la máquina, no se pueden colocar más trabajos de servicio allí, incluso si la utilización real de los recursos es solo el 30% de la capacidad. Pero nosotros *pueden*, en Borg, pondrá trabajos por lotes en el 70 % libre, con la política de que si alguno de los trabajos de servicio necesita memoria o CPU, lo recuperaremos de los trabajos por lotes (congelándolos en el caso de CPU o CPU). matando en el caso de la memoria RAM). Debido a que los trabajos por lotes están interesados en el rendimiento (medido en conjunto a través de cientos de trabajadores, no para tareas individuales) y sus réplicas individuales son ganado de todos modos, estarán más que felices de absorber esta capacidad adicional de atender trabajos.

Dependiendo de la forma de las cargas de trabajo en un grupo determinado de máquinas, esto significa que toda la carga de trabajo por lotes se está ejecutando de manera efectiva con recursos gratuitos (porque de todos modos estamos pagando por ellos en la holgura de los trabajos de servicio) o todo el trabajo de servicio. La carga está pagando efectivamente solo por lo que usan, no por la capacidad ociosa que necesitan para la resistencia a fallas (porque los trabajos por lotes se ejecutan con esa capacidad ociosa). En el caso de Google, la mayoría de las veces, ejecutamos lotes de manera efectiva de forma gratuita.

Multiusuario para servir trabajos

Anteriormente, discutimos una serie de requisitos que debe cumplir un servicio de cómputo para que sea adecuado para ejecutar trabajos de servicio. Como se discutió anteriormente, hay múltiples ventajas en tener los trabajos de servicio administrados por una solución de cómputo común, pero esto también presenta desafíos. Un requisito particular que vale la pena repetir es un servicio de descubrimiento, discutido en "[Conexión a un servicio](#)" en la página 528. Hay una serie de otros requisitos que son nuevos cuando queremos ampliar el alcance de una solución informática administrada para servir tareas, por ejemplo:

- Es necesario acelerar la reprogramación de trabajos: aunque probablemente sea aceptable eliminar y reiniciar el 50 % de las réplicas de un trabajo por lotes (porque solo provocará un parpadeo temporal en el procesamiento y lo que realmente nos importa es el rendimiento), es poco probable para ser aceptable eliminar y reiniciar el 50% de las réplicas de un trabajo de servicio (porque es probable que los trabajos restantes sean muy pocos para poder atender el tráfico de usuarios mientras se espera que los trabajos reiniciados vuelvan a funcionar).
- Por lo general, un trabajo por lotes se puede eliminar sin previo aviso. Lo que perdemos es parte del procesamiento ya realizado, que se puede rehacer. Cuando un trabajo de servicio se cancela sin previo aviso, es probable que nos arriesguemos a que el tráfico de cara al usuario devuelva errores o (en el mejor de los casos) tenga una mayor latencia; es preferible dar varios segundos de anticipación para que el trabajo termine de atender las solicitudes que tiene en vuelo y no acepte nuevas.

Por las razones de eficiencia antes mencionadas, Borg cubre tanto los trabajos por lotes como los de servicio, pero las múltiples ofertas de cómputo dividen los dos conceptos: por lo general, un grupo compartido de máquinas para trabajos por lotes y grupos de máquinas dedicados y estables para trabajos de servicio. Independientemente de si se usa la misma arquitectura informática para ambos tipos de trabajos, ambos grupos se benefician de ser tratados como ganado.

Configuración enviada

El programador Borg recibe la configuración de un servicio replicado o trabajo por lotes para ejecutar en la celda como contenido de una llamada de procedimiento remoto (RPC). Es posible que el operador del servicio lo administre mediante una interfaz de línea de comandos (CLI) que envía esos RPC y tiene los parámetros para la CLI almacenados en la documentación compartida o en su cabecera.

Dependiendo de la documentación y el conocimiento tribal sobre el código enviado a un repositorio rara vez es una buena idea en general porque tanto la documentación como el conocimiento tribal tienen una tendencia a deteriorarse con el tiempo (ver[Capítulo 3](#)). Sin embargo, el siguiente paso natural en la evolución (envolver la ejecución de la CLI en un script desarrollado localmente) sigue siendo inferior al uso de un lenguaje de configuración dedicado para especificar la configuración de su servicio.

Con el tiempo, la presencia en tiempo de ejecución de un servicio lógico generalmente crecerá más allá de un solo conjunto de contenedores replicados en un centro de datos en muchos ejes:

- Extenderá su presencia a través de múltiples centros de datos (tanto por afinidad de usuarios como por resistencia a fallas).
- Se bifurcará en tener entornos de preparación y desarrollo además del entorno/configuración de producción.
- Acumulará contenedores replicados adicionales de diferentes tipos en forma de servicios adjuntos, como un memcached que acompaña al servicio.

La gestión del servicio se simplifica mucho si esta configuración compleja se puede expresar en un lenguaje de configuración estandarizado que permita una fácil expresión de las operaciones estándar (como "actualizar mi servicio a la nueva versión del binario, pero eliminando no más del 5% de la capacidad en cualquier momento dado").

Un lenguaje de configuración estandarizado proporciona una configuración estándar que otros equipos pueden incluir fácilmente en su definición de servicio. Como de costumbre, enfatizamos el valor de dicha configuración estándar a lo largo del tiempo y la escala. Si cada equipo escribe un fragmento diferente de código personalizado para respaldar su servicio de Memcached, se vuelve muy difícil realizar tareas en toda la organización, como cambiar a una nueva implementación de Memcache (por ejemplo, por razones de rendimiento o de licencia) o para enviar una actualización de seguridad a todas las implementaciones de Memcache. Tenga en cuenta también que dicho lenguaje de configuración estandarizado es un requisito para la automatización en la implementación (ver[capítulo 24](#)).

Elegir un servicio de cómputo

Es poco probable que alguna organización siga el camino que tomó Google, construyendo su propia arquitectura informática desde cero. Actualmente, las ofertas informáticas modernas están disponibles tanto en el mundo de código abierto (como Kubernetes o Mesos o, en otro

nivel de abstracción, OpenWhisk o Knative), o como ofertas administradas en la nube pública (nuevamente, en diferentes niveles de complejidad, desde elementos como Managed Instance Groups de Google Cloud Platform o Amazon Web Services Elastic Compute Cloud [Amazon EC2] autoescalado; contenedores similares a Borg, como Microsoft Azure Kubernetes Service [AKS] o Google Kubernetes Engine [GKE], hasta una oferta sin servidor como AWS Lambda o Google's Cloud Functions).

Sin embargo, la mayoría de las organizaciones escogerán un servicio de cómputo, tal como lo hizo Google internamente. Tenga en cuenta que una infraestructura informática tiene un alto factor de bloqueo. Una de las razones es que el código se escribirá de forma que aproveche todas las propiedades del sistema (Ley de Hyrum); así, por ejemplo, si elige una oferta basada en VM, los equipos modificarán sus imágenes de VM particulares; y si elige una solución específica basada en contenedores, los equipos llamarán a las API del administrador de clústeres. Si su arquitectura permite que el código trate a las máquinas virtuales (o contenedores) como mascotas, los equipos lo harán, y luego será difícil pasar a una solución que dependa de que sean tratados como ganado (o incluso como diferentes formas de mascotas).

Para mostrar cómo incluso los detalles más pequeños de una solución informática pueden terminar bloqueados, considere cómo Borg ejecuta el comando que el usuario proporcionó en la configuración. En la mayoría de los casos, el comando será la ejecución de un binario (posiblemente seguido de varios argumentos). Sin embargo, por conveniencia, los autores de Borg también incluyeron la posibilidad de pasar un script de shell; por ejemplo, mientras que cierto; hacer ./ mi_binario; hecho.¹⁷ Sin embargo, mientras que una ejecución binaria se puede realizar a través de un simple fork-and-exec (que es lo que hace Borg), el script de shell debe ejecutarse mediante un shell como Bash. Entonces, Borg realmente ejecutó /usr/bin/bash -c \$COMANDO_USUARIO, que también funciona en el caso de una ejecución binaria simple.

En algún momento, el equipo de Borg se dio cuenta de que, a la escala de Google, los recursos (principalmente memoria) consumidos por este envoltorio de Bash no son despreciables y decidió pasar a utilizar un shell más ligero: ash. Entonces, el equipo hizo un cambio en el proceso. código de corredor para ejecutar /usr/bin/ash -c \$COMANDO_USUARIOen cambio.

Pensarías que este no es un cambio arriesgado; después de todo, controlamos el entorno, sabemos que ambos binarios existen, por lo que no debería haber forma de que esto no funcione. En realidad, la forma en que esto no funcionó es que los ingenieros de Borg no fueron los primeros en notar la sobrecarga de memoria adicional al ejecutar Bash. Algunos equipos fueron creativos en su deseo de limitar el uso de la memoria y reemplazaron (en su superposición de sistema de archivos personalizado) el comando Bash con un código personalizado de "ejecutar el segundo argumento". Estos equipos, por supuesto, eran muy conscientes de su uso de memoria, por lo que cuando el equipo de Borg cambió el ejecutor de procesos para usar ash (que no fue sobreescrito por

¹⁷ Este comando en particular es activamente dañino bajo Borg porque evita que los mecanismos de Borg para tratar con falla de activación. Sin embargo, contenedores más complejos que hacen eco de partes del entorno para el registro, por ejemplo, todavía se usan para ayudar a depurar problemas de inicio.

el código personalizado), su uso de memoria aumentó (porque comenzó a incluir el uso de cenizas en lugar del uso del código personalizado), y esto provocó alertas, revirtiendo el cambio y una cierta cantidad de infelicidad.

Otra razón por la que la elección de un servicio de cómputo es difícil de cambiar con el tiempo es que cualquier elección de servicio de cómputo eventualmente se verá rodeada por un gran ecosistema de servicios auxiliares: herramientas para registro, monitoreo, depuración, alertas, visualización, análisis sobre la marcha, lenguajes de configuración y metalenguajes, interfaces de usuario y más. Estas herramientas tendrían que reescribirse como parte de un cambio en el servicio informático, e incluso comprender y enumerar esas herramientas probablemente sea un desafío para una organización mediana o grande.

Por lo tanto, la elección de una arquitectura informática es importante. Al igual que con la mayoría de las opciones de ingeniería de software, esta implica compensaciones. Hablemos de algunos.

Centralización frente a personalización

Desde el punto de vista de la sobrecarga de administración de la pila de cómputo (y también desde el punto de vista de la eficiencia de los recursos), lo mejor que puede hacer una organización es adoptar una única solución CaaS para administrar toda su flota de máquinas y usar solo las herramientas disponibles, ahí para todos. Esto garantiza que, a medida que crece la organización, el costo de administrar la flota siga siendo manejable. Este camino es básicamente lo que Google ha hecho con Borg.

Necesidad de personalización

Sin embargo, una organización en crecimiento tendrá necesidades cada vez más diversas. Por ejemplo, cuando Google lanzó Google Compute Engine (la oferta de nube pública "VM como servicio") en 2012, Borg administraba las VM, como casi todo lo demás en Google. Esto significa que cada VM se ejecutaba en un contenedor separado controlado por Borg. Sin embargo, el enfoque de "ganado" para la gestión de tareas no se adaptaba a las cargas de trabajo de Cloud, porque cada contenedor en particular era en realidad una VM que un usuario en particular estaba ejecutando, y los usuarios de Cloud, por lo general, no trataban las VM como ganado.¹⁸

Conciliar esta diferencia requirió un trabajo considerable por ambas partes. La organización de la nube se aseguró de admitir la migración en vivo de máquinas virtuales; es decir, la capacidad de tomar una VM ejecutándose en una máquina, hacer girar una copia de esa VM en otra máquina, convertir la copia en una imagen perfecta y, finalmente, redirigir todo el tráfico a la copia, sin

¹⁸ Mi servidor de correo no es intercambiable con su trabajo de procesamiento de gráficos, incluso si ambas tareas se ejecutan: ning en la misma forma de VM.

causando un período notable cuando el servicio no está disponible.¹⁹ Borg, por otro lado, tuvo que adaptarse para evitar la eliminación voluntaria de contenedores que contenían máquinas virtuales (para proporcionar tiempo para migrar el contenido de la máquina virtual a la nueva máquina), y también, dado que todo el proceso de migración es más costoso, Los algoritmos de programación de Borg se adaptaron para optimizar y disminuir el riesgo de que se necesitara una reprogramación.²⁰ Por supuesto, estas modificaciones se implementaron solo para las máquinas que ejecutan las cargas de trabajo en la nube, lo que llevó a una bifurcación (pequeña, pero aún notable) de la oferta informática interna de Google.

Un ejemplo diferente, pero que también conduce a una bifurcación, proviene de la Búsqueda. Alrededor de 2011, uno de los contenedores replicados que atendía el tráfico web de la Búsqueda de Google tenía un índice gigante creado en discos locales, que almacenaba la parte del índice de Google de la web a la que se accede con menos frecuencia (las consultas más comunes se atendieron mediante cachés en memoria). de otros contenedores). La creación de este índice en una máquina en particular requería la capacidad de varios discos duros y tomó varias horas completar los datos. Sin embargo, en ese momento, Borg asumió que si alguno de los discos en los que un contenedor en particular tenía datos se había estropeado, el contenedor no podrá continuar y deberá reprogramarse en una máquina diferente. Esta combinación (junto con la tasa de fallas relativamente alta de los discos giratorios, en comparación con otro hardware) provocó graves problemas de disponibilidad; los contenedores se bajaban todo el tiempo y luego tardaban una eternidad en volver a ponerse en marcha. Para abordar esto, Borg tuvo que agregar la capacidad de un contenedor para manejar la falla del disco por sí mismo, optando por no recibir el tratamiento predeterminado de Borg; mientras que el equipo de Búsqueda tuvo que adaptar el proceso para continuar la operación con pérdida parcial de datos.

Muchas otras bifurcaciones, que cubren áreas como la forma del sistema de archivos, el acceso al sistema de archivos, el control, la asignación y el acceso a la memoria, la ubicación de la CPU/memoria, el hardware especial, las restricciones de programación especiales y más, hicieron que la superficie API de Borg se volviera grande y difícil de manejar, y la intersección de comportamientos se volvió difícil de predecir y aún más difícil de probar. Nadie sabía realmente si sucedía lo esperado si un contenedor solicitaba *ambas* *cosas* el trato especial de Cloud para el desalojo y el tratamiento personalizado de búsqueda para fallas en el disco (y en muchos casos, ni siquiera era obvio qué significa "esperado").

19 Esta no es la única motivación para hacer posible la migración en vivo de las máquinas virtuales de los usuarios; también ofrece considerables frente a los beneficios porque significa que el sistema operativo del host puede ser parcheado y el hardware del host actualizado sin interrumpir la máquina virtual. La alternativa (utilizada por otros proveedores importantes de la nube) es entregar "avisos de eventos de mantenimiento", lo que significa que la VM puede, por ejemplo, reiniciarse o detenerse y luego iniciarse por el proveedor de la nube.

20 Esto es particularmente relevante dado que no todas las máquinas virtuales de los clientes están optadas por la migración en vivo; por algún trabajo-cargas incluso el breve período de rendimiento degradado durante la migración es inaceptable. Estos clientes recibirán avisos de eventos de mantenimiento y Borg evitará desalojar los contenedores con esas máquinas virtuales a menos que sea estrictamente necesario.

Después de 2012, el equipo de Borg dedicó mucho tiempo a limpiar la API de Borg. Descubrió que algunas de las funcionalidades que ofrecía Borg ya no se usaban en absoluto.²¹ El grupo de funcionalidades más preocupante fueron aquellas que fueron utilizadas por múltiples contenedores, pero no estaba claro si intencionalmente, el proceso de copiar los archivos de configuración entre proyectos condujo a la proliferación del uso de funciones que originalmente estaban destinadas solo a usuarios avanzados. . Se introdujeron listas blancas para ciertas características para limitar su propagación y marcarlas claramente como solo para usuarios avanzados. Sin embargo, la limpieza aún está en curso y algunos cambios (como el uso de etiquetas para identificar grupos de contenedores) aún no se han realizado por completo.²²

Como es habitual con las compensaciones, aunque hay formas de invertir esfuerzo y obtener algunos de los beneficios de la personalización sin sufrir las peores desventajas (como la lista blanca mencionada anteriormente para la funcionalidad de energía), al final hay que tomar decisiones difíciles. . Estas opciones generalmente toman la forma de múltiples preguntas pequeñas: ¿aceptamos expandir la superficie API explícita (o peor aún, implícita) para acomodar a un usuario particular de nuestra infraestructura, o incomodamos significativamente a ese usuario, pero mantenemos una mayor coherencia?

Nivel de abstracción: sin servidor

La descripción de la domesticación del entorno informático por parte de Google se puede leer fácilmente como una historia de aumento y mejora de la abstracción: las versiones más avanzadas de Borg se ocuparon de más responsabilidades de gestión y aislaron más el contenedor del entorno subyacente. Es fácil tener la impresión de que esta es una historia simple: más abstracción es buena; menos abstracción es mala.

Por supuesto, no es tan simple. El paisaje aquí es complejo, con múltiples ofertas. En “[Domar el entorno informático](#)” en la página 518, discutimos la progresión desde el manejo de mascotas que funcionan con máquinas básicas (ya sea propiedad de su organización o alquiladas de un centro de colocación) hasta la gestión de contenedores como ganado. En el medio, como una ruta alternativa, se encuentran las ofertas basadas en máquinas virtuales en las que las máquinas virtuales pueden pasar de ser un sustituto más flexible del hardware completo (en ofertas de Infraestructura como servicio como Google Compute Engine [GCE] o Amazon EC2) a sustitutos más pesados de los contenedores. (con escalado automático, redimensionamiento y otras herramientas de gestión).

Según la experiencia de Google, la elección de administrar ganado (y no mascotas) es la solución para administrar a escala. Para reiterar, si cada uno de sus equipos necesita solo una máquina mascota en cada uno de sus centros de datos, sus costos de administración aumentarán de manera superlineal con su

21 Un buen recordatorio de que monitorear y rastrear el uso de sus funciones es valioso con el tiempo.

22 Esto significa que Kubernetes, que se benefició de la experiencia de limpiar Borg pero no se vio obstaculizado por una amplia base de usuarios existente para empezar, fue significativamente más moderno en bastantes aspectos (como el tratamiento de las etiquetas) desde el principio. Dicho esto, Kubernetes sufre algunos de los mismos problemas ahora que tiene una amplia adopción en una variedad de tipos de aplicaciones.

el crecimiento de la organización (porque tanto el número de equipos es probable que aumente el número de centros de datos que ocupa un equipo). Y después de que se toma la decisión de administrar el ganado, los contenedores son una opción natural para el manejo; son más livianos (lo que implica menores gastos generales de recursos y tiempos de inicio) y lo suficientemente configurables como para que, si necesita proporcionar acceso de hardware especializado a un tipo específico de carga de trabajo, puede (si así lo desea) permitir perforar un agujero fácilmente.

La ventaja de las máquinas virtuales como ganado radica principalmente en la capacidad de traer nuestro propio sistema operativo, lo cual es importante si sus cargas de trabajo requieren un conjunto diverso de sistemas operativos para ejecutarse. Varias organizaciones también tendrán experiencia preexistente en la administración de máquinas virtuales y configuraciones y cargas de trabajo preexistentes basadas en máquinas virtuales, por lo que podrían optar por usar máquinas virtuales en lugar de contenedores para reducir los costos de migración.

¿Qué es sin servidor?

Un nivel aún más alto de abstracción es *sin servidor*²³. Suponga que una organización ofrece contenido web y utiliza (o está dispuesta a adoptar) un marco de servidor común para manejar las solicitudes HTTP y atender las respuestas. El rasgo definitorio clave de un marco es la inversión del control, por lo que el usuario solo será responsable de escribir una "Acción" o un "Manejador" de algún tipo, una función en el idioma elegido que toma los parámetros de la solicitud y devuelve la respuesta. .

En el mundo Borg, la forma en que ejecutas este código es que levantas un contenedor replicado, cada réplica contiene un servidor que consiste en el código del marco y tus funciones. Si el tráfico aumenta, lo manejará escalando (agregando réplicas o expandiéndose a nuevos centros de datos). Si el tráfico disminuye, se reducirá. Tenga en cuenta que se requiere una presencia mínima (Google generalmente asume al menos tres réplicas en cada centro de datos en el que se ejecuta un servidor).

Sin embargo, si varios equipos diferentes usan el mismo marco, es posible un enfoque diferente: en lugar de solo hacer que las máquinas sean multiusuario, también podemos hacer que los servidores del marco sean multiusuario. En este enfoque, terminamos ejecutando una mayor cantidad de servidores de marco, cargamos/descargamos dinámicamente el código de acción en diferentes servidores según sea necesario y dirigimos dinámicamente las solicitudes a aquellos servidores que tienen cargado el código de acción relevante. Los equipos individuales ya no ejecutan servidores, por lo tanto, "sin servidor".

La mayoría de las discusiones sobre marcos sin servidor los comparan con el modelo de "máquinas virtuales como mascotas". En este contexto, el concepto sin servidor es una verdadera revolución, ya que trae todos los beneficios de la gestión del ganado: escalado automático, gastos generales más bajos, falta de aprovisionamiento explícito de servidores. Sin embargo, como se describió anteriormente, el paso a un sistema compartido, multiinquilino,

²³ FaaS (función como servicio) y PaaS (plataforma como servicio) son términos relacionados con serverless. Hay diferentes diferencias entre los tres términos, pero hay más similitudes, y los límites son algo borrosos.

el modelo basado en ganado ya debería ser un objetivo para una organización que planea escalar; por lo que el punto de comparación natural para las arquitecturas sin servidor debería ser la arquitectura de "contenedores persistentes" como Borg, Kubernetes o Mesosphere.

Pros y contras

Primero tenga en cuenta que una arquitectura sin servidor requiere que su código sea *verdaderamente apátrida*; es poco probable que podamos ejecutar las máquinas virtuales de sus usuarios o implementar Spanner dentro de la arquitectura sin servidor. Todas las formas de administrar el estado local (excepto no usarlo) de las que hablamos anteriormente no se aplican. En el mundo en contenedores, puede pasar unos segundos o minutos al inicio configurando conexiones a otros servicios, poblando cachés del almacenamiento en frío, etc., y espera que, en el caso típico, se le otorgue un período de gracia antes de la terminación. En un modelo sin servidor, no hay un estado local que realmente persista entre las solicitudes; todo lo que desea usar, debe configurarlo en el alcance de la solicitud.

En la práctica, la mayoría de las organizaciones tienen necesidades que no pueden satisfacerse con cargas de trabajo verdaderamente sin estado. Esto puede conducir a depender de soluciones específicas (ya sean propias o de terceros) para problemas específicos (como una solución de base de datos administrada, que es un compañero frecuente de una oferta sin servidor de nube pública) o a tener dos soluciones: un contenedor uno basado y otro sin servidor. Vale la pena mencionar que muchos o la mayoría de los marcos sin servidor se construyen sobre otras capas informáticas: AppEngine se ejecuta en Borg, Knative se ejecuta en Kubernetes, Lambda se ejecuta en Amazon EC2.

El modelo sin servidor administrado es atractivo para *escalado adaptable* del costo del recurso, especialmente en el extremo de poco tráfico. En, digamos, Kubernetes, su contenedor replicado no puede reducirse a cero contenedores (porque se supone que hacer girar tanto un contenedor como un nodo es demasiado lento para hacerlo en el momento de atender la solicitud). Esto significa que hay un costo mínimo de solo tener una aplicación disponible en el modelo de clúster persistente. Por otro lado, una aplicación sin servidor puede reducirse fácilmente a cero; y por lo tanto, el costo de poseerlo escala con el tráfico.

En el extremo de tráfico muy alto, necesariamente estará limitado por la infraestructura subyacente, independientemente de la solución informática. Si su aplicación necesita usar 100 000 núcleos para atender su tráfico, debe haber 100 000 núcleos físicos disponibles en cualquier equipo físico que respalde la infraestructura que está usando. En el extremo algo más bajo, donde su aplicación tiene suficiente tráfico para mantener varios servidores ocupados pero no lo suficiente como para presentar problemas al proveedor de infraestructura, tanto la solución de contenedor persistente como la solución sin servidor pueden escalar para manejarlo, aunque la escala de la solución sin servidor será más reactiva y granular que la del contenedor persistente.

Finalmente, adoptar una solución serverless implica una cierta pérdida de control sobre su entorno. En algún nivel, esto es algo bueno: tener el control significa tener que ejercerlo, y eso significa gastos generales de gestión. Pero, por supuesto, esto también significa que si

necesita alguna funcionalidad adicional que no está disponible en el marco que utiliza, se convertirá en un problema para usted.

Para tomar un ejemplo específico de eso, el equipo de Google Code Jam (realizando un concurso de programación para miles de participantes, con una interfaz que se ejecuta en Google AppEngine) tenía un script personalizado para llegar a la página web del concurso con un pico de tráfico artificial varios minutos antes de que comience el concurso, a fin de calentar suficientes instancias de la aplicación para atender el tráfico real que ocurrió cuando comenzó el concurso. Esto funcionó, pero es el tipo de ajuste manual (y también piratería) del que uno esperaría evitar eligiendo una solución sin servidor.

el intercambio

La elección de Google en esta compensación fue no invertir mucho en soluciones sin servidor. La solución de contenedores persistentes de Google, Borg, es lo suficientemente avanzada como para ofrecer la mayoría de los beneficios sin servidor (como escalado automático, varios marcos para diferentes tipos de aplicaciones, herramientas de implementación, registro unificado y herramientas de monitoreo, y más). Lo único que falta es el escalado más agresivo (en particular, la capacidad de reducir a cero), pero la gran mayoría de la huella de recursos de Google proviene de servicios de alto tráfico, por lo que es comparativamente más barato aprovisionar en exceso los servicios pequeños. Al mismo tiempo, Google ejecuta múltiples aplicaciones que no funcionarían en el mundo “verdaderamente sin estado”, desde GCE, a través de sistemas de bases de datos locales como [BigQuery](#) o Spanner, a servidores que tardan mucho tiempo en llenar el caché, como los trabajos de servicio de búsqueda de cola larga antes mencionados. Por lo tanto, los beneficios de tener una arquitectura unificada común para todas estas cosas superan las ganancias potenciales de tener una pila sin servidor separada para una parte de las cargas de trabajo.

Sin embargo, la elección de Google no es necesariamente la elección correcta para todas las organizaciones: otras organizaciones se han basado con éxito en arquitecturas mixtas de contenedor/sin servidor, o en arquitecturas puramente sin servidor que utilizan soluciones de almacenamiento de terceros.

Sin embargo, el atractivo principal de la tecnología sin servidor no se presenta en el caso de que una gran organización tome la decisión, sino en el caso de una organización o equipo más pequeño; en ese caso, la comparación es intrínsecamente injusta. El modelo sin servidor, aunque es más restrictivo, permite que el proveedor de la infraestructura se haga cargo de una parte mucho mayor de los gastos generales de administración general y, por lo tanto, *disminuir los gastos generales de gestión* para los usuarios. Ejecutar el código de un equipo en una arquitectura sin servidor compartida, como AWS Lambda o Google Cloud Run, es significativamente más simple (y económico) que configurar un clúster para ejecutar el código en un servicio de contenedor administrado como GKE o AKS si el clúster no se comparte entre muchos equipos. Si su equipo quiere aprovechar los beneficios de una oferta informática administrada, pero su organización más grande no quiere o no puede pasar a una solución basada en contenedores persistentes, es probable que una oferta sin servidor de uno de los proveedores de nube pública sea atractiva para usted porque la costo (en recursos y

administración) de un clúster compartido se amortiza bien solo si el clúster es realmente compartido (entre varios equipos de la organización).

Tenga en cuenta, sin embargo, que a medida que su organización crezca y se extienda la adopción de tecnologías administradas, es probable que supere las limitaciones de una solución puramente sin servidor. Esto hace que las soluciones en las que existe una ruta de ruptura (como de KNative a Kubernetes) sean atractivas dado que proporcionan una ruta natural a una arquitectura informática unificada como la de Google, en caso de que su organización decida seguir esa ruta.

Público versus Privado

Cuando Google estaba comenzando, las ofertas de CaaS eran principalmente de cosecha propia; si querías uno, lo construías. Su única opción en el espacio público versus privado era entre poseer las máquinas o alquilarlas, pero toda la administración de su flota dependía de usted.

En la era de la nube pública, hay opciones más baratas, pero también hay más opciones y una organización tendrá que hacerlas.

Una organización que utiliza una nube pública está subcontratando efectivamente (una parte de) los gastos generales de gestión a un proveedor de nube pública. Para muchas organizaciones, esta es una propuesta atractiva: pueden centrarse en proporcionar valor en su área específica de especialización y no necesitan aumentar significativamente la experiencia en infraestructura. Aunque los proveedores de la nube (por supuesto) cobran más que el costo básico del metal para recuperar los gastos de administración, ya tienen la experiencia acumulada y la comparten con múltiples clientes.

Además, una nube pública es una forma de escalar la infraestructura más fácilmente. A medida que crece el nivel de abstracción, desde las colocaciones, pasando por la compra de tiempo de máquina virtual, hasta los contenedores administrados y las ofertas sin servidor, la facilidad de ampliación aumenta, desde tener que firmar un contrato de alquiler para el espacio de colocación, hasta la necesidad de ejecutar una CLI para obtener algunas máquinas virtuales más, hasta herramientas de escalado automático para las cuales su huella de recursos cambia automáticamente con el tráfico que recibe. Especialmente para organizaciones o productos jóvenes, predecir los requisitos de recursos es un desafío, por lo que las ventajas de no tener que aprovisionar recursos por adelantado son significativas.

Una preocupación importante a la hora de elegir un proveedor de nube es el miedo al bloqueo: el proveedor podría aumentar repentinamente sus precios o simplemente fracasar, dejando a la organización en una posición muy difícil. Uno de los primeros proveedores de ofertas sin servidor, Zimki, un entorno de plataforma como servicio para ejecutar JavaScript, cerró en 2007 con un aviso de tres meses.

Una mitigación parcial para esto es usar soluciones de nube pública que se ejecuten con una arquitectura de código abierto (como Kubernetes). Esto tiene como objetivo garantizar que exista una ruta de migración, incluso si el proveedor de infraestructura en particular se vuelve inaceptable por algún motivo. Si bien esto mitiga una parte importante del riesgo, no es una solución perfecta.

estrategia. Debido a la Ley de Hyrum, es difícil garantizar que no se utilizarán piezas que sean específicas de un proveedor determinado.

Dos extensiones de esa estrategia son posibles. Una es usar una solución de nube pública de nivel inferior (como Amazon EC2) y ejecutar una solución de código abierto de nivel superior (como Open-Whisk o KNative) encima. Esto intenta garantizar que, si desea migrar, puede realizar los ajustes que haya realizado en la solución de nivel superior, las herramientas que creó sobre ella y las dependencias implícitas que tiene junto con usted. La otra es ejecutar multinube; es decir, usar servicios administrados basados en las mismas soluciones de código abierto de dos o más proveedores de nube diferentes (por ejemplo, GKE y AKS para Kubernetes). Esto proporciona una ruta aún más fácil para la migración desde uno de ellos y también hace que sea más difícil depender de los detalles de implementación específicos disponibles en uno de ellos.

Una estrategia más relacionada, menos para administrar el bloqueo y más para administrar la migración, es ejecutar en una nube híbrida; es decir, tenga una parte de su carga de trabajo general en su infraestructura privada y otra parte se ejecute en un proveedor de nube pública. Una de las formas en que esto se puede usar es usar la nube pública como una forma de lidiar con el desbordamiento. Una organización puede ejecutar la mayor parte de su carga de trabajo típica en una nube privada, pero en caso de escasez de recursos, escala algunas de las cargas de trabajo a una nube pública. Nuevamente, para que esto funcione de manera efectiva, se debe usar la misma solución de infraestructura informática de código abierto en ambos espacios.

Tanto las estrategias de nubes múltiples como las de nube híbrida requieren que los entornos múltiples estén bien conectados, a través de la conectividad de red directa entre máquinas en diferentes entornos y API comunes que están disponibles en ambos.

Conclusión

En el transcurso de la construcción, el perfeccionamiento y el funcionamiento de su infraestructura informática, Google aprendió el valor de una infraestructura informática común y bien diseñada. Tener una infraestructura única para toda la organización (p. ej., uno o un pequeño número de clústeres de Kubernetes compartidos por región) proporciona ganancias de eficiencia significativas en la gestión y los costos de recursos y permite el desarrollo de herramientas compartidas además de esa infraestructura. En la construcción de una arquitectura de este tipo, los contenedores son una herramienta clave para permitir compartir una máquina física (o virtual) entre diferentes tareas (lo que conduce a la eficiencia de los recursos), así como para proporcionar una capa de abstracción entre la aplicación y el sistema operativo que brinda resiliencia. tiempo extraordinario.

El buen uso de una arquitectura basada en contenedores requiere el diseño de aplicaciones para usar el modelo de "ganado": la ingeniería de su aplicación para que consista en nodos que puedan reemplazarse fácil y automáticamente permite escalar a miles de instancias. Escribir software para que sea compatible con ese modelo requiere diferentes patrones de pensamiento; por ejemplo, tratar todo el almacenamiento local (incluido el disco) como efímero y evitar nombres de host de codificación fija.

Dicho esto, aunque Google, en general, ha estado satisfecho y exitoso con su elección de arquitectura, otras organizaciones elegirán entre una amplia gama de servicios informáticos, desde el modelo de "mascotas" de máquinas virtuales o máquinas administradas a mano, hasta "ganado". contenedores replicados, hasta el modelo abstracto "sin servidor", todos disponibles en sabores gestionados y de código abierto; su elección es una compensación compleja de muchos factores.

TL; DR

- La escala requiere una infraestructura común para ejecutar cargas de trabajo en producción.
- Una solución informática puede proporcionar una abstracción y un entorno estandarizados y estables para el software.
- El software debe adaptarse a un entorno informático gestionado y distribuido.
- La solución de cómputo para una organización debe elegirse cuidadosamente para proporcionar los niveles adecuados de abstracción.

PARTE V

Conclusión

Epílogo

La ingeniería de software en Google ha sido un experimento extraordinario sobre cómo desarrollar y mantener una base de código grande y en evolución. He visto equipos de ingeniería abrir camino en este frente durante mi tiempo aquí, haciendo avanzar a Google como una empresa que afecta a miles de millones de usuarios y como líder en la industria de la tecnología. Esto no habría sido posible sin los principios descritos en este libro, así que estoy muy emocionado de ver que estas páginas cobran vida.

Si los últimos 50 años (o las páginas anteriores aquí) han demostrado algo, es que la ingeniería de software está lejos de estar estancada. En un entorno en el que la tecnología cambia constantemente, la función de ingeniería de software tiene un papel particularmente importante dentro de una organización determinada. Hoy en día, los principios de la ingeniería de software no se refieren simplemente a cómo administrar una organización de manera efectiva; se trata de cómo ser una empresa más responsable para los usuarios y el mundo en general.

Las soluciones a los problemas comunes de ingeniería de software no siempre están ocultas a simple vista; la mayoría requiere un cierto nivel de agilidad resuelta para identificar soluciones que funcionen para los problemas actuales y que también resistan los cambios inevitables en los sistemas técnicos. Esta agilidad es una cualidad común de los equipos de ingeniería de software con los que he tenido el privilegio de trabajar y aprender desde que me uní a Google en 2008.

La idea de sostenibilidad también es fundamental para la ingeniería de software. Durante la vida útil esperada de una base de código, debemos ser capaces de reaccionar y adaptarnos a los cambios, ya sea en la dirección del producto, las plataformas tecnológicas, las bibliotecas subyacentes, los sistemas operativos y más. Hoy en día, confiamos en los principios descritos en este libro para lograr una flexibilidad crucial al cambiar piezas de nuestro ecosistema de software.

Ciertamente, no podemos demostrar que las formas que hemos encontrado para lograr la sustentabilidad funcionen para todas las organizaciones, pero creo que es importante compartir estos aprendizajes clave. La ingeniería de software es una nueva disciplina, por lo que muy pocas organizaciones han tenido la oportunidad de lograr tanto la sostenibilidad como la escala. Al proporcionar esta descripción general de lo que hemos visto, así como los baches en el camino, nuestra esperanza es demostrar el valor y

viabilidad de la planificación a largo plazo para la salud del código. No se puede ignorar el paso del tiempo y la importancia del cambio.

Este libro describe algunos de nuestros principios rectores clave en relación con la ingeniería de software. En un alto nivel, también ilumina la influencia de la tecnología en la sociedad. Como ingenieros de software, es nuestra responsabilidad asegurarnos de que nuestro código esté diseñado con inclusión, equidad y accesibilidad para todos. Construir con el único propósito de innovar ya no es aceptable; la tecnología que ayuda sólo a un conjunto de usuarios no es innovadora en absoluto.

Nuestra responsabilidad en Google siempre ha sido proporcionar a los desarrolladores, internos y externos, un camino bien iluminado. Con el surgimiento de nuevas tecnologías como la inteligencia artificial, la computación cuántica y la computación ambiental, todavía tenemos mucho que aprender como empresa. Estoy particularmente emocionado de ver hacia dónde lleva la industria la ingeniería de software en los próximos años, y confío en que este libro ayudará a dar forma a ese camino.

—Asim Husain
Vicepresidente de Ingeniería, Google

Índice

simbolos

@anotación obsoleta,[322](#)
anotación @DoNotMock,[266](#)

A

Pruebas de diferencias A/B,[299](#)

limitaciones de,[300](#)
ejecución de presumisión,[304](#)
de las conductas del COU,[306](#)
296 compatibilidad ABI,[434](#)
Rappel, promesas de compatibilidad,[434](#)
interacciones de grupos adversarios,[47](#)
desaprobaciones de asesoramiento,[316](#) IA
(inteligencia artificial)
software de reconocimiento facial, desventaja
algunas poblaciones,[73](#)
semilla de datos, sesgos en,[73](#)
282 avión, parábola de,[108](#)
fatiga alerta,[318](#)
“Siempre de liderazgo”
estar siempre decidiendo,[108](#)
decidir, luego iterar,[110](#)
Siempre se va,[112](#) Siempre
estar escalando,[116](#)
resultados de análisis de analizadores de código,[404](#)
anotaciones, por prueba, documentación de propiedad,
308
Hormiga,[376](#)
realizar compilaciones proporcionando tareas a
línea de comando,[377](#)
reemplazo por sistemas de construcción más modernos,
378
antipatrones en suites de prueba,[220](#)
API

comentarios de la API,[193](#)
beneficios de la documentación para,[187](#) C++,
documentación para,[193](#) Búsqueda de código,
exposición de,[359](#) documentación conceptual y,[198](#)
declarar un tipo no debe ser objeto de burla,[266](#)
 fingiendo,[270](#)

backend de la interfaz de usuario del servicio que proporciona una API pública,
292
pruebas a través de API públicas,[234](#)-
[237](#) disculpándose por los errores,[94](#)
Ejemplo de AppEngine, exportar recursos,[454](#) Sello de
aprobación de los revisores,[412](#) aprobaciones para
cambios de código en Google,[167](#) arquitectura para el
fracaso,[523](#) sistemas de construcción basados en
artefactos,[380-386](#)
perspectiva funcional,[381](#)
concretando con Bazel,[381](#) otros
ingeniosos trucos de Bazel,[383-386](#)
tiempo, escala y compensaciones,[390](#)
haciendo preguntas,[48](#)
preguntar a los miembros del equipo si necesitan algo,
100
preguntando a la comunidad,[50](#)-
[52](#) técnica de gestión zen,[95](#)
afirmaciones
entre múltiples llamadas al sistema bajo
prueba,[243](#)
en la prueba de Java, usando la biblioteca Truth,[248](#) funciones
cortadas que tienen una relación directa
con,[275](#)
afirmación de prueba en Go,[248](#) verificar
el comportamiento de los COU,[295](#)
cambios atómicos, barreras a,[463-465](#)

atomicidad para confirmaciones en VCS,**328,332**
atención de los ingenieros (QUANTS),**131** críticas
de la audiencia,**199**
creación de pruebas grandes,**305** autorización para
cambios a gran escala,**473** sistema de construcción
automatizado,**372** pruebas automatizadas

comprobaciones de corrección de código,
172 límites de,**229**

automatización
comunicados A/B automatizados,**512** en
continua integración,**483-485** de revisiones
de código,**179** automatización del trabajo
en CaaS,**518-520**
programación automatizada,**519**
automatizaciones simples,**519** autonomía
para los miembros del equipo,**104** escalado
automático,**522**

B

reincidir, previniendo en proceso de depreciación,
322

retrocompatibilidad y reacciones a efi-
mejora de la ciencia,**11**

trabajos por lotes frente a trabajos de servicio,
525 bazel,**371,380**
versiones de dependencia,**394**
extender el sistema de compilación,
384 concretando con,**381**
parallelización de los pasos de construcción,**382**
realizar compilaciones con la línea de comandos,
382
reconstruir solo un conjunto mínimo de objetivos
cada vez,**383**
aislar el medio ambiente,**384** haciendo que las
dependencias externas sean determinantes
tic,**385**
independencia de la plataforma usando cadenas de herramientas,
384
almacenamiento en caché remoto y compilaciones reproducibles,
387
rapidez y corrección,**372** herramientas
como dependencias,**383**

Secciones de inicio, medio y final para documentos.
mentos,**202**

comportamientos
revisiones de código para cambios en,**181**
pruebas en lugar de métodos,**241-246**

nombrar las pruebas después del comportamiento que se está probando,
244

pruebas de estructuración para enfatizar comportamientos,
243

imprevisto, probando para,**284** actualizaciones de pruebas
para cambios en,**234** las mejores prácticas, la aplicación de
las reglas de la guía de estilo,**152** regla de beyoncé,**14,221**
sesgos,**18**

pequeñas expresiones de en interacciones,**48**
presencia universal de,**70**

binarios, interactuando, pruebas funcionales de,
297 autopsias sin culpa,**39-41,88** Resplandor,**371,**
380

gráfico de dependencia global,**496**

anteojeras, identificación,**109**

en el estudio de caso de latencia de búsqueda web,**110**

Biblioteca Boost C++, promesas de compatibilidad,**435**

administración de sucursales,**336-339**
nombres de sucursales en VCS,**330**
ramas de desarrollo,**337-339**

pocas sucursales longevas en Google,
343 liberar ramas,**339**

el trabajo en progreso es similar a una rama,**336**

"idiotas brillantes",**57**

pruebas frágiles,**224**

previniendo,**233-239**
luchando por pruebas inmutables,**233** estado
de prueba, no interacciones,**238** pruebas a
través de API públicas,**234-237** sistemas de
grabación/reproducción que causan,**492** con el uso
excesivo de stubing,**273** Pruebas de navegadores y
dispositivos,**297** Dólar,**380**

ataques de errores,**299**

corrección de errores,**181,234**

insectos
capturar más adelante en el desarrollo, costos de,**207** en
implementaciones reales provocando una cascada de
fallas en las pruebas,**265**

lógica que oculta un error en una prueba,**246** no se
previene solo con la habilidad del programador,
210

Archivos BUILD, reformateo,**162**

crear guiones
dificultades de los sistemas de construcción basados en tareas con,
379

escribir como tareas,**378**

construir sistemas,**371-398**

tratar con módulos y dependencias,
390-396
 administrar dependencias,**392-396**
 minimizando la visibilidad del módulo,**392**
 usando módulos de grano fino y 1:1:
 regla,**391**
 moderno,**375-390**
 basado en artefactos,**380-**
 386 dependencias y,**375**
 compilaciones distribuidas,**386-390**
 basado en tareas,**376-380**
 tiempo, escala y compensaciones,**390**
 propósito de,**371**
 utilizando otras herramientas en lugar de,**372-375**
 compiladores,**373**
 guiones de shell,**373**
patrón de constructor,**252**
archivos de compilación,**376**
 crear guiones y,**378**
 en sistemas de construcción basados en artefactos,**380**
 bazel,**381**
 construyendo para todos,**77** modelos de
 distribución agrupados,**441** factor de
 autobús,**31,112**

C

Lenguaje C, proyectos escritos, cambios a,**10**
C++
 API, documentación de referencia para,**193** Boost
 biblioteca, promesas de compatibilidad,**435** promesas
 de compatibilidad,**434** guía para desarrolladores para
 Googlers,**200** marco de burla de googlernock,**262**
 bibliotecas de banderas de línea de comandos de
 código abierto,
452
 scoped_ptr a std::unique_ptr,**467** almacenamiento en caché de los
 resultados de compilación mediante dependencias externas
 cias,**395**
Nomenclatura de CamelCase en Python,**154**
análisis canario,**301**
canario,**482**
 documentación canónica,**188** carreras, seguimiento
 de los miembros del equipo,**101** método de gestión
 del palo y la zanahoria,**86** catalizador, ser,**96**

analogía de ganado versus mascotas
 aplicar a los cambios en una base de
 código,**474** aplicar a la gestión del
 servidor,**524** CD (ver entrega continua)

celebridad,**29**
centralización versus personalización en computación
 servicios,**537-539**
sistemas centralizados de control de versiones (VCS),**332**
 futuro de,**348**
 desarrollado internamente, Piper en Google,**340**
 operaciones escalando linealmente con el tamaño de un
 cambio,**463**
 fuente de verdad en,**334**
 cambios locales no comprometidos y comprometidos
 cambios en una rama,**336**
gestión de cambios para cambios a gran escala,
470
Búsqueda de lista de cambios,**411**
listas de cambios (CL), aprobación de legibilidad para,**63**
cambios en el código
 cambiar las aprobaciones o puntuar un cambio,**412**
 creación de cambios en el proceso LSC,**473**
 comentando en,**408**
 cometiendo,**413**
 creando,**402-406**
 a gran escala (ver cambios a gran escala)
 historial de seguimiento de,**414** seguimiento
 en VCS,**328**
 tipos de cambios en el código de producción,
 233 comprender el estado de,**410-412** escribir
 buenas descripciones de cambios,**178** escribir
 pequeños cambios,**177**
caos e incertidumbre, blindando a tu equipo
 de,**102**
ingeniería del caos,**222,302** Valla de
Chesterson, principio de,**49** regla de
abandono,**13**
limpio y ordenado,**160**
 integración con Tricorder,**422**
comentarios de clase,**194**
clases y charlas técnicas,**52**
pruebas clásicas,**265**
código "limpio" y "mantenible",**10**
limpieza en proceso LSC,**477** pruebas
claras, escritura,**239-248**
 dejando la lógica fuera de las pruebas,**246** hacer que las
 pruebas grandes sean comprensibles,**307** hacer
 pruebas completas y concisas,**240** probar
 comportamientos, no métodos,**241-246**
 prueba basada en el comportamiento,
 242 prueba basada en métodos,**241**
 nombrar las pruebas después del comportamiento que se está probando,
244

- pruebas de estructuración para enfatizar comportamientos,
243
- escribir mensajes de error claros,**247**
- código "inteligente",**10**
- Ecosistema de gestión de paquetes Clojure,**447**
- proveedores de nube, públicos versus privados,**543**
- entrenar a un jugador de bajo rendimiento,**90** código
- beneficios de las pruebas,**213-214**
- código como un pasivo, no como un activo,**167**,
313 incrustando documentación con g3doc,
190
- expresando las pruebas como,**212**
- intercambio de conocimientos con,**56**
- calidad de,**131**
- cobertura de código,**222**
- formateadores de código,**161**
- revisones de código,**56,62-66,165-183**
- beneficios de,**166,170-176**
- consistencia del código,**173**
- comprensión del código,**172**
- corrección del código,**171** el
- intercambio de conocimientos,**175**
- psicológica y cultural,**174**
- mejores prácticas,**176-180**
- automatizando donde sea posible,**179** ser
eduado y profesional,**176** mantener los
revisores al mínimo,**179** escribir buenas
descripciones de cambios,**178** escribir
pequeños cambios,**177** código como un
pasivo,**167** flujo,**400**
- para cambios a gran escala,**467,476**
- cómo trabajan en Google,**167-169**
- propiedad del código,**169-170** entra,
166
- tipos de,**180-182**
- cambios de comportamiento, mejoras y
optimizaciones,**181**
- correcciones de errores y reversiones,**181**
- revisiones de campo nuevo,**180**
- refactorizaciones y cambios a gran escala,**182**
- Búsqueda de código,**178,351-370**
- implementación de Google,**361-365**
- clasificación,**363-365**
- índice de búsqueda,**361**
- cómo lo usan los Googlers,**353-355**
- responder dónde está algo en el
código base,**353**
- respondiendo quién y cuándo alguien introduce
código producido,**355**
- responder por qué el código se está comportando de manera
caminos esperados,**354**
- responder cómo otros han hecho algo
cosa,**354**
- respondiendo qué parte del código base es
haciendo,**354**
- impacto de la escala en el diseño,**359-361**
- latencia de índice,**360**
- latencia de consulta de búsqueda,**359** razones para una
herramienta web separada,**355-359**
- integración con otras herramientas de desarrollo,
356-359
- escala del código base de Google,
355 especialización,**356**
- vista de código global de configuración cero,**356**
- compensaciones en la implementación,**366-369**
- integridad, todo frente a lo más relevante
resultados,**366**
- integridad, cabeza vs. ramas vs. todo
- historia vs espacios de trabajo,**367**
- integridad, depósito en la cabecera,**366**
- expresividad, token vs. subcadena vs.
expresión regular,**368**
- interfaz de usuario,**352**
- compartir código, pruebas y,**248-255**
- definición de la infraestructura de prueba,**255**
- ayudantes compartidos y validación,**254**
- configuración compartida,**253**
- valores compartidos,**251**
- prueba que está demasiado SECA,**249** las
pruebas deben ser HÚMEDAS,**250** código
base
- análisis de cambios a gran escala y,**470** comentarios
en, documentación de referencia generada
borrado de,**193**
- factores que afectan la flexibilidad de,
dieciséis escalabilidad,**12**
- sostenibilidad,**12**
- valor de la consistencia de toda la base de código,**64** laboratorios
de código,**60**
- comentando los cambios en Crítica,**408**
- comentarios
- código,**193**
- reglas de la guía de estilo para,
145 comunidades
- transorganizacional, compartiendo conocimientos en,
62

obtener ayuda de la comunidad,[50-52](#) integración del compilador con análisis estático,[426](#) actualización del compilador (ejemplo),[14-dieciséis](#) compiladores, usando en lugar de construir sistemas,[373](#) exhaustividad y concisión en las pruebas,[240](#) integridad, precisión y claridad en la documentación **mención**,[202](#)

comprensión del código,[172](#) depreciación obligatoria,[317](#) Cómputo como servicio (CaaS),[517-545](#)

- elijir un servicio de cómputo,[535-544](#)
- centralización versus personalización, [537-539](#)
- nivel de abstracción, sin servidor,[539-543](#)
- público versus privado,[543](#) con el tiempo y la escala,[530-535](#)
- contenedores como una abstracción,[530-532](#)
- un servicio para gobernarlos a todos,[533](#)
- configuración enviada,[535](#) domar el entorno informático,[518-523](#)
- automatización del trabajo,[518-520](#)
- contenedorización y multiusuario, [520-522](#)
- escribir software para computación administrada, [523-530](#)
- arquitectura para el fracaso,[523](#)
- lote versus porción,[525](#)
- conectarse a un servicio,[528](#)
- Estado administrador,[527](#)
- código único,[529](#)

documentación conceptual,[198](#) comportamientos condescendientes y poco acogedores,[47](#) problemas de configuración con pruebas unitarias,[283](#) la creación de consenso,[96](#) consistencia dentro del código base,[146](#)

ventajas de,[146](#)

- construyendo en consistencia, reglas para,[153](#)
- asegurando con revisiones de código,[173](#) excepciones a, concediendo a los aspectos prácticos, [150](#)
- ineficiencia de consistencia perfecta en muy base de código grande,[148](#)
- Regla de una versión y,[342](#) estableciendo el estándar,[148](#) crítica constructiva,[37](#) pruebas de contrato impulsadas por el consumidor,[293](#)
- contenedorización y multiusuario,[520-522](#)

- redimensionamiento y escalado automático,[522](#)
- contenedores como una abstracción,[530-532](#)

contenedores y dependencias implícitas,[532](#) contexto, comprensión,[49](#) construcción continua (CB),[483](#) entrega continua (CD),[483,505-515](#)

dividir la implementación en manejable piezas,[507](#)

cambiar la cultura del equipo para construir disciplina en el despliegue,[513](#)

evaluando los cambios de forma aislada, flag-características de protección,[508](#)

modismos de CD en Google,[506](#)

calidad y enfoque en el usuario, enviando solo lo que se acostumbra,[511](#)

cambiar a la izquierda y tomar decisiones basadas en datos siones anteriores,[512](#)

luchando por la agilidad, configurando un tren de liberación, [509-510](#)

implementación continua (CD), ramas de lanzamiento y,[339](#)

integración continua (CI),[14,479-503](#)

- alertando,[487-493](#)
- desafíos de CI,[490](#)
- pruebas herméticas,[491](#)
- conceptos básicos,[481-487](#)
- automatización,[483-485](#) pruebas continuas, [485-487](#) bucles de retroalimentación rápidos, [481-483](#) ramas de desarrollo y,[338](#) revisiones totalmente nuevas que requieren una

proyecto,[180](#)

implementación en Google,[493-503](#)

- estudio de caso, Google Takeout,[496-502](#)
- TAP, construcción continua global,[494-496](#)

Live at Head gestión de dependencia y, [442](#)

sistema en Google,[223](#)

contratos falsos,[272](#)

interacciones grupales cooperativas,[47](#)

corrección en los sistemas de construcción, [372](#) corrección del código,[171](#) costos

en ingeniería de software,[12](#) reducir al encontrar problemas antes en desarrollo,[17](#)

compensaciones y,[18-23](#)

- decidir entre tiempo y escala (examen por favor),[22](#)
- compilaciones distribuidas (ejemplo),[20](#)
- insumos para la toma de decisiones,[20,20](#)

errores en la toma de decisiones,[22](#)
tipos de costos,[18](#)
marcadores de pizarra (ejemplo),[19](#) crítica,
aprendiendo a dar y recibir,[37](#) Herramienta de
revisión de código de crítica,[165,353,399-416](#)
cambiar las aprobaciones,[412](#)
flujo de revisión de código,[400](#)
principios de herramientas de revisión de
código,[399](#) cometiendo un cambio,[413](#)
creando un cambio,[402-406](#)
resultados de análisis,[404](#)
diferenciando,[403](#)
integración estrecha de herramientas,[406](#)
visor de diferencias, advertencias de Tricorder
activadas,[421](#) solicitar revisión,[406-408](#)
comprender y comentar un
cambio,[408-412](#)
vista de corrección de análisis
estático,[424](#) hashes criptográficos,[385](#)
búsqueda de culpables y aislamiento de fallas,[490](#)
usando TAP,[495](#)
cultura
construyendo disciplina en el despliegue,[513](#) cambios en
las normas que rodean a las LSC,[469](#) cultivar una cultura
de intercambio de conocimientos,
56-58
beneficios culturales de las revisiones de código,
174 cultura del aprendizaje,[43](#) basado en datos,[19,](#)
22
cultura saludable de pruebas automatizadas,
213 probando la cultura hoy en Google,[228](#)
clientes, documentación para,[192](#)
CVS (Sistema de Versiones Concurrentes),[328,332](#)

D

HÚMEDO,[249](#)
complementario a DRY, no un reemplazo,
251
prueba reescrita para ser DAMP,[250](#) panel de control
y sistema de búsqueda (Critique),[411](#) estructuras de
datos en bibliotecas, listados de,[158](#) cultura basada en
datos
acerca de,[19](#)
admitiendo errores,[22](#) decisiones basadas en
datos, tomando antes,[512](#) centros de datos,
automatización de la gestión de,[523](#) depuración
versus prueba,[210](#) decisiones
admitiendo haber cometido errores,[22](#)

decidir, luego iterar,[110](#)
en un grupo de ingeniería, justificaciones para,
19
insumos para la toma de decisiones,[20](#) haciendo en los
niveles más altos de liderazgo,[108](#) delegación de
subproblemas a los jefes de equipo,[113](#) dependencias
Bazel trata las herramientas como dependencias entre sí.
objetivo,[383](#)
construir sistemas y,[375](#)
construcción cuando se utilizan implementos reales.
ciones en las pruebas,[268](#)
contenedores y dependencias implícitas,[532](#)
gestión de dependencias frente a versión
control,[336](#)
externo, causando no determinismo en las pruebas,
268
externos, compiladores y,[373](#) bifurcación/
reimplementación versus agregar un
dependencia,[22](#)
en sistemas de compilación basados en tareas,[377](#) hacer que las
dependencias externas sean deterministas
en bazel,[385](#)
gestión de módulos en sistemas de compilación,
392-396
gestión automática vs. manual,[394](#) almacenamiento en
caché de los resultados de la compilación utilizando
dependencias,[395](#)
dependencias externas,[393](#)
dependencias internas,[392](#)
regla de una versión,[394](#)
seguridad y confiabilidad de los
dependencias,[396](#)
dependencias externas transitivas,[395](#) nuevo,
previniendo la introducción en la depresión
sistema alimentado,[322](#)
en valores en métodos de configuración compartidos,[253](#)
reemplazando todos en una clase con dobles de prueba,[265](#)
alcance de la prueba y,[219](#)
desconocido, descubriendo durante la desaprobación,
318
inyección de dependencia
marcos para,[261](#)
introduciendo costuras con,[260](#)
gestión de dependencia,[429-457](#)
dificultad de, razones de,[431-433](#)
requisitos contradictorios y diamante
dependencias,[431-433](#)
importar dependencias,[433-439](#)

promesas de compatibilidad,[433-436](#)
consideraciones en,[436](#)
el manejo de Google de,[437-439](#)
En teoría,[439-443](#)
modelos de distribución agrupados,[441](#)
Vive en Head,[442](#)
nada cambia (dependencia estática
modelo),[439](#)
versionado semántico,[440](#) limitaciones del
versionado semántico,[443-449](#)
Selección de versión mínima,[447](#)
motivaciones,[446](#)
sobre límite,[444](#)
compatibilidad demasiado
prometedora,[445](#) cuestionando si
funciona,[448](#) con infinitos recursos,[449-455](#)
exportar dependencias,[452-455](#)
despliegue
rompiendo en pedazos manejables,[507](#)
construyendo disciplina en,[513](#) pruebas de
configuración de implementación,[298](#) deprecación,[311-323](#)
como ejemplo de problemas de escala,[13](#)
dificultad de,[313-315](#) durante el
diseño,[315](#)
gestionando el proceso,[319-322](#)
herramientas de desaprobación,[321-322](#) hitos,[320](#)
propietarios de procesos,[320](#)
de documentación antigua,[203](#) prevenir nuevos
usos del objeto en desuso,
477
razones para,[312](#)
análisis estático en desuso de API,[417](#)
tipos de,[316-319](#)
desaprobación de asesoramiento,[316](#)
depreciación obligatoria,[317](#)
advertencias de desaprobación,[318](#)
Frases descriptivas y significativas (ver
HÚMEDO)
documentos de diseño,[195](#)
revisiones de diseño para nuevos códigos o proyectos,[180](#)
diseñar sistemas para que eventualmente sean obsoletos,
315
determinismo en las pruebas,[267](#)
ramas de desarrollo,[337-339](#)
sin ramas longevas y,[343](#) guías
para desarrolladores,[59](#)
felicidad del desarrollador, centrarse en, con análisis estático
sis,[419](#)
herramientas de desarrollo, integración de Code Search con,
356-359
flujo de trabajo del desarrollador, pruebas grandes y,[304-309](#)
creación de pruebas grandes,[305](#)
ejecutando grandes pruebas,[305-308](#)
expulsando la descamación,[306](#) hacer que las
pruebas sean comprensibles,[307](#) poseer
grandes pruebas,[308](#) pruebas de aceleración,[305](#)
flujo de trabajo del desarrollador, haciendo parte del análisis estático
de,[420](#)
DevOps
filosofía sobre la productividad tecnológica,[32](#)
desarrollo basado en troncales popularizado por,
327
Investigación y evaluación de DevOps (DORA)
sin ramas longevas y,[343](#) relación predictiva entre la
transmisión basada en troncales
organizaciones de desarrollo y alto
desempeño,[343](#)
investigación sobre ramas de liberación,[339](#)
problema de dependencia de diamantes,[394,431-433](#)
cambios de código diferentes,[403](#)
resumen de cambios y vista de diferencias,[404](#)
dirección, dar a los miembros del equipo,[104](#) pruebas
de recuperación ante desastres,[302](#) descubrimiento
(en desuso),[321](#) compilaciones distribuidas,[386-390](#)
en google,[389](#)
almacenamiento en caché remoto,[386](#)
ejecución remota,[387](#)
ejemplo de compensaciones y costos,[20](#) sistemas de
control de versiones distribuidas (DVCS),
332
compresión de datos históricos,[367](#)
escenario, ninguna fuente clara de verdad,[335](#) fuente de verdad,[334](#) diversidad
haciéndolo procesable,[74](#)
comprender la necesidad de,[72](#)
Estibador,[531](#)
documentación,[53-55,185-205](#)
acerca de,[185](#)
beneficios de,[186-187](#)
código,[56](#)
Integración de Code Search en,[358](#)
creando,[54](#)

para cambios de código,¹⁷⁸ conociendo a tu audiencia,¹⁹⁰⁻¹⁹²

- tipos de audiencias,¹⁹¹
- filosofía,²⁰¹⁻²⁰⁴
- secciones inicial, intermedia y final,²⁰²
- documentos en desuso,²⁰³ parámetros de una buena documentación,²⁰² quién, qué, por qué, cuándo, dónde y cómo,²⁰¹
- promoviendo,⁵⁵
- tratando como código,¹⁸⁸⁻¹⁹⁰
- Wiki de Google y,¹⁸⁹
- tipos de,¹⁹²⁻¹⁹⁹
- conceptual,¹⁹⁸
- documentos de diseño,¹⁹⁵
- páginas de destino,¹⁹⁸
- referencia,¹⁹³⁻¹⁹⁵
- tutoriales,¹⁹⁶
- actualizando,⁵⁴
- cuando necesite redactores técnicos,²⁰⁴
- comentarios de documentación,¹⁴⁵ revisiones de documentación,¹⁹⁹⁻²⁰¹ conocimiento documentado,⁴⁵
- Dominio del conocimiento de la audiencia de la documentación. ces,¹⁹¹
- Principio DRY (Don't Repeat Yourself)
- pruebas y código compartido, HÚMEDO, no SECO,²⁴⁸⁻²⁵⁵
- HÚMEDO como complemento de SECO,²⁵¹ prueba que está demasiado SECA,²⁴⁹ violando para pruebas más claras,²⁴¹ DVCS (ver sistema de control de versiones distribuidas).

elementos)

mi

Edison, Tomás,³⁸

formación de ingenieros de software,⁷²

- se necesita más educación inclusiva,⁷⁴ mejoras de eficiencia, cambio de código para,¹¹ ego, perder,^{36,93}
- Eisenhower, Dwight D.,¹¹⁸ correo electrónico en Google,⁵¹ Emerson, Ralph Waldo,¹⁵⁰ pruebas de extremo a extremo,²¹⁹

Gerentes de Ingeniería,^{82,86-88,88}

(ver también liderando un equipo; gerentes y tecnología Guías)

- gerentes contemporáneos,⁸⁷

hacerle saber al equipo que el fracaso es una opción,⁸⁷

gerente como palabra de cuatro letras,⁸⁶ productividad de ingeniería mejorando con las pruebas,²³¹ programa de legibilidad y, sesenta y cinco Equipo de investigación de productividad de ingeniería (EPR),
sesenta y cinco

productividad de ingeniería, medición,¹²³⁻¹³⁸

- evaluar el valor de medir,¹²⁵⁻¹²⁸
- metas,¹³⁰
- métrica,¹³²
- razones para,¹²³⁻¹²⁵
- seleccionar métricas significativas con objetivos y señales,¹²⁹⁻¹³⁰
- señales,¹³²

tomar medidas y hacer un seguimiento de los resultados después de formación de la investigación,¹³⁷

validar métricas con datos,¹³³⁻¹³⁷

ingeniería equitativa e inclusiva,⁶⁹⁻⁷⁹

sesgo y,⁷⁰

- construir capacidad multicultural,⁷²⁻⁷⁴
- desafiando los procesos establecidos,⁷⁶
- hacer que la diversidad sea acciónable,⁷⁴
- necesidad de diversidad,⁷² inclusión racial,⁷⁰
- rechazando enfoques singulares,⁷⁵ manteniendo la curiosidad y empujando hacia adelante,⁷⁸ valores versus resultados,⁷⁷
- herramientas de comprobación de errores,¹⁶⁰ Herramienta propensa a errores (Java),¹⁶⁰

anotación @DoNotMock,²⁶⁶ integración con Tricorder,⁴²² construcciones sorprendentes y propensas a errores en el código, evitando,¹⁴⁹

tiempo de ejecución de las pruebas,²⁶⁷

- pruebas de aceleración,³⁰⁵

niveles de experiencia para audiencias de documentación,¹⁹¹

experimentos y banderas de características,⁴⁸²

pericia

todo o nada,⁴⁴

- asesoramiento personalizado de un experto,⁴⁵ y foros de comunicación compartidos,¹⁴ problema de explotación versus exploración,³⁶³ prueba exploratoria,^{229,298} motivación extrínseca versus intrínseca,¹⁰⁴

F

"Fallar temprano, fallar rápido, fallar a menudo",

31 fracasos

abordar fallas en las pruebas,213 arquitectura para fallas en el software para el hombre

cómputo envejecido,523

error en la implementación real que causa una cascada de fallas en las pruebas,265

código claro que ayuda a diagnosticar fallas en las pruebas, 218

búsqueda de culpables y aislamiento de fallas,490

fallar rápido e iterar,38 el fracaso es una opción,87

gestión de fallos con TAP,495 gran prueba que falla,307 razones de las fallas en las pruebas,239 prueba de falla del sistema,222

escribir mensajes de error claros para las pruebas,247

fingiendo,263,269-272

backend hermético falso,491 fidelidad de

las falsificaciones,271 importancia de las

falsificaciones,270 probando

falsificaciones,272

cuando las falsificaciones no están

disponibles,272 cuando escribir fakes,270

falsos negativos en análisis estático,419 falsos positivos en análisis estáticos,419 banderas de características,482

funciones, novedades,234

estilo federado/virtual-monorepo (VMR) repositorio,346

comentario

acelerar el ritmo de progreso con,32 bucles de retroalimentación rápidos en CI,481-483 para documentación,54

dar retroalimentación dura a los miembros del equipo,98

canales de retroalimentación integrados en Tricorder,

423

solicitando a los desarrolladores sobre análisis estático, 419

fidelidad

de falsificaciones,271

de IVU,290

de dobles de prueba,258

de pruebas,282

comentarios de archivos,194

bloqueo de archivos en VCS,329,332 abstracción del

sistema de archivos,531 sistemas de archivos, VCS como forma de extender,329

características de protección de la bandera,

508 pruebas escamosas,216,267,490

eliminando la descamación en pruebas grandes,306 expensas de,218

Fragua,389,496 bifurcación/reimplementación

versus agregar un

dependencia,22

comentarios de funciones,195

lenguajes de programación funcionales,381

pruebas funcionales,219

prueba de uno o más binarios que interactúan, 297

GRAMO

g3doc,190

puertas, bill,28

archivos generados, índice de búsqueda de código y, 366 mito del genio,28

herramienta de revisión de código Gerrit,

414 Git,333

mejoras a,347

sintetizando el comportamiento monorepo,346 dado/

cuando/entonces, expresando comportamientos,242

alternando bloques cuando/entonces,244

prueba bien estructurada con,243 Ir

lenguaje de programación

promesas de compatibilidad,434 estudio de caso de la herramienta gofmt,161-163 ecosistema de gestión de paquetes estándar,

447

afirmación de prueba en,248

ir/enlaces,60

uso con documentación canónica,188,201

metas

definido,129

líder del equipo que establece objetivos claros,97

Marco de objetivos/señales/métricas (GSM),

129-133

metas,130

métrica,132

señales,132

uso para métricas en el estudio del proceso de legibilidad,

134

asistente de google,492

Búsqueda de Google,

y bifurcación de la comunicación interna de Google

ofrenda puta,538

pruebas más grandes en Google,286 probando

manualmente la funcionalidad de,210

subdividiendo el problema de latencia de, 113 estudio de caso de Google Takeout, 496-502 Servidor web de Google (GWS), 209 Wiki de Google (GooWiki), 189 "Googley", siendo, 41

Gradle, 376

versiones de dependencia, 394
mejoras en hormiga, 378
revisiones de código greenfield, 180
comando grep, 352
charlas grupales, 50
Gruñido, 376

H

código "hacky" o "inteligente", 10 Hamming, Ricardo, 35, 36 felicidad, seguimiento para tu equipo, 99 fuera de la oficina y en sus carreras, 100 ataques de inundación de hash, 9 orden hash (ejemplo), 9 cementerios embrujados, 44, 464 descorazonado, 10

Tutoriales de "Hola Mundo", 196

métodos auxiliares

ayudantes compartidos y validación, 254
valores compartidos en, 252

código hermético, no determinismo y, 268

IVU herméticos, 290

beneficios de, 291

pruebas herméticas, 491

asistente de google, 492

culto a los héroes, 28

escondiendo tu trabajo

Genio Mito y, 29 efectos nocivos de, 30-34

factor de autobús, 31

ingenieros y oficinas, 32 renunciando a la detección temprana de fallas o asuntos, 31

ritmo de progreso, 32

contratación de ingenieros de software

comprometer la barra de contratación (antipatrón), 92

contratación de pushovers (antipatrón), 89 hacer que la diversidad sea accionable, 75 historial, indexación en Code Search, 367 honestidad, ser honesto con tu equipo, 98 "La esperanza no es una estrategia", 89 antipatrón de reloj de arena en las pruebas, 220

cuestiones humanas, ignorando en un equipo, 90 problemas humanos, solución, 29 humildad, 35

siendo "Googley", 41
practicando, 36-39

IVU híbridos, 291

ley de Hyrum, 8

consideración en las pruebas unitarias, 284 desprecio y, 313
orden hash (ejemplo), 9

yo

antipatrón de cono de helado en las pruebas, 220, 287

idempotencia, 529

IDE (entornos de desarrollo integrado)

razones para usar Code Search en lugar de, 355-359

análisis estático y, 427

reconocimiento de imagen, inclusión racial y, 70

lenguajes de programación imperativos, 381

comentarios de implementación, 145, 193 problemas importantes versus urgentes, 118 mejoras al código existente, revisiones de código

por, 181

incentivos y reconocimiento para el conocimiento compartido

En g, 57

construcciones incrementales, dificultad en tareas basadas

construir sistemas, 379

índices

Búsqueda de código versus IDE, 355 soltar archivos del

índice de búsqueda de código, 366 indexar múltiples

versiones de un repositorio,

367

latencia en búsqueda de código, 360 índice de búsqueda en Code Search, 361 ingenieros

individuales, aumentando la productividad de, 124

influencia, estar abierto a, 40

influir sin autoridad (estudio de caso), 83 islas de información, 43 información, fuentes canónicas de, 58-61

laboratorios de código, 60

guías para desarrolladores, 59

ir/enlaces, 60

análisis estático, 61

inseguridad, 28

crítica y, 37

manifestación en Genius Myth, 29

pruebas de integración, 219

complejidad intelectual (QUANTS),[131](#)
pruebas de interacción,[238,275-280](#)
usos apropiados de,[277](#)
mejores prácticas,[277](#)
evitando la sobreespecificación,[278](#)
actuar solo para cambiar de estado
funciones,[277](#)
prefiriendo las pruebas estatales a las[275](#)
limitaciones de las pruebas de interacción,[276](#)
usando dobles de prueba,[264](#) interoperabilidad de
código,[151](#)
diferenciación intralínea que muestra diferencias en el nivel de los caracteres
encias,[403](#)
motivación intrínseca versus extrínseca,[104](#) iteración,
haciendo que sus equipos se sientan cómodos con,
110

j

Java

aserción en una prueba usando la biblioteca Truth,[248](#) compilador javac,[373](#)
Mockito marco de burla para,[262](#)
sombreado,[342](#)
archivos JAR de terceros,[373](#)
paradoja de jevons,[21](#)
trabajos, steve,[28,92](#)
Jordán, Michael,[28](#)
JUnit,[305](#)

k

abstracciones clave y estructuras de datos en bibliotecas,
listados de,[158](#)
el intercambio de conocimientos,[43-67](#)
como beneficio de las revisiones de
código,[175](#) preguntando a la comunidad,
50-52 desafíos para el aprendizaje,[43](#)
papel crítico de la seguridad psicológica,[46-48](#)
aumentando tu conocimiento,[48,49](#)
haciendo preguntas,[48](#)
comprensión del contexto,[49](#) aumentar el
conocimiento trabajando con otros
er,[31](#)
filosofía de,[45](#)
proceso de legibilidad y revisiones de código,[158](#)
escalar el conocimiento de su organización,
56-62
cultivar una cultura de intercambio de conocimientos,
56-58

establecer fuentes canónicas de información
mación,[58-61](#)
mantenerse en el bucle,[61-62](#) tutoría
estandarizada a través del código
reseñas,[62-66](#)
enseñando a otros,[52-56](#)
Kondo, María,[119](#)
clústeres de Kubernetes,[533](#)
prestigio,[58](#)
Kythe,[470](#)
integración con Code Search,[351](#) navegar
por referencias cruzadas con,[406](#)

L

páginas de destino,[198](#)
Herramientas y procesos de cambio a gran escala,[148](#)
grandes pruebas,[217](#)
(ver también pruebas más grandes)
cambios a gran escala,[372,459-478](#)
barreras a los cambios atómicos,[463-465](#)
heterogeneidad,[464](#)
fusionar conflictos,[463](#)
sin cementerios embrujados,[464](#)
limitaciones técnicas,[463](#)
pruebas,[465](#)
revisiones de código para,[182](#)
importancia del desarrollo basado en troncos
y,[343](#)
infraestructura,[468-472](#)
gestión del cambio,[470](#)
conocimiento de la base de código,[470](#)
ayuda de idioma,[471](#)
Operación RoseHub,[472](#)
políticas y cultura,[469](#)
pruebas,[471](#)
pruebas más grandes omitidas durante,[285](#)
proceso,[472-477](#)
autorización,[473](#)
creación de cambios,[473](#)
limpiar,[477](#)
fragmentación y presentación,[474-477](#)
cualidades de,[460](#)
Responsabilidad para,[461-462](#)
pruebas,[466-468](#)
revisiones de código,[467](#)
viajando en el tren TAP,[466](#) scoped_ptr a
std::unique_ptr,[467](#) pruebas más
grandes,[281-309](#)
ventajas de,[282](#)

desafíos y limitaciones de,**285**
características de,**281** fidelidad de las pruebas,**282**
grandes pruebas y flujo de trabajo del desarrollador,**304-309**
 creación de pruebas grandes,**305**
 ejecutando grandes pruebas,**305-308**
pruebas más grandes en Google,**286-289**
 Escala de Google y,**288**
 tiempo y,**286**
estructura de una prueba grande,**289-296**
sistemas bajo prueba (SUT),**290-294** datos de prueba,**294**
verificación,**295**
tipos de pruebas grandes,**296-304**
 Diferencia A/B (regresión),**299** Pruebas de navegadores y dispositivos,**297** pruebas de configuración de implementación,**298** recuperación ante desastres e ingeniería del caos,**302**
prueba exploratoria,**298**
pruebas funcionales de binarios que interactúan,**297**
pruebas de rendimiento, carga y estrés,**297**
probers y análisis de canarios,**301**
UAT,**301**
evaluación del usuario,**303**
pruebas unitarias que no proporcionan una buena mitigación de riesgos cobertura,**283-284**
bases de datos de reconocimiento facial de aplicación de la ley, sesgo racial en,**74** liderazgo, idiotas brillantes y,**57** liderazgo, convertirse en un líder realmente bueno,**107-122**
 Abordar la latencia de búsqueda web (estudio de caso),**110-112**
 estar siempre decidiendo,**108** Siempre se va,**112** Siempre estar escalando,**116** decidir, luego iterar,**110** identificar compensaciones clave,**109** identificando las anteojeras,**109** problemas importantes vs. urgentes,**118-119** aprendiendo a dropar balones,**119** protegiendo tu energía,**120** liderando un equipo,**81-105**
antipatrones,**88-93**
 ser amigo de todos,**91** comprometiendo la barra de contratación,**92**
contratando empujones,**89**
ignorando los problemas humanos,**90** ignorando a los de bajo rendimiento,**89** tratar a su equipo como niños,**92** preguntar a los miembros del equipo si necesitan algo,**100**
Director de Ingeniería,86-88
 el fracaso como opción,**87**
 historia de los gerentes,**86**
 gerente de hoy,**87**
satisfacer las diferentes necesidades de los miembros del equipo,**103**
motivación,**104**
gerentes y líderes tecnológicos,81-83
 estudio de caso, influencia sin autoridad,**83**
 Director de Ingeniería,**82** líder tecnológico,**82**
 gerente líder de tecnología,**82**
pasando de colaborador individual a líder
 papel de ership,**83-86**
 razones por las que la gente no quiere ser dirigida er,**84**
 liderazgo de servicio,**85**
 otros consejos y trucos para,**101**
 patrones positivos,**93-100**
 siendo un catalizador,**96**
 siendo un maestro y mentor,**97**
 ser un maestro zen,**94** Siendo honesto,**98**
 perdiendo el ego,**93**
 eliminando obstáculos,**96**
 establecer metas claras,**97**
 rastreando la felicidad,**99**
aprendizaje,**46**
 (ver también intercambio de conocimientos) desafíos para,**43**
Sello LGTM (me parece bien) de revisor,166
 cambiar la aprobación con,**401** la aprobación del propietario del código y,**168** controles de corrección y comprensión,**167** del revisor principal,**178** el significado de,**412**
separación de la aprobación de la legibilidad,**173** los líderes tecnológicos envían un cambio de código después,**168** bibliotecas, compiladores y,**373** linters en Tricorder,**425** linux

desarrolladores de,²⁸
 parches del kernel, fuentes de verdad para,³³⁵
 Vive en el modelo Head,⁴⁴² Prueba de carga,²⁹⁷

visor de registros, integración de Code Search con,
³⁵⁷ la lógica, no poner en pruebas,²⁴⁶ LSC (ver
 cambios a gran escala)

METRO

listas de correo,⁵⁰
 mantenibilidad de las pruebas,²³²
 "gerencialismo",⁸⁴
 gerentes y líderes tecnológicos,⁸¹⁻⁸³
 antipatrones,⁸⁸⁻⁹³
 ser amigo de todos,⁹¹ comprometiendo
 la barra de contratación,⁹² contratando
 empujones,⁸⁹
 ignorando los problemas humanos,⁹⁰
 ignorando a los de bajo rendimiento,⁸⁹ tratar a
 su equipo como niños,⁹² estudio de caso, influir
 sin autoridad,⁸³ Director de Ingeniería,^{82,86-88}

contemporáneo,⁸⁷
 el fracaso como opción,⁸⁷
 historia de los gerentes,⁸⁶

pasando de colaborador individual a líder

 papel de ership,⁸³⁻⁸⁶
 razones por las que la gente no quiere ser dirigida
 er,⁸⁴
 liderazgo de servicio,⁸⁵

patrones positivos,⁹³⁻¹⁰⁰

 siendo un catalizador,⁹⁶
 siendo un maestro y mentor,⁹⁷
 ser un maestro zen,⁹⁴ Siendo
 honesto,⁹⁸
 perdiendo el ego,⁹³ eliminando
 obstáculos,⁹⁶ establecer
 objetivos claros,⁹⁷ rastreando
 la felicidad,⁹⁹

líder tecnológico,⁸²
 gerente líder de tecnología (TLM),
⁸² prueba manual,²⁸⁶

Reducción,¹⁹⁰

dominio para los miembros del equipo,¹⁰⁴
 experto,³⁷⁶
 mejoras en hormiga,³⁷⁸

mediciones,¹²³

(ver también productividad de ingeniería, medición
 En g)

en áreas difíciles de cuantificar,²⁰
 pruebas medianas,²¹⁷

fusión y espectro,¹¹

tutoría,⁴⁶

 ser un maestro y mentor para su equipo,
⁹⁷

estandarizado, a través de revisiones de código,⁶²⁻⁶⁶
 conflictos de fusión, tamaño de los cambios y,⁴⁶³ fusiona

proceso de ramificación y fusión, desarrollo como,
³³⁰

coordinación de la fusión de ramas de desarrollo,³³⁸
 ramas de desarrollo y,³³⁸ combinar seguimiento en VCS,
³²⁹ pruebas basadas en métodos,²⁴¹

prueba de ejemplo,²⁴¹
 patrones de nomenclatura de métodos de muestra,²⁴⁶

métrica

evaluar el valor de medir,¹²⁵⁻¹²⁸ en el marco
 GSM,^{129,132} significativa, seleccionando con
 metas y señales,
¹²⁹⁻¹³⁰

usando datos para validar,¹³³⁻¹³⁷

migraciones

en el proceso de desaprobación,³²² migrar
 usuarios de un sistema obsoleto,
³¹⁷

hitos de un proceso de desaprobación,³²⁰ Selección de
 versión mínima (MVS),⁴⁴⁷ dispositivos móviles, pruebas
 de navegadores y dispositivos,²⁹⁷ burlón,²⁵⁷

(ver también dobles de prueba)

pruebas de interacción y,²⁶⁴

mal uso de objetos simulados, causando pruebas frágiles,
²²⁴

se burla de volverse rancio,
²⁸³ marcos burlones

acerca de,²⁶¹

para los principales lenguajes de programación,²⁶²

pruebas de interacción realizadas a través de,²⁶⁴

exceso de confianza en,^{239,259} vía de stubing,²⁶⁴

prueba burlona,²⁶⁵

Mockito

 ejemplo de uso,²⁶²
 ejemplo de tropezar,²⁶³

módulos, que se ocupan de los sistemas de construcción,
³⁹⁰⁻³⁹⁶

 administrar dependencias,³⁹²⁻³⁹⁶

minimizando la visibilidad del módulo,³⁹² utilizando módulos de grano fino y la regla 1:1:¹,
391
monorrepos,³⁴⁵
argumentos en contra,³⁴⁶
organizaciones que citan los beneficios de,³⁴⁶ motivando a tu equipo,¹⁰³
motivación intrínseca vs extrínseca,¹⁰⁴ detección de movimiento para fragmentos de código,⁴⁰³
capacidad multicultural, construcción,⁷²⁻⁷⁴
cómo las desigualdades en la sociedad afectan el lugar de trabajo ces,⁷⁴
SUT multimáquina,²⁹¹
multiusuario, contenedrización y,⁵²¹⁻⁵²²
multiusuario para servir trabajos,⁵³⁴
servidores de marco multiusuario,⁵⁴⁰

norte
recursos con nombre, gestión en la máquina,
531
puertos de red, contenedores y,⁵³¹
boletines,⁶¹
ningún binario es perfecto,⁵⁰⁹ funciones que no cambian de estado,²⁷⁸ comportamiento no determinista en las pruebas,^{216,218,}
267
notificaciones de Critica,⁴⁰²

O

horario de oficina, uso para compartir conocimientos,⁵²
regla 1:1:³⁹¹
código único,⁵²⁹
regla de una versión,^{340,342,394}
monorrepos y,³⁴⁵
Software de código abierto (OSS)
gestión de la dependencia y,⁴³⁰
monorrepos y,³⁴⁷
banderas de código abierto,
452 Operación RoseHub,⁴⁷²
optimizaciones de código existente, revisiones de código por,¹⁸¹
sobreespecificación de las pruebas de interacción,²⁷⁸ propiedad del código,¹⁶⁹⁻¹⁷⁰
propietarios del proceso de desaprobación,³²⁰ para revisiones de greenfield,¹⁸⁰
propiedad granular en Google monorepo,
340
poseer grandes pruebas,³⁰⁸

PAG

prueba de contrato de pacto,²⁹³
Pantalones,³⁸⁰
parallelización de pasos de compilación
dificultad en los sistemas basados en tareas,³⁷⁸
en bazel,³⁸³
parallelización de pruebas,²⁶⁷
repitiendo,⁴⁴
Pascual, Blaise,¹⁹¹
pacienza y amabilidad en responder preguntas,
49
pacienza, aprendizaje,³⁹
bonos de compañeros,⁵⁸
Por fuerza, números de revisión para variar,
336 actuación
acomodar optimizaciones en el código.
base,¹⁵¹
pruebas,²⁹⁷
rendimiento de los ingenieros de software
fallas en las calificaciones de desempeño,⁷⁶
ignorando a los de bajo rendimiento,⁸⁹
gastos de personal,¹⁸
“Principio de Pedro”,⁸⁴
Gaitero,³⁴⁰
Integración de Code Search con,³⁵³ herramientas construidas sobre,⁴⁰⁶ políticas para cambios a gran escala,⁴⁶⁹ cortesía y profesionalismo en las revisiones de código,
176
post mortem, sin culpa,^{39-41,88} revisiones previas al compromiso,⁴⁰⁰
presupone,¹⁷⁹
cheques en Tricorder,⁴²⁵ pruebas continuas y,⁴⁸⁵ infraestructura para grandes pruebas,³⁰⁵
optimización de,^{490,494}
probando fusiones en la rama dev,³³⁸
versus envío posterior,⁴⁸⁶
sondeadores,³⁰¹
problemas
dividir el espacio del problema,¹¹³⁻¹¹⁶
importante vs urgente,¹¹⁸ estabilidad del producto, ramas de desarrollo y,³³⁷
producción
riesgos de las pruebas en,²⁹²
probando en,⁴⁸⁷
profesionalismo en revisiones de código,¹⁷⁶
programación
código inteligente y,¹⁰

ingeniería de software versus,^{3,23}
guía de programación,¹⁵⁷ lenguajes de
programación
consejos para las áreas más difíciles de corregir,
158
evitando el uso de errores propensos y sorprendentes
construcciones,¹⁴⁹
desgloses de nuevas características y consejos sobre
utilizarlos,158
documentando,²⁰²
imperativo y funcional,³⁸¹ limitaciones
en nuevos y no-todavía-bien-
características entendidas,¹⁵²
lógica en,²⁴⁶
documentación de referencia,¹⁹³ guías de estilo
para cada idioma,¹⁴² apoyo para cambios a gran
escala,⁴⁷¹ Herramienta Proyecto Salud (pH),²²⁸
personalización a nivel de proyecto en Tricorder,⁴²⁵
análizador Proto Best Practices,⁴²⁴ el protocolo
amortigua el análisis estático de,⁴²⁵ proveedores,
documentación para,¹⁹² beneficios psicológicos de las
revisões de código,¹⁷⁴ seguridad psicológica,⁴⁶⁻⁴⁸

construyendo a través de la tutoría,⁴⁶
catalizando a su equipo construyendo,⁸⁷
en grandes grupos,⁴⁷ falta de,⁴³

servicios informáticos públicos frente a privados,⁵⁴³
API públicas,²³⁷
propósito para los miembros del equipo,¹⁰⁵
propósito de los usuarios de la documentación,¹⁹¹
Piton,²⁸
marco unittest.mock para,²⁶² Guías
de estilo de Python
evitación de características de potencia como la reflexión
ción,¹⁴⁹
Nomenclatura CamelCase vs. snake_case,
154 sangría del código,¹⁴⁹

q
métricas cualitativas,¹³³ calidad y
atención al usuario en CD,⁵¹¹ calidad
de código,¹³¹
QUANTS en métricas de productividad de ingeniería,
130
en el estudio del proceso de legibilidad,¹³⁴
señales dependientes de la consulta,³⁶⁴
consultar señales independientes,³⁶³

sistema de preguntas y respuestas (YAQS),⁵¹
preguntas, preguntando (ver haciendo preguntas)

R

sesgo racial en las bases de datos de reconocimiento facial,⁷⁴
inclusión racial,⁷⁰
Rastrillo,³⁷⁶
clasificación en búsqueda de código,³⁶³⁻³⁶⁵
señales dependientes de la consulta,³⁶⁴
consultar señales independientes,³⁶³
diversidad de resultados,³⁶⁵
recuperación,³⁶⁵
RCS (Sistema de Control de Revisión),^{329,332}
legibilidad,^{56,62-66}
aprobación de cambios de código en Google,¹⁶⁸
asegurando con revisiones de código,¹⁷³ proceso
de legibilidad,⁵⁶
acerca de,⁶³
ventajas de,64
implementaciones reales, utilizando en lugar de prueba dou-
bendiciones,²⁶⁴⁻²⁶⁹
decidir cuándo usar implementaciones reales,
266-269
construcción de dependencia,²⁶⁸
determinismo en las pruebas,²⁶⁷
Tiempo de ejecución,²⁶⁷
prefiriendo el realismo al aislamiento,²⁶⁵
sesgo de recuerdo,¹³⁴
sesgo de actualidad,¹³⁴
reconocimiento por el intercambio de conocimientos,⁵⁷
recomendaciones sobre los resultados de la investigación,
¹³⁷ sistemas de grabación/reproducción,^{293,492}
redundancia en la documentación,²⁰² refactorizaciones,²³³

revisiones de código para,¹⁸²
a gran escala, y el uso de referencias para la clasificación
En g,³⁶⁴
basado en buscar y reemplazar,³⁶⁰ trabajo no
comprometido como similar a una rama,³³⁷
documentación de referencia,¹⁹³⁻¹⁹⁵
comentarios de clase,¹⁹⁴
comentarios de archivos,¹⁹⁴
comentarios de funciones,¹⁹⁵
referencias, utilizando para la clasificación,³⁶³
pruebas de regresión,²⁹⁹
(ver también pruebas de diferencias A/B)
búsqueda de expresiones regulares (regex),³⁶⁸
reimplementación/bifurcación versus agregar un
dependencia,22

- liberar ramas,[339](#)
Google y,[344](#)
- pruebas de candidatos de lanzamiento,[486](#)
- lanzamientos
- luchando por la agilidad, configurando un tren de liberación,[509](#)
 - cumplir con su fecha límite de liberación,[510](#)
 - ningún binario es perfecto,[509](#) fiabilidad de las dependencias externas,[396](#) almacenamiento en caché remoto en compilaciones distribuidas,[386](#)
 - caché remota de Google,[389](#)
 - ejecución remota de compilaciones distribuidas,[387](#)
 - Sistema de ejecución remota de Google, Forge,[389](#)
 - repositorios,[328](#)
 - repositorio central para un proyecto en DVCS,[333](#)
 - de grano más fino vs. monorepos,[345](#) bifurcación del repositorio, no utilizada en Google,[223](#) prueba representativa,[512](#) limitaciones de recursos, CI y,[490](#) el respeto,[35](#)
 - siendo "Googley",[41](#)
 - practicando,[36-39](#),[57](#) diversidad de resultados en la búsqueda,[365](#)
 - recuperación,[365](#)
 - revisores de código, manteniendo al mínimo,[179](#)
 - redimensionamiento y escalado automático,[522](#) riesgos
- hacer del fracaso una opción,
[87](#) de trabajar solo,[30](#)
- barricadas, eliminación,[96](#)
- retrocesos,[181](#)
- herramienta rosa,[470](#)
- fragmentación y envío en el proceso LSC,
[474-477](#)
- reglas que rigen el código,[141](#)
- categorias de reglas en las guías de estilo
 - construcción de reglas en consistencia,[153](#)
 - reglas que hacen cumplir las mejores prácticas,[152](#) reglas para evitar el peligro,[151](#) temas no tratados,[153](#) cambiando,[154-157](#) haciendo cumplir,[158-163](#)
- estudio de caso gofmt,[161-163](#) usando formateadores de código,[161](#) usando verificadores de errores,[160](#) principios rectores para,[143-151](#)
 - evitando errores propensos y sorprendentes construcciones,[149](#)
- siendo consistente,[146](#)
- concediendo a los aspectos prácticos,[150](#)
- optimizando para el lector de código, no el autor,[144](#)
- las reglas deben tirar de su peso,[144](#)
- razones para tener,[142](#) reglas, definiendo en Bazel,[384](#)
- ## S
- sesgo de muestreo,[134](#)
- sandboxing
- pruebas herméticas y,[492](#)
 - uso por Bazel,[384](#)
- satisfacción (CUANTOS),[131](#)
- escalabilidad
- bifurcación y,[22](#)
 - de herramientas de análisis estático,[418](#)
 - escala
 - decidir entre el tiempo y,[22](#) en ingeniería de software,[4](#) problemas en ingeniería de software,[5](#) escala y eficiencia,[11-17](#)
 - actualización del compilador (ejemplo),[14-dieciséis](#) encontrar problemas antes en el trabajo del desarrollador.
 - flujo,[17](#)
- políticas que no escalan,[12](#)
- políticas que escalan bien,[14](#)
- escalada
- habilitado por la consistencia en el código base,[147](#)
 - impacto de la escala en el diseño de Code Search,
[359-361](#)
- programación, automatizado,[519](#),[524](#)
- alcance de las pruebas,[219-221](#),[281](#)
- definir el alcance de una unidad,[237](#)
 - prueba más pequeña posible,[289](#)
- scoped_ptr en C++,[467](#) marcando un cambio,[413](#) costuras,[260](#)
- índice de búsqueda en Code Search,[361](#) latencia de consulta de búsqueda, búsqueda de código y,[359](#) seguridad
- de dependencias externas,[396](#) reaccionar ante amenazas y vulnerabilidades,[10](#) riesgos introducidos por dependencias externas,
[385](#)
- datos sembrados,[294](#)
- buscadores (de documentación),
[191](#) auto confianza,[36](#)
- equipo autónomo, edificio,[112-116](#)

- cadenas de versión semántica,[394](#)
versionado semántico,[440](#)
limitaciones de,[443-449](#)
 Selección de versión mínima,[447](#)
 motivaciones,[446](#)
 sobrelimita,[444](#)
 compatibilidad demasiado prometedora,[445](#) preguntando si funciona,[448](#) SemVer (ver versión semántica)
liderazgo de servicio,[85](#)
sin servidor,[539-543](#)
 acerca de,[540](#)
 pros y contras de,[541](#) marcos sin servidor,[541](#)
 compensación,[542](#)
servicios, conectándose a software para gestión calcular,[528](#)
sirviendo trabajos,[526](#)
 multiusuario para,[534](#)
sombreado (en Java),[342](#)
fragmentación y envío en el proceso LSC, [474-477](#)
entorno compartido SUT,[291](#) scripts de shell, usando para compilaciones,[373](#) cambiando a la izquierda,[17,32](#)
 tomar decisiones basadas en datos antes,[512](#)
envío solo lo que se usa,[511](#) señales definido,[129](#)
Marco de objetivos/señales/métricas (GSM), [129](#)
punto único de falla (SPOF),[44](#)
 líder como,[112](#)
IVU de una sola máquina,[291](#)
SUT de un solo proceso,[290](#)
pequeñas correcciones en el código base con LSC,[462](#)
pequeñas pruebas,[216,231,281](#) interacción social siendo "Googley",[41](#)
entrenar a un jugador de bajo rendimiento,[90](#)
patrones de interacción grupal,[47](#) humildad, respeto y confianza en la práctica, [36-39](#)
pilares de,[34](#)
 por qué importan los pilares,[35](#)
habilidades sociales,[29](#)
costos sociales,[18](#)
Ingeniería de software código inteligente y,[10](#)
Pensamientos concluyentes,[549-550](#) desprecio y,[311](#)
programación versus,[3,23](#)
 sistemas de control de versiones y,[329](#)
 escala y eficiencia,[11-17](#) el tiempo y el cambio,[6-11](#) compensaciones y costos,[18-23](#) ingenieros de software
revisones de código y,[170](#)
oficinas para,[32](#)
fuente de control
 gestión de la dependencia y,[430](#) Git como sistema dominante,[333](#)
 trasladar la documentación a,[189](#)
fuente de verdad,[334-336](#)
 Una Versión como única fuente de verdad,[340](#)
 escenario, ninguna fuente clara de verdad,[335](#) obra en curso y sucursales,[336](#) solución dispersa de n-gramas, índice de búsqueda en Código Buscar,[362](#)
velocidad en los sistemas de construcción, [371](#) pruebas de aceleración,[305](#) contratos de nube de primavera,[293](#)
marcos de pila, integración de búsqueda de código en,[357](#)
lanzamientos por etapas,[512](#)
estandarización, falta de, en pruebas más grandes,[285](#)
pruebas estatales,[238](#)
 prefiriendo las pruebas de interacción,[275](#)
estado, gestión,[527](#)
funciones de cambio de estado, [277](#) análisis estático,[417-428](#)
 efectivo, características de,[418-419](#)
 escalabilidad,[418](#)
 usabilidad,[418](#)
ejemplos de,[417](#)
hacer que funcione, lecciones clave en,[419-421](#)
empoderar a los usuarios para que contribuyan,[420](#) centrarse en la felicidad del desarrollador,[419](#) haciendo que el análisis estático sea parte del desarrollo central
 flujo de trabajo operativo,[420](#)
plataforma tricorder,[421-427](#)
 análisis durante la edición y navegación código,[427](#)
 integración del compilador,[426](#) canales de retroalimentación integrados,[423](#)
 herramientas integradas,[422](#)
 personalización por proyecto,[424](#)
 presupone,[425](#)
 correcciones sugeridas,[424](#)

- herramientas de análisis estático,[61](#)
para la corrección del código,[172](#)
modelo de dependencia estática,[439](#)
std::unique_ptr en C++,[153](#),[468](#) efecto
de farola,[129](#) pruebas de estrés,[297](#)
- golpeando,[263](#),[272](#)-[275](#)
uso apropiado de,[275](#) peligros del
uso excesivo,[273](#) stumblers,
documentación para,[192](#) árbitros de
estilo,[156](#)
guías de estilo para código,[59](#),[141](#)
ventajas de tener reglas,[142](#)
aplicando las reglas,[158](#)-[163](#)
categorías de reglas en,[151](#)
construcción de reglas en consistencia,[153](#)
reglas que hacen cumplir las mejores
prácticas,[152](#) reglas para evitar el peligro,
[151](#) temas no tratados,[153](#) cambiando las
reglas,[154](#)-[157](#)
haciendo excepciones a las reglas,[156](#)
proceso para,[155](#)
árbitros de estilo,[156](#)
creando las reglas,[143](#)-[151](#)
principios rectores,[143](#)-[151](#) para cada
lenguaje de programación,[141](#) guía de
programación,[157](#) búsqueda de
subcadenas,[369](#)
Subversión,[332](#)
éxito, ciclo de,[116](#)
solución basada en matriz de sufijos, índice de búsqueda en
Búsqueda de código,[362](#)
recuperación suplementaria,[365](#)
sostenibilidad
código base,[12](#)
bifurcación y,[22](#)
para software,[4](#)
pruebas del sistema,[219](#)
sistemas bajo prueba (SUT),[290](#)-[294](#)
tratar con servicios dependientes pero subsidiarios
helados,[293](#)
ejemplos de,[290](#)
fidelidad de las pruebas al comportamiento de,[282](#)
en la prueba funcional de binarios que interactúan,[297](#)
pruebas más grandes para,[288](#)
producción vs. SUT herméticos aislados,[305](#) reducir
el tamaño en los límites de prueba de problemas,
[292](#)
- riesgos de las pruebas en producción y Webdriver
Torso,[292](#)
alcance de, alcance de la prueba y,[289](#)
sembrando el estado SUT,[294](#)
verificación del comportamiento,[295](#)
- T**
- TAP (consulte Plataforma de automatización de pruebas)
sistemas de compilación basados en tareas,[376](#)-[380](#)
lado oscuro de,[378](#)
dificultad para mantener y depurar compilación
guiones,[379](#)
dificultad de parallelizar pasos de compilación,[378](#)
dificultad de realizar compilaciones incrementales,
[379](#)
tiempo, escala y compensaciones,
[390](#) maestro y mentor, ser,[97](#) equipos
anclar la identidad de un equipo,[115](#)
ingenieros y oficinas, opiniones sobre,[32](#)
Genio Mito y,[28](#) principal,[81](#)
- (ver también liderar un equipo) ingeniería de
software como esfuerzo de equipo,
[34](#)-[42](#)
siendo "Googley",[41](#)
cultura post mortem intachable,[39](#)-[41](#)
humildad, respeto y confianza en la práctica,
[36](#)-[39](#)
pilares de la interacción social,[34](#)
por qué son importantes los pilares de interacción social,[35](#)
- líder técnico (TL),[82](#)
gerente líder de tecnología (TLM),[82](#) charlas y
clases de tecnología,[52](#) fenómeno de las
celebridades tecnológicas,[29](#) revisiones
técnicas,[199](#)
redactores técnicos, redacción de documentación,
[204](#) tiempo y velocidad (QUANTS),[131](#)
Plataforma de automatización de pruebas (TAP),[223](#),[494](#)-[496](#)
hallazgo de culpables,[495](#)
manejo de fallas,[495](#) presumir
optimización,[494](#) limitaciones
de recursos y,[496](#) probando
fragmentos de LSC,[475](#)
modelo de tren y prueba de LSC,[466](#) datos de
prueba para pruebas más grandes,[294](#) dobles de
prueba,[219](#),[257](#)-[280](#)
en google,[258](#)
ejemplo,[259](#)

- fingiendo,[269-272](#)
 impacto en el desarrollo de software,[258](#)
 pruebas de interacción,[275-280](#) marcos
 burlones,[261](#) costuras,[260](#)
- golpeando,[272-275](#) técnicas
 de uso,[262-264](#)
 fingiendo,[263](#)
 pruebas de interacción,[264](#)
 golpeando,[263](#)
- infiel,[283](#)
 usando en la prueba de interacción frágil,[238](#)
 utilizando implementaciones reales en lugar de,
[264-269](#)
 decidir cuándo usar implementaciones reales
 ción,[266-269](#)
 prefiriendo el realismo al aislamiento,[265](#)
- infraestructura de prueba,[255](#) prueba de inestabilidad,[491](#)
- alcance de la prueba (ver alcance de las pruebas)
 tamaños de prueba,[215](#)
 en la práctica,[219](#)
 grandes pruebas,[217,281](#)
 pruebas medianas,[217](#)
 propiedades comunes a todos los tamaños,[218](#)
 pequeñas pruebas,[216](#)
 alcance de la prueba y,[220](#)
 pruebas unitarias,[231](#)
- Banco de pruebas,[208](#)
 grandes trampas de,[224](#)
- tráfico de prueba,[294](#)
- comprobabilidad
 código comprobable,[260](#)
 escribir código comprobable temprano,[261](#)
- pruebas,[207-230](#)
 como barrera a los cambios atómicos,[465](#)
 a escala de Google,[223-225](#) automatizado,
 límites de,[229](#)
 automatizar para mantenerse al día con el desarrollo moderno
 opción,[210](#)
 beneficios de probar el código,[213-214](#)
 integración continua y,[480](#) pruebas
 continuas en CI,[485-487](#) diseñar un
 conjunto de pruebas,[214-223](#)
 regla de beyoncé,[221](#)
 cobertura de código,[222](#)
 ámbito de prueba,[219-221](#)
 tamaño de prueba,[215](#)
- hermético,[491](#)
- historia en google,[225-229](#)
 cultura de prueba contemporánea,[228](#) clases
 de orientación,[226](#) Programa certificado de
 prueba,[227](#) Prueba en el inodoro (TotT),[227](#) en
 infraestructura de cambio a gran escala,[471](#)
 más grandes (ver pruebas más grandes) de
 cambios a gran escala,[466-468](#) razones para
 escribir pruebas,[208-214](#)
- Servidor web de Google, historia de,[209](#) pruebas
 para falsificaciones,[272](#)
 escribir, ejecutar, reaccionar en la automatización de pruebas,
[212-213](#)
- Prueba en el inodoro (TotT),[227](#)
 pruebas
 volverse quebradizo con el uso excesivo de stubbing,
[273](#)
 cada vez menos eficaz con el uso excesivo de
 golpeando,[273](#)
 volverse confuso con el uso excesivo de stubbing,
[273](#)
 haciendo comprensible,[307](#) uso
 excesivo de stubing, ejemplo de,[274](#)
 refactorización para evitar stubing,[274](#)
 acelerando,[305](#)
- directorio de terceros,[437](#)
- hora
 decidir entre el tiempo y la escala,[22](#) en los
 sistemas de control de versiones,[329](#) pruebas
 más grandes y el paso del tiempo,[286](#) tiempo y
 cambio en proyectos de software,[6-11](#)
 aspirar a que nada cambie,[10](#)
 orden hash (ejemplo),[9](#) ley de
 Hyrum,[8](#)
 duración de los programas y,[3](#) TL
(ver líder técnico)
- Búsquedas basadas en tokens de TLM
(consulte el administrador de líderes
 tecnológicos),[368](#) cadenas de herramientas,
 uso de Bazel,[384](#) Torvalds, Linus,[28](#)
 trazabilidad, mantenimiento de métricas,[130](#) seguimiento del
 historial de cambios de código en Critique,
[414](#)
- sistemas de seguimiento para el trabajo,[119](#)
- compensaciones
 coste-beneficio,[18-23](#)
 decidir entre tiempo y escala (examen
 por favor),[22](#)
- compilaciones distribuidas (ejemplo),[20](#)

- errores en la toma de decisiones,[22](#)
marcadores de pizarra (ejemplo),[19](#) para líderes,[109](#)
en la productividad de la ingeniería,[130](#) en el estudio de caso de latencia de búsqueda web,[111](#)
clave, identificación,[109](#)
- dependencias transitivas,[392](#)
externo,[395](#)
estricto, hacer cumplir,[393](#)
- conocimientos tribales,[45](#)
- Plataforma de análisis estático Tricorder,[322, 421-427](#)
análisis durante la edición y navegación del código,
[427](#)
integración del compilador,[426](#) criterios para nuevos controles,[422](#) canales de retroalimentación integrados,[423](#)
herramientas integradas,[422](#)
personalización por proyecto,[424](#)
presuponer cheques,[425](#)
correcciones sugeridas,[424](#)
- enfoque basado en trigramas, índice de búsqueda en Code Buscar,[361](#)
- desarrollo basado en troncos,[327, 339](#)
correlación con buenos resultados técnicos,
[339](#)
- Modelo Live at Head y,[442](#)
relación predictiva entre alta organizaciones ejecutantes y,[343](#)
preguntas de control de fuente y,[429](#)
- confianza,[35](#)
siendo "Googley",[41](#)
revisiones de código y,[400](#)
practicando,[36-39](#)
tratar a su equipo como niños (antipat-
golondrina de mar),[92](#)
- confiar en tu equipo y perder el ego,[93](#)
vulnerabilidad y,[40](#)
- Biblioteca de afirmación de la verdad,
[248](#) tutoriales,[196](#)
ejemplo de un mal tutorial,[196](#) ejemplo, mal tutorial hecho mejor,[197](#)
- tú**
- UAT (prueba de aceptación del usuario),[301](#)
IU
pruebas de extremo a extremo de la interfaz de usuario del servicio en su parte posterior fin,[292](#)
en el ejemplo de SUT bastante pequeño,[288](#)
pruebas para, poco fiables y costosas,[292](#)
- pruebas inmutables,[233](#)
examen de la unidad,[231-256](#)
brechas comunes en las pruebas unitarias,[283-284](#)
problemas de configuración,[283](#)
Comportamientos emergentes y el vacío.
efecto,[284](#)
problemas que surgen bajo carga,[284](#) Comportamientos, entradas y efectos secundarios imprevistos.
efectos,[284](#)
dobles de prueba infieles,[283](#) tiempo de ejecución de las pruebas,[267](#) vida útil del software probado,[286](#)
limitaciones de las pruebas unitarias,[282](#) mantenibilidad de las pruebas, importancia de,[232](#) pruebas de alcance limitado (o pruebas unitarias),[219](#) prevención de pruebas frágiles,[233-239](#) propiedades de buenas pruebas unitarias,[285](#) pruebas y código compartido, HÚMEDO, no SECO,
- 248-255**
- prueba de humedad,[250](#)
definición de la infraestructura de prueba,[255](#)
ayudantes compartidos y validación,[254](#)
configuración compartida,[253](#)
valores compartidos,[251](#)
- escribir pruebas claras,[239-248](#)
dejando la lógica fuera de las pruebas,[246](#) hacer pruebas completas y concisas,[240](#) probar comportamientos, no métodos,[241-246](#) escribir mensajes de error claros,[247](#) unidades (en pruebas unitarias),[237](#) Unix, desarrolladores de,[28](#) construcciones irreproducibles,[385](#) actualizaciones,[4](#)
- ejemplo de actualización del compilador,[14-dieciséis](#) la vida útil de los proyectos de software y la importancia tancia de,[6](#)
- usabilidad de análisis estáticos,[418](#)
pruebas de evaluación de usuarios,[303](#)
enfoque en el usuario en CD, enviando solo lo que se usa,
[511](#)
- usuarios
ingenieros que crean software para todos los usuarios,[72](#)
centrándose primero en los usuarios más afectados por sesgo y discriminación,[78](#) relegando la consideración de los grupos de usuarios a tarde en el desarrollo,[76](#)
- V**
- efecto de vacío, pruebas unitarias y,[284](#)

validación, ayudantes compartidos y,²⁵⁴ valores versus resultados en ingeniería equitativa

En g,⁷⁷

Van Rossum, Guido,²⁸

VCS (sistemas de control de versiones),³²⁷

(ver también control de versiones)

mezcla entre repositorios de grano fino y monorrepos,³⁴⁶

temprano,³²⁹

la velocidad es un deporte de equipo,⁵⁰⁷

vender las dependencias de su proyecto,³⁹⁶

control de versiones,³²⁷⁻³³⁶

acerca de,³²⁸

en google,³⁴⁰⁻³⁴⁵

pocas ramas longevas,³⁴³

regla de una versión,^{340,342}

liberar ramas,³⁴⁴

escenario, múltiples versiones disponibles,³⁴¹

administración de sucursales,³³⁶⁻³³⁹ VCS

centralizados frente a distribuidos,³³¹⁻³³⁴ frente a la gestión de la dependencia,³³⁶ futuro de,³⁴⁶

importancia de,³²⁹⁻³³¹

monorrepos,³⁴⁵

fuente de verdad,³³⁴⁻³³⁶

máquinas virtuales (VM),⁵²⁴

para el aislamiento en el servidor de cómputo multiusuario

helados,⁵²¹

monorrepos virtuales (VMR),^{346,347} visibilidad, minimizando los módulos en el sistema de compilación

elementos,³⁹²

vulnerabilidad, mostrando,⁴⁰

W

Estudio de caso de latencia de búsqueda web,¹¹⁰⁻¹¹²

Incidente de Webdriver Torso,²⁹² pruebas de interacción bien especificadas,²⁷⁹ preguntas sobre quién, qué, cuándo, dónde y por qué,

respondiendo en la documentación,²⁰¹

espacios de trabajo

diferencias con el repositorio global,³⁶⁸ local, soporte de búsqueda de código para,³⁶²

estrecha integración entre la Crítica y,⁴⁰⁶

escribir reseñas (para documentos técnicos),¹⁹⁹

Y

YAQS ("Otro Sistema de Preguntas más"),⁵¹

Z

maestro zen, ser,⁹⁴

Sobre los autores

titο inviernοses ingeniero de software sénior en Google, donde ha trabajado desde 2010. Actualmente, es el presidente del subcomité global para el diseño de la biblioteca estándar de C++. En Google, es el líder de la biblioteca para la base de código C++ de Google: 250 millones de líneas de código que serán editadas por 12 000 ingenieros distintos en un mes. Durante los últimos siete años, Titus y sus equipos han estado organizando, manteniendo y desarrollando los componentes fundamentales de la base de código C++ de Google utilizando automatización y herramientas modernas. En el camino, ha iniciado varios proyectos de Google que se cree que se encuentran entre las 10 refactorizaciones más grandes de la historia de la humanidad. Como resultado directo de ayudar a construir herramientas de refactorización y automatización, Titus ha encontrado de primera mano una gran cantidad de atajos que los ingenieros y programadores pueden tomar para "simplemente hacer que algo funcione".

Tom Mansreckes redactor técnico del personal dentro de ingeniería de software en Google desde 2005, responsable de desarrollar y mantener muchas de las guías de programación principales de Google en infraestructura y lenguaje. Desde 2011, ha sido miembro del equipo de la biblioteca de C++ de Google, desarrollando el conjunto de documentación de C++ de Google, lanzando (con Titus Winters) las clases de capacitación de C++ de Google y documentando Abseil, el código C++ de fuente abierta de Google. Tom tiene una licenciatura en Ciencias Políticas y una licenciatura en Historia del Instituto de Tecnología de Massachusetts. Antes de Google, Tom trabajó como editor gerente en Pearson/Prentice Hall y en varias empresas emergentes.

hyrum-wrightes ingeniero de software de plantilla en Google, donde ha trabajado desde 2012, principalmente en las áreas de mantenimiento a gran escala del código base C++ de Google. Hyrum ha realizado más ediciones individuales en la base de código de Google que cualquier otro ingeniero en la historia de la empresa y lidera el grupo de herramientas de cambio automatizado de Google. Hyrum recibió un doctorado en ingeniería de software de la Universidad de Texas en Austin y también tiene una maestría de la Universidad de Texas y una licenciatura de la Universidad Brigham Young, y es un miembro docente visitante ocasional en la Universidad Carnegie Mellon. Es un orador activo en conferencias y colaborador de la literatura académica sobre mantenimiento y evolución de software.

Colofón

El animal de la portada de *Ingeniería de software en Google* es un flamenco americano (*Phoenicopterus ruber*). Esta ave se puede encontrar principalmente cerca de la costa en América Central y del Sur y el Golfo de México, aunque a veces viaja hasta el sur de Florida en los Estados Unidos. El hábitat del flamenco consiste en marismas y lagunas costeras de agua salada.

El icónico plumaje rosado del flamenco se adquiere a medida que el ave madura y proviene de los pigmentos carotenoides de su alimento. Debido a que estos pigmentos se encuentran más fácilmente en sus fuentes de alimentos naturales, los flamencos silvestres tienden a exhibir un plumaje más vibrante que sus contrapartes en cautiverio, aunque los zoológicos a veces agregan pigmentos suplementarios a sus dietas. Los flamencos suelen medir alrededor de 42 pulgadas de alto y su envergadura de punta negra se extiende aproximadamente cinco pies. Un ave zancuda, el flamenco tiene patas rosadas palmeadas de tres dedos. Aunque no hay distinciones comunes entre flamencos machos y hembras, los machos tienden a ser un poco más grandes.

Los flamencos se alimentan por filtración y usan sus largas patas y cuellos para alimentarse en aguas profundas, y pasan la mayor parte del día buscando comida. Tienen dos filas de láminas dentro de sus picos, que son cerdas en forma de peine que filtran su dieta de semillas, algas, organismos microscópicos y pequeños camarones. Los flamencos viven en grandes grupos de hasta 10,000 y migrarán cuando hayan comido toda la comida en un solo lugar. Además de ser aves sociales, los flamencos son extremadamente vocales. Tienen llamadas de ubicación para ayudar a encontrar compañeros específicos y llamadas de alarma para advertir al grupo más grande.

Aunque alguna vez se consideró parte de la misma especie que el gran flamenco (*Phoenicopterus roseus*), que se puede encontrar en África, Asia y el sur de Europa, el flamenco americano ahora se considera una especie separada. Si bien el estado de conservación actual del flamenco americano figura actualmente como de Preocupación menor, muchos de los animales en las portadas de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo.

La ilustración de la portada es de Karen Montgomery, basada en un grabado en blanco y negro de *Historia natural de Cassell*. Las fuentes de la portada son Gilroy Semibold y Guardian Sans. La fuente del texto es Adobe Minion Pro; la fuente del encabezado es Adobe Myriad Condensed; y la fuente del código es Ubuntu Mono de Dalton Maag.



O'REILLY®

hay mucho más de dónde vino esto.

Experimente libros, videos, cursos de capacitación en línea en vivo y más de O'Reilly y nuestros más de 200 socios, todo en un solo lugar.

Obtenga más información en oreilly.com/online-learning