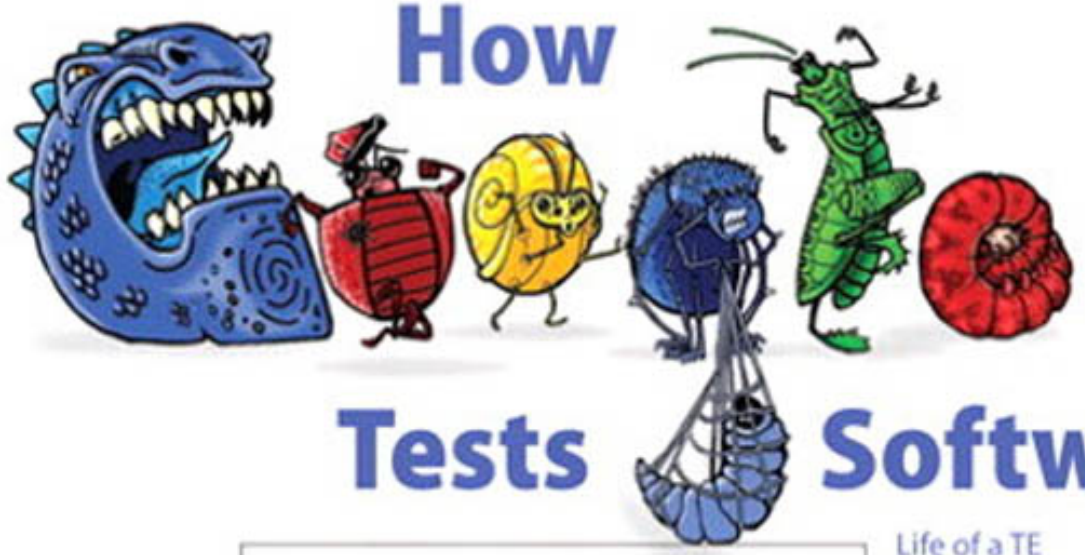

How Tests Software

Help me test like Google

Life of a TE
Life of an SET
Interviews with Googlers
and more

James Whittaker • Jason Arbon • Jeff Carollo

Praise for *How Google Tests Software*

"James Whittaker has long had the pulse of the issues that are shaping testing practice. In the decade of the Cloud Transformation, this book is a must read not just for Googlers, but for all testers who want their practices to remain relevant, competitive, and meaningful."

—Sam Guckenheimer, Product Owner,
Visual Studio Strategy, Microsoft

"Google has consistently been an innovator in the app testing space—whether it's blending test automation with manual efforts, melding in-house and outsourced resources, or more recently, pioneering the use of in-the-wild testing to complement their in-the-lab efforts. This appetite for innovation has enabled Google to solve new problems and launch better apps.

In this book, James Whittaker provides a blueprint for Google's success in the rapidly evolving world of app testing."

—Doron Reuveni, CEO and Cofounder, uTest

"This book is a game changer, from daily releases to heads-up displays. James Whittaker takes a computer-science approach to testing that will be the standard for software companies in the future. The process and technical innovations we use at Google are described in a factual and entertaining style. This book is a must read for anyone involved in software development."

—Michael Bachman, Senior Engineering Manager
at Google Inc., AdSense/Display

"By documenting much of the magic of Google's test engineering practices, the authors have written the equivalent of the Kama Sutra for modern software testing."

—Alberto Savoia, Engineering Director, Google

"If you ship code in the cloud and want to build a strategy for ensuring a quality product with lots of happy customers, you must study and seriously consider the methods in this book."

—Phil Waligora, Salesforce.com

"James Whittaker is an inspiration and mentor to many people in the field of testing. We wouldn't have the talent or technology in this field without his contributions. I am consistently in awe of his drive, enthusiasm, and humor. He's a giant in the industry and his writing should be required reading for anyone in the IT industry."

—Stewart Noakes, Chairman TCL Group Ltd.,
United Kingdom

"I worked with James Whittaker during his time at Microsoft, and although I miss having him here at Microsoft, I knew he would do great things at Google. James, Jason Arbon, and Jeff Carollo have packed this book with innovative testing ideas, practical examples, and insights into the Google testing machine. Anyone with an ounce of curiosity about Google's approach to testing and quality or with the smallest desire to discover a few new ideas in testing will find value in these pages."

—Alan Page, Microsoft Xbox, and Author
of *How We Test Software at Microsoft*

How Google Tests Software

This page intentionally left blank

How Google Tests Software

James Whittaker
Jason Arbon
Jeff Carollo

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The Library of Congress cataloging-in-publication data is on file.

ISBN-13: 978-0-321-80302-3

ISBN-10: 0-321-80302-7

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing: March 2012

Publisher
Paul Boger

Executive Editor
Chris Guzikowski

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Ginny Bess Munroe

Indexer
Erika Millen

Proofreader
Mike Henry

Editorial Assistant
Olivia Basegio

Cover Designer
Anne Jones

Compositor
Gloria Schurick

To all testers at Google, Microsoft, and elsewhere who've made me think differently.
—James A. Whittaker

To my wife Heather and my children Luca, Mateo, Dante, and Odessa who thought I worked at Starbucks all this time.
—Jason Arbon

To Mom, Dad, Lauren, and Alex.
—Jeff Carollo

This page intentionally left blank

Table of Contents

Foreword by Alberto Savoia

Writing a foreword for a book you wish you had written yourself is a dubious honor; it's a bit like serving as best man for a friend who is about to spend the rest of his life with the girl *you* wanted to marry. But James Whittaker is a cunning guy. Before asking me if I'd be willing to write this preface, he exploited my weakness for Mexican food by treating me to a very nice dinner and plying me with more than a couple Dos Equis before he "popped the question." By the time this happened, I was as malleable and agreeable as the bowl of guacamole I had just finished. "*Si senor*," was pretty much all I could say. His ploy worked and here he stands with his book as his bride and I get to make the wedding speech.

As I said, he's one cunning guy.

So here we go…a preface to the book I wish I had written myself. Cue the mushy wedding music.

Does the world really need yet another software testing book, especially yet another software testing book from the prolific James Whittaker, whom I've publicly called "the Octomom[1] of test book publishing" on more than one occasion? Aren't there enough books out there describing the same old tired testing methodologies and dishing out dubious and dated advice? Well, there are enough of those books, but this book I am afraid is not one of them. That's why I wish I had written it myself. The world actually needs this particular testing book.

The Internet has dramatically changed the way most software is designed, developed, and distributed. Many of the testing best practices, embodied in any number of once popular testing books of yesteryear, are at best inefficient, possibly ineffective, and in some cases, downright counterproductive in today's environment. Things have been moving so fast in our industry that many of the software testing books written as recently as a few years ago are the equivalent of surgery books containing advice about leeches and skull drilling to rid the body of evil spirits; it would be best to recycle them into adult diapers to make sure they don't fall into the hands of the gullible.

Given the speed at which things are evolving in the software industry, I would not be too surprised if ten years from now this book will also be obsolete. But until the paradigm shifts again, *How Google Tests Software* gives you a very timely and applicable insider's view into how one of the world's most successful and fastest growing Internet companies deals with the unique challenges of software testing in the twenty-first century. James Whittaker and his coauthors have captured the very essence of how Google is successful at testing some of the most complicated and popular

1. Don't get the Octomom reference? Google it!

software of our times. I know this is the case because I've been there through the transition.

I first joined Google as engineering director in 2001. At the time, we had about two hundred developers and…a whopping *three* testers! My developers were already taking responsibility for testing their own code, but test-driven development and test automation tools such as JUnit were just entering the scene, so our testing was mostly *ad-hoc* and dependent on the diligence of the individual writing the code. But that was okay; we were a startup and we had to move fast and take risks or we couldn't compete with the big established players.

However, as the company grew in size and our products became more mission-critical for our users and customers (such as AdWords, one of the products I was responsible for, was quickly becoming a major source of monetizing websites), it became clear that we had to increase our focus and investment in testing. With only three testers, we had no choice but to get developers more involved in testing. Along with a few other Googlers, I introduced, taught, and promoted unit testing. We encouraged developers to make testing a priority and use tools such as JUnit to automate them. But adoption was slow and not everybody was sold on the idea of developers testing their own code. To keep the momentum going, every week at the company's Friday afternoon beer bash (appropriately named TGIF), I gave out testing awards to the developers who wrote tests. It felt a lot like an animal trainer giving treats to doggies for performing a trick, but at least it drew attention to testing. Could I be so lucky that getting developers to test would be this simple?

Unfortunately, the treats didn't work. Developers realized that in order to have adequate tests, they had to write two or three lines of unit test code for every line of code under test and that those tests required at least as much maintenance as the functional code itself and had just as much chance of being buggy. It also became clear to no one's surprise that developer-unit testing was not sufficient. We still needed integration tests, system tests, UI tests, and so on. When it came to testing, we had a lot of growing up (and substantial learning) to do, and we had to do it fast. Very fast!

Why the urgency? Well, I don't believe that any amount of testing can turn a bad idea or an ill-advised product into a success. I do believe that the wrong approach to testing can kill the chances of a good product or company or, at the very least, slow down its growth and open the door for the competition. Google was at that point. Testing had become one of the biggest barriers to continued success and coming up with the right testing strategy to keep up with our ultra-rapid growth in users, products, and employees without slowing the company down involved a lot of innovative approaches, unorthodox solutions, and unique tools. Not everything

worked, of course, but in the process, we learned valuable lessons and practices that are applicable to any company that wants to grow or move at the speed of Google. We learned how to have attention to quality without derailing development or the speed at which we got things done. The resulting process, with some insights into the thinking behind them and what makes them work, is what this book is about. If you want to understand how Google met the challenges of testing in the twenty-first century on modern Internet, mobile, and client applications, then you have come to the right place. I may wish it was me who was telling the rest of the story, but James and his coauthors beat me to it and they have nailed the essence of what testing is like here at Google.

One final note on the book: James Whittaker is the guy who made this book happen. He came to Google, dug in to the culture, took on big and important projects, and shipped products such as Chrome, Chrome OS, and dozens of smaller ones. Somewhere in that time, he became the public face of Google testing. But, unlike some of his other books, much of this material is not his. He is as much a reporter on the evolution of how Google tests software as he is a contributor to it. Keep that in mind as you read it because James will probably try to take all the credit for himself!

As Google grew from 200 to over 20,000 employees, there were many people who played important roles in developing and putting into action our testing strategy. James credits many of them and they have contributed directly by writing sidebars and giving interviews that are published in this book. However, no one, not me, James, or anyone else mentioned in this book, has had as much influence as Patrick Copeland, the architect of our current organizational structure and leader of Google's Engineering Productivity team. Every tester in the company reports up through Patrick and he is the executive whose vision created what James has documented and contributed to here. If anyone can take credit for how Google tests software today, it's Patrick. I am not just saying this because he's my boss; I am saying it because he's my boss *and* he told me to say it!

Alberto Savoia is an engineering director and innovation agitator at Google. He first joined Google in 2001 when, among other things, he managed the launch of Google AdWords and played a key role in kick-starting a developer / unit testing culture in the company. He is also the author of *The Way of Testivus* and of "Beautiful Tests" in O'Reilly's *Beautiful Code*.

Note by James Whittaker: I couldn't agree more! As a scribe and journalist in this process, I owe most of the material to the organization that Patrick has created. And I am not just saying this because he gave me permission to write this book. As my boss, he made me write this book!

This page intentionally left blank

Foreword by Patrick Copeland

My adventure with Google started in March of 2005. If you read Alberto's foreword, you know a bit about the conditions of Google around that time. It was small but beginning to show signs of serious growing pains. It was also a time of rapid technological change as the web world was welcoming dynamic content and the cloud was becoming a viable alternative to the then dominant world of client-server.

That first week, I sat with the rest of the Nooglers topped with a tri-colored propeller hat listening to the founders discussing corporate strategy at a weekly company meeting called TGIF. I knew little about what I had gotten myself into. I was naïve enough to be excited and aware enough to be more than a little intimated. The speed and scale of Google made my previous decade of 5-year ship cycles seem like poor preparation. Worse still, I think I was the only tester wearing one of those Noogler hats. Surely there were more of us somewhere!

I joined Google when engineering was just shy of 1,000 people. The testing team had about 50 full timers and some number of temporary workers I never could get my head around. The team was called "Testing Services" and focused the majority of its energy on UI validation and jumping into projects on an as-needed basis. As you might imagine, it wasn't exactly the most glorified team at Google.

But at that time it was enough. Google's primary businesses were Search and Ads. The Google world was much smaller than it is today, and a thorough job of exploratory testing was enough to catch most quality concerns. But the world was slowly changing. Users were hitting the web in unprecedented numbers and the document-based web was giving way to the app-based web. You could feel the inevitability of growth and change where the ability to scale and get to market quickly would be the difference between relevance and…a place you did not want to be.

Inside Google, the scale and complexity issues were buckling Testing Services. What worked well with small homogenous projects was now burning out good testers as they leapt from one project that was on fire to the next. And topping all that off was Google's insistence to release quicker. Something needed to give and I had the choice between scaling this manually intensive process or changing the game completely. Testing Services needed a radical transformation to match the radical change happening to the industry and the rest of Google.

I would very much like to say that I drew upon my vast wealth of experience and conjured the perfect test organization, but the truth is that my experience had taught me little more than the *wrong way to do things*. Every test organization I ever worked as part of or led was dysfunctional in one way or the other. Stuff was always broken. The code was broken, the

tests were broken, and the team was broken! I knew what it meant to be buried under quality and technical debt where every innovative idea was squashed lest it risk breaking the fragile product it depended on. If my experience taught me anything, it was how *not* to do testing.

In all my interactions up to this point, one thing about Google was clear. It respected computer science and coding skill. Ultimately, if testers were to join this club, they would have to have good computer science fundamentals and some coding prowess. First-class citizenship demanded it.

If I was going to change testing at Google, I needed to change what it meant to be a tester. I used to try to imagine the perfect team and how such a team would shoulder the quality debt and I kept coming back to the same premise: The only way a team can write quality software is when the entire team is responsible for quality. That meant product managers, developers, testers…everyone. From my perspective, the best way to do this was to have testers capable of making *testing* an actual feature of the code base. The testing feature should be equal to any feature an actual customer might see. The skill set I needed to build features was that of a developer.

Hiring testers who can code features is difficult; finding feature developers who can test is even more difficult. But the status quo was worse than either so I forged ahead. I wanted testers who could do more for their products and at the same time, I wanted to evolve the nature and ownership of the testing work, which meant asking for far larger investment from the development teams. This is the one organizational structure I had yet to see implemented in all my time in the industry and I was convinced it was right for Google, and I thought that as a company, we were ready for it.

Unfortunately, few others in the company shared my passion for such profound and fundamental change. As I began the process of socializing my equal-but-different vision for the software testing role, I eventually found it difficult to find a lunch partner! Engineers seemed threatened by the very notion that they would have to play a bigger role in testing, pointing out "that's what test is for." Among testers, the attitude was equally unsavory as many had become comfortable in their roles and the status quo had such momentum that change was becoming a very hard problem.

I kept pressing the matter mostly out of fear that Google's engineering processes would become so bogged down in technical and quality debt that I'd be back to the same five-year ship cycles I had so happily left behind in the old client-server world. Google is a company of geniuses focused on innovation and that entrepreneurial makeup is simply incompatible with long product ship cycles. This was a battle worth fighting and I convinced myself that once these geniuses understood the idea of development and testing practices for a streamlined and repeatable "technology factory," they would come around. They would see that we were not a startup anymore and with our rapidly growing user base and increasing technical debt of bugs and poorly structured code would mean the end of their coder's playground.

I toured the product teams making my case and trying to find the sweet spot for my argument. To developers, I painted a picture of continuous builds, rapid deployment, and a development process that moved quickly, leaving more time for innovation. To testers, I appealed to their desire to be full engineering partners of equal skill, equal contribution, and equal compensation.

Developers had the attitude that if we were going to hire people skilled enough to do feature development, then we should have them do feature development. Some of them were so against the idea that they filled my manager's inbox with candid advice on how to deal with my madness. Fortunately, my manager ignored those recommendations.

Testers, to my surprise, reacted similarly. They were vested in the way things were and quick to bemoan their status, but slow to do anything about it.

My manager's reaction to the complaints was telling: "This is Google, if you want something done, then do it."

And so that's what I did. I assembled a large enough cadre of like-minded folks to form interview loops and we began interviewing candidates. It was tough going. We were looking for developer skills and a tester mindset. We wanted people who could code and wanted to apply that skill to the development of tools, infrastructure, and test automation. We had to rethink recruiting and interviewing and then explain that process to the hiring committees who were entrenched and comfortable with the way things were.

The first few quarters were rough. Good candidates were often torpedoed in the vetting process. Perhaps they were too slow to solve some arcane coding problem or didn't fare well in something that someone thought was important but that had nothing to do with testing skill. I knew hiring was going to be difficult and made time each week to write hiring justification after hiring justification. These went to none other than Google co-founder Larry Page who was (and still is) the last approval in the hiring process. He approved enough of them that my team began to grow. I often wonder if every time Larry hears my name he still thinks, "Hiring testers!"

Of course, by this time, I had made enough noise trying to get buy-in that we had no choice but to perform. The entire company was watching and failure would have been disastrous. It was a lot to expect from a small test team supported by an ever-changing cast of vendors and temps. But even as we struggled to hire and I dialed back the number of temps we used, I noticed change was taking hold. The more scarce testing resources became, the more test work was left for developers to do. Many of the teams rose to the challenge. I think if technology had stayed the same as it was, this alone would have taken us nearly where we needed to be.

But technology wasn't standing still and the rules of development and testing were changing rapidly. The days of static web content were gone. Browsers were still trying to keep up. Automation around the browser was a year behind the already tardy browsers. Making testing a development

problem at the same time those same developers were facing such a huge technology shift seemed a fool's errand. We lacked the ability to properly test these applications manually, much less with automation.

The pressure on the development teams was just as bad. Google began buying companies with rich and dynamic web applications. YouTube, Google Docs, and so on stretched our internal infrastructure. The problems I was facing in testing were no more daunting than the problems the development teams were facing in writing code for me to test! I was trying to solve a testing problem that simply couldn't be solved in isolation. Testing and development, when seen as separate disciplines or even as distinct problems, was wrong-headed and continuing down such a path meant we would solve neither. Fixing the test team would only get us an incremental step forward.

Progress was happening. It's a funny thing about hiring smart people: They tend to make progress! By 2007, the test discipline was better positioned. We were managing the endgame of the release cycle well. Development teams knew they could count on us as partners to production. But our existence as a late-in-the-cycle support team was confining us to the traditional QA model. Despite our ability to execute well, we were still not where I wanted us to be. I had a handle on the hiring problem and testing was moving in the right direction, but we were engaged too late in the process.

We had been making progress with a concept we called "Test Certified" (which the authors explain in some detail later in this book) where we consulted with dev teams and helped them get better code hygiene and unit testing practices established early. We built tools and coached teams on continuous integration so that products were always in a state that made them testable. There were countless small improvements and tweaks, many of them detailed in this book, that erased much of the earlier skepticism. Still, there was a lack of identity to the whole thing. Dev was still dev; test was still test. Many of the ingredients for culture change were present, but we needed a catalyst for getting them to stick.

As I looked around the organization that had grown from my idea to hire developers in a testing role, I realized that testing was only part of what we did. We had tool teams building everything from source repositories to building infrastructure to bug databases. We were test engineers, release engineers, tool developers, and consultants. What struck me was just how much the nontesting aspect of our work was impacting productivity. Our name may have been Testing Services, but our charter was so much more.

So I decided to make it official and I changed the name of the team to Engineering Productivity. With the name change also came a cultural adjustment. People began talking about productivity instead of testing and quality. Productivity is our job; testing and quality are the job of everyone involved in development. This means that developers own testing and

developers own quality. The productivity team is responsible for enabling development to nail those two things.

In the beginning, the idea was mostly aspirational and our motto of "accelerating Google" may have rung hollow at first, but over time and through our actions, we delivered on these promises. Our tools enabled developers to go faster and we went from bottleneck to bottleneck clearing the clogs and solving the problems developers faced. Our tools also enabled developers to write tests and then see the result of those tests on build after build. Test cases were no longer executed in isolation on some tester's machine. Their results were posted on dashboards and accumulated over successive versions so they became part of the public record of the application's fitness for release. We didn't just demand developers get involved; we made it easy for them to do so. The difference between productivity and testing had finally become real: Google could innovate with less friction and without the accumulation of technical debt.

And the results? Well, I won't spoil the rest of the book because that's why it was written. The authors took great pains to scour their own experiences and those of other Googlers to boil our secret sauce down to a core set of practices. But we were successful in many ways—from orders of magnitude, decreases in build times, to run-it-and-forget-it test automation, to open sourcing some truly innovative testing tools. As of the writing of this preface, the Productivity Team is now about 1,200 engineers or a bit larger than all of Google Engineering in 2005 when I joined. The productivity brand is strong and our charter to accelerate Google is an accepted part of the engineering culture. The team has travelled light years from where we were on my first day sitting confused and uncertain at TGIF. The only thing that hasn't changed since that day is my tri-colored propeller hat, which sits on my desk serving as a reminder of how far we've come.

Patrick Copeland is the senior director of Engineering Productivity and the top of the testing food chain at Google. All testers in the company report up through Patrick (whose skip-level manager, incidentally, is Larry Page, Google's Co-founder and CEO). Patrick's career at Google was preceded by nearly a decade at Microsoft as a director of Test. He's a frequent public speaker and known around Google as the architect of Google's technology for rapid development, testing, and deployment of software.

This page intentionally left blank

Preface

Software development is hard. Testing that software is hard, too. And when you talk about development and testing at the scale of the entire web, you are talking about Google. If you are interested in how one of the biggest names in the Internet handles such large-scale testing, then you have found the right book.

Google tests and releases hundreds of millions of lines of code distributed across millions of source files daily. Billions of build actions prompt millions of automated tests to run across hundreds of thousands of browser instances daily. Operating systems are built, tested, and released within the bounds of a single calendar year. Browsers are built daily. Web applications are released at near continuous pace. In 2011, 100 features of Google+ were released over a 100-day period.

This is Google scale and Google speed—the scale of the Web itself—and this is the testing solution described in this book. We reveal how this infrastructure was conceived, implemented, and maintained. We introduce the reader to the many people who were instrumental in developing both the concepts and the implementation and we describe the infrastructure that it takes to pull it off.

But it wasn't always this way. The route Google took to get where it is today is just as interesting as the technology we use to test. Turn the clock back six years and Google was more like other companies we once worked for: Test was a discipline off the mainstream. Its practitioners were under-appreciated and over-worked. It was a largely manual process and people who were good at automating it were quickly absorbed into development where they could be more "impactful." The founding team of what Google calls "Engineering Productivity" has to overcome bias against testing and a company culture that favored heroic effort over engineering rigor. As it stands today, Google testers are paid on the same scale as developers with equivalent bonuses and promotion velocity. The fact that testers succeeded and that this culture has lived on through significant company growth (in terms of products, variety, and revenue) and structural reorganizations should be encouraging to companies following in Google's footsteps. Testing can be done right and it can be appreciated by product teams and company executives alike.

As more and more companies find their fortunes and futures on the Web, testing technology and organizational structure as described in this book might become more prevalent. If so, consider this the playbook for how to get there.

This Google testing playbook is organized according to the roles involved. In the first section, we discuss all the roles and introduce all the

concepts, processes, and intricacies of the Google quality processes. This section is a must read.

The chapters can be read in any order whatsoever. We first cover the SET or software engineer in test role because this is where modern Google testing began. The SET is a technical tester and the material in that chapter is suitably technical but at a high enough level that anyone can grasp the main concepts. The SET chapter is followed by a chapter covering the other primary testing role, that of the TE or test engineer. This is a big chapter because the TE's job is broad and Google TEs cover a lot of ground during the product cycle. This is a role many traditional testers find familiar, and we imagine it will be the most widely read section of the book because it applies to the broadest practitioner audience.

The balance of the book is about test management and interviews with key Googlers who either played a part in the history of Google test or are key players on key Google products. These interviews will likely be of interest to anyone who is trying to develop Google-like testing processes or teams.

There is a final chapter that really should not be missed by any interested reader. James Whittaker gives insight into how Google testing is still evolving and he makes some predictions about where Google and the industry at large are heading test-wise. We believe many readers will find it insightful and some might even find it shocking.

A NOTE ABOUT FIGURES

Due to the complexity of the topics discussed and the graphical nature of the Internet, some figures from Chapter 3 in this book are very detailed and are intended only to provide a high-level view of concepts. Those figures are representational and not intended to be read in detail. If you prefer to view these figures on your computer, you can download them at www.informit.com/title/9780321803023.

A Word About This Book

When Patrick Copeland originally suggested that I write this book, I was hesitant to do so, and my reasons all turned out to be good ones. People would question whether I was the best Googler to write it. (They did.) Too many people would want to get involved. (This also turned out to be true.) But mostly, it was because all my prior books were written for beginners. Both the *How to Break…*series and my *Exploratory Testing* book can be told as a complete, end-to-end story. Not so with this book. Readers might well read this book in a single sitting, but it's meant as more of a reference for how Google actually performs the tasks, both small ones and big ones, that make up our practice of testing. I expect people who have tested software in a corporate environment will get more out of it than beginners because they will have a basis for comparing our Google processes to the ones they are used to. I picture experienced testers, managers, and executives picking it up, finding a topic of interest, and reading that section to see how Google performs some specific task. This is not a writing style I have often assumed!

Two heretofore unpublished co-authors have joined me in this endeavor. Both are excellent engineers and have been at Google longer than I have been. Jason Arbon's title is a test engineer, but he's an entrepreneur at heart and his impact on many of the ideas and tools that appear in the test engineer chapters of this book has been profound. Having our careers intersect has changed us both. Jeff Carollo is a tester turned developer and is categorically the best test developer I have ever met. Jeff is one of the few people I have seen succeed at "walk away automation"—which is test code written so well and so self contained that the author can create it and leave it to the team to run without intervention. These two are brilliant and we tried hard to make this book read with one voice.

There are any number of guest Googlers who have supplied material. Whenever the text and subject matter is the work of a single author, that person is identified in the heading that precedes his work. There are also a number of interviews with key Googlers who had profound impact on the way we do testing. It was the best way we could think of to include as many of the people who defined Google testing without having a book

with 30 authors! Not all readers will be interested in all the interviews, but they are clearly marked in the text so they can be skipped or read individually. We thank all these contributors profusely and accept any blame if our ability to do justice to their work fell short in any way. The English language is a poor medium to describe sheer brilliance.

Happy reading, happy testing, and may you always find (and fix) the bug you are looking for.

James Whittaker
Jason Arbon
Jeff Carollo
Kirkland, Washington

Acknowledgments

We want to acknowledge every engineer at Google who has worked tirelessly to improve the quality. We also want to mention our appreciation for the open and distributed culture of Google engineering and management, which allowed us to treat our testing methodologies and practices much as we build products—with experimentation and the freedom to explore.

We'd like to specifically mention the following people who spent their energy and took risks to push testing into the cloud: Alexis O. Torres, Joe Muharksy, Danielle Drew, Richard Bustamante, Po Hu, Jim Reardon, Tejas Shah, Julie Ralph, Eriel Thomas, Joe Mikhail, and Ibrahim El Far. Also, thanks to our editors, Chris Guzikowski and Chris Zahn, who politely tolerated our engineering-speak. Thanks to interviewees who shared their views and experiences: Ankit Mehta, Joel Hynoski, Lindsay Webster, Apple Chow, Mark Striebeck, Neal Norwitz, Tracy Bialik, Russ Rufer, Ted Mao, Shelton Mar, Ashish Kumar, Sujay Sahni, Brad Green, Simon Stewart, and Hung Dang. A special thanks goes to Alberto Savoia for the inspiration to prototype and iterate quickly. Thanks to Google and the cafeteria staff and chefs for the great food and coffee. Thanks to candid feedback from Phil Waligora, Alan Page, and Michael Bachman. Ultimately, thanks to Pat Copeland for assembling and funding such an energetic and talented set of engineers who are focused on quality.

This page intentionally left blank

About the Authors

James Whittaker is an engineering director at Google and has been responsible for testing Chrome, maps, and Google web apps. He used to work for Microsoft and was a professor before that. James is one of the best-known names in testing the world over.

Jason Arbon is a test engineer at Google and has been responsible for testing Google Desktop, Chrome, and Chrome OS. He also served as development lead for an array of open-source test tools and personalization experiments. He worked at Microsoft prior to joining Google.

Jeff Carollo is a software engineer in test at Google and has been responsible for testing Google Voice, Toolbar, Chrome, and Chrome OS. He has consulted with dozens of internal Google development teams helping them improve initial code quality. He converted to a software engineer in 2010 and leads development of Google+ APIs. He also worked at Microsoft prior to joining Google."

This page intentionally left blank

CHAPTER 1
Introduction to Google Software Testing

James Whittaker

There is one question I get more than any other. Regardless of the country I am visiting or the conference I am attending, this one question never fails to surface. Even Nooglers ask it as soon as they emerge from new-employee orientation: "*How does Google test software*?"

I am not sure how many times I have answered that question or even how many different versions I have given, but the answer keeps evolving the longer time I spend at Google and the more I learn about the nuances of our various testing practices. I had it in the back of my mind that I would write a book and when Alberto, who likes to threaten to turn testing books into adult diapers to give them a reason for existence, actually suggested I write such a book, I knew it was going to happen.

Still, I waited. My first problem was that I was not the right person to write this book. There were many others who preceded me at Google and I wanted to give them a crack at writing it first. My second problem was that I as test director for Chrome and Chrome OS (a position now occupied by one of my former directs) had insights into only a slice of the Google testing solution. There was so much more to Google testing that I needed to learn.

At Google, software testing is part of a centralized organization called Engineering Productivity that spans the developer and tester tool chain, release engineering, and testing from the unit level all the way to exploratory testing. There are a great deal of shared tools and test infrastructure for web properties such as search, ads, apps, YouTube, and everything else we do on the Web. Google has solved many of the problems of speed and scale and this enables us, despite being a large company, to release software at the pace of a start-up. As Patrick Copeland pointed out in his preface to this book, much of this magic has its roots in the test team.

At Google, software testing is part of a centralized organization called Engineering Productivity.

When Chrome OS released in December 2010 and leadership was successfully passed to one of my directs, I began getting more heavily involved in other products. That was the beginning of this book, and I tested the waters by writing the first blog post,[1] "How Google Tests Software," and the rest is history. Six months later, the book was done and I wish I had not waited so long to write it. I learned more about testing at Google in the last six months than I did my entire first two years, and Nooglers are now reading this book as part of their orientation.

This isn't the only book about how a big company tests software. I was at Microsoft when Alan Page, BJ Rollison, and Ken Johnston wrote *How We Test Software at Microsoft* and lived first-hand many of the things they wrote about in that book. Microsoft was on top of the testing world. It had elevated test to a place of honor among the software engineering elite. Microsoft testers were the most sought after conference speakers. Its first director of test, Roger Sherman, attracted test-minded talent from all over the globe to Redmond, Washington. It was a golden age for software testing.

And the company wrote a big book to document it all.

I didn't get to Microsoft early enough to participate in that book, but I got a second chance. I arrived at Google when testing was on the ascent. Engineering Productivity was rocketing from a couple of hundred people to the 1,200 it has today. The growing pains Pat spoke of in his preface were in their last throes and the organization was in its fastest growth spurt ever. The Google testing blog was drawing hundreds of thousands of page views every month, and GTAC[2] had become a staple conference on the industry testing circuit. Patrick was promoted shortly after my arrival and had a dozen or so directors and engineering managers reporting to him. If you were to grant software testing a renaissance, Google was surely its epicenter.

This means the Google testing story merits a big book, too. The problem is, I don't write big books. But then, Google is known for its simple and straightforward approach to software. Perhaps this book is in line with that reputation.

How Google Tests Software contains the core information about what it means to be a Google tester and how we approach the problems of scale, complexity, and mass usage. There is information here you won't find anywhere else, but if it is not enough to satisfy your craving for how we test, there is more available on the Web. Just Google it!

There is more to this story, though, and it must be told. I am finally ready to tell it. The way Google tests software might very well become the way many companies test as more software leaves the confines of the desktop for the freedom of the Web. If you've read the Microsoft book, don't expect to find much in common with this one. Beyond the number of

---

1. http://googletesting.blogspot.com/2011/01/how-google-tests-software.html.

2. GTAC is the Google Test Automation Conference (www.GTAc.biz).

authors—both books have three—and the fact that each book documents testing practices at a large software company, the approaches to testing couldn't be more different.

The way Google tests software might very well become the way many companies test as more software leaves the confines of the desktop for the freedom of the Web.

Patrick Copeland dealt with how the Google methodology came into being in his preface to this book and since those early days, it has continued to evolve organically as the company grew. Google is a melting pot of engineers who used to work somewhere else. Techniques proven ineffective at former employers were either abandoned or improved upon by Google's culture of innovation. As the ranks of testers swelled, new practices and ideas were tried and those that worked in practice at Google became part of Google and those proven to be baggage were jettisoned. Google testers are willing to try anything once but are quick to abandon techniques that do not prove useful.

Google is a company built on innovation and speed, releasing code the moment it is useful (when there are few users to disappoint) and iterating on features with early adopters (to maximize feedback). Testing in such an environment has to be incredibly nimble and techniques that require too much upfront planning or continuous maintenance simply won't work. At times, testing is interwoven with development to the point that the two practices are indistinguishable from each other, and at other times, it is so completely independent that developers aren't even aware it is going on.

At times, testing is interwoven with development to the point that the two practices are indistinguishable from each other, and at other times, it is so completely independent that developers aren't even aware it is going on.

Throughout Google's growth, this fast pace has slowed only a little. We can nearly produce an operating system within the boundaries of a single calendar year; we release client applications such as Chrome every few weeks; and web applications change daily—all this despite the fact that our start-up credentials have long ago expired. In this environment, it is almost easier to describe what testing is not—dogmatic, process-heavy, labor-intensive, and time-consuming—than what it is, although this book is an attempt to do exactly that. One thing is for sure: Testing must not create friction that slows down innovation and development. At least it will not do it twice.

Google's success at testing cannot be written off as owing to a small or simple software portfolio. The size and complexity of Google's software testing problem are as large as any company's out there. From client operating systems, to web apps, to mobile, to enterprise, to commerce and social,

Google operates in pretty much every industry vertical. Our software is big; it's complex; it has hundreds of millions of users; it's a target for hackers; much of our source code is open to external scrutiny; lots of it is legacy; we face regulatory reviews; our code runs in hundreds of countries and in many different languages, and on top of this, users expect software from Google to be simple to use and to "just work." What Google testers accomplish on a daily basis cannot be credited to working on easy problems. Google testers face nearly every testing challenge that exists every single day.

Whether Google has it right (probably not) is up for debate, but one thing is certain: The approach to testing at Google is different from any other company I know, and with the inexorable movement of software away from the desktop and toward the cloud, it seems possible that Google-like practices will become increasingly common across the industry. It is my and my co-authors' hope that this book sheds enough light on the Google formula to create a debate over just how the industry should be facing the important task of producing reliable software that the world can rely on. Google's approach might have its shortcomings, but we're willing to publish it and open it to the scrutiny of the international testing community so that it can continue to improve and evolve.

Google's approach is more than a little counterintuitive: We have fewer dedicated testers in our entire company than many of our competitors have on a single product team. Google Test is no million-man army. We are small and elite Special Forces that have to depend on superior tactics and advanced weaponry to stand a fighting chance at success. As with military Special Forces, it is this scarcity of resources that forms the base of our secret sauce. The absence of plenty forces us to get good at prioritizing, or as Larry Page puts it: "Scarcity brings clarity." From features to test techniques, we've learned to create high impact, low-drag activities in our pursuit of quality. Scarcity also makes testing resources highly valued, and thus, well regarded, keeping smart people actively and energetically involved in the discipline. The first piece of advice I give people when they ask for the keys to our success: Don't hire too many testers.

The first piece of advice I give people when they ask for the keys to our success: Don't hire too many testers.

How does Google get by with such small ranks of test folks? If I had to put it simply, I would say that at Google, the burden of quality is on the shoulders of those writing the code. Quality is never "some tester's" problem. Everyone who writes code at Google is a tester, and quality is literally the problem of this collective (see Figure 1.1). Talking about dev to test ratios at Google is like talking about air quality on the surface of the sun. It's not a concept that even makes sense. If you are an engineer, you are a tester. If you are an engineer with the word test in your title, then you are an enabler of good testing for those other engineers who do not.

FIGURE 1.1 Google engineers prefer quality over features.

The fact that we produce world-class software is evidence that our particular formula deserves some study. Perhaps there are parts of it that will work in other organizations. Certainly there are parts of it that can be improved. What follows is a summary of our formula. In later chapters, we dig into specifics and show details of just how we put together a test practice in a developer-centric culture.

Quality ≠ Test

"Quality cannot be tested in" is so cliché it has to be true. From automobiles to software, if it isn't built right in the first place, then it is never going to *be* right. Ask any car company that has ever had to do a mass recall how expensive it is to bolt on quality after the fact. Get it right from the beginning or you've created a permanent mess.

However, this is neither as simple nor as accurate as it sounds. Although it is true that quality cannot be tested in, it is equally evident that without testing, it is impossible to develop anything of quality. How does one decide if what you built is high quality without testing it?

The simple solution to this conundrum is to stop treating development and test as separate disciplines. Testing and development go hand in hand. Code a little and test what you built. Then code some more and test some more. Test isn't a separate practice; it's part and parcel of the development process itself. Quality is not equal to test. Quality is achieved by putting

development and testing into a blender and mixing them until one is indistinguishable from the other.

Quality is not equal to test. Quality is achieved by putting development and testing into a blender and mixing them until one is indistinguishable from the other.

At Google, this is exactly our goal: to merge development and testing so that you cannot do one without the other. Build a little and then test it. Build some more and test some more. The key here is *who* is doing the testing. Because the number of actual dedicated testers at Google is so disproportionately low, the only possible answer has to be the developer. Who better to do all that testing than the people doing the actual coding? Who better to find the bug than the person who wrote it? Who is more incentivized to avoid writing the bug in the first place? The reason Google can get by with so few dedicated testers is because developers own quality. If a product breaks in the field, the first point of escalation is the developer who created the problem, not the tester who didn't catch it.

This means that quality is more an act of prevention than it is detection. Quality is a development issue, not a testing issue. To the extent that we are able to embed testing practice inside development, we have created a process that is hyper-incremental where mistakes can be rolled back if any one increment turns out to be too buggy. We've not only prevented a lot of customer issues, we have greatly reduced the number of dedicated testers necessary to ensure the absence of recall-class bugs. At Google, testing is aimed at determining how well this prevention method works.

Manifestations of this blending of development and testing are inseparable from the Google development mindset, from code review notes asking "where are your tests?" to posters in the bathrooms reminding developers about best-testing practices.[3] Testing must be an unavoidable aspect of development, and the marriage of development and testing is where quality is achieved.

Testing must be an unavoidable aspect of development, and the marriage of development and testing is where quality is achieved.

Roles

In order for the "you build it, you break it" motto to be real (and kept real over time), there are roles beyond the traditional feature developer that are necessary. Specifically, engineering roles that enable developers to do testing efficiently and effectively have to exist. At Google, we have created

3. http://googletesting.blogspot.com/2007/01/introducing-testing-on-toilet.html.

roles in which some engineers are responsible for making other engineers more productive and more quality-minded. These engineers often identify themselves as *testers*, but their actual mission is one of productivity. Testers are there to make developers more productive and a large part of that productivity is avoiding re-work because of sloppy development. Quality is thus a large part of that productivity. We are going to spend significant time talking about each of these roles in detail in subsequent chapters; therefore, a summary suffices for now.

The **software engineer (**SWE**)** is the traditional developer role. SWEs write functional code that ships to users. They create design documentation, choose data structures and overall architecture, and they spend the vast majority of their time writing and reviewing code. SWEs write a lot of test code, including test-driven design (TDD), unit tests, and, as we explain later in this chapter, participate in the construction of small, medium, and large tests. SWEs own quality for everything they touch whether they wrote it, fixed it, or modified it. That's right, if a SWE has to modify a function and that modification breaks an existing test or requires a new one, they must author that test. SWEs spend close to 100 percent of their time writing code.

The **software engineer in test (**SET**)** is also a developer role, except his focus is on testability and general test infrastructure. SETs review designs and look closely at code quality and risk. They refactor code to make it more testable and write unit testing frameworks and automation. They are a partner in the SWE codebase, but are more concerned with increasing quality and test coverage than adding new features or increasing performance. SETs also spend close to 100 percent of their time writing code, but they do so in service of quality rather than coding features a customer might use.

SETs are partners in the SWE codebase, but are more concerned with increasing quality and test coverage than adding new features or increasing performance. SETs write code that allows SWEs to test their features.

The **test engineer** (TE) is related to the SET role, but it has a different focus. It is a role that puts testing on behalf of the user first and developers second. Some Google TEs spend a good deal of their time writing code in the form of automation scripts and code that drives usage scenarios and even mimics the user. They also organize the testing work of SWEs and SETs, interpret test results, and drive test execution, particularly in the late stages of a project as the push toward release intensifies. TEs are product experts, quality advisers, and analyzers of risk. Many of them write a lot of code; many of them write only a little.

Note

The TE role puts testing on behalf of the user first. TEs organize the overall quality practices, interpret test results, drive test execution, and build end-to-end test automation.

From a quality standpoint, SWEs own features and the quality of those features in isolation. They are responsible for fault-tolerant designs, failure recovery, TDD, unit tests, and working with the SET to write tests that exercise the code for their features.

SETs are developers who provide testing features. A framework that can isolate newly developed code by simulating an actual working environment (a process involving such things as stubs, mocks, and fakes, which are all described later) and submit queues for managing code check-ins. In other words, SETs write code that enables SWEs to test their features. Much of the actual testing is performed by the SWEs. SETs are there to ensure that features are testable and that the SWEs are actively involved in writing test cases.

Clearly, an SET's primary focus is on the developer. Individual feature quality is the target and enabling developers to easily test the code they write is the primary focus of the SET. User-focused testing is the job of the Google TE. Assuming that the SWEs and SETs performed module- and feature-level testing adequately, the next task is to understand how well this collection of executable code and data works together to satisfy the needs of the user. TEs act as double-checks on the diligence of the developers. Any obvious bugs are an indication that early cycle developer testing was inadequate or sloppy. When such bugs are rare, TEs can turn to the primary task of ensuring that the software runs common user scenarios, meets performance expectations, is secure, internationalized, accessible, and so on. TEs perform a lot of testing and manage coordination among other TEs, contract testers, crowd sourced testers, dogfooders,[4] beta users, and early adopters. They communicate among all parties the risks inherent in the basic design, feature complexity, and failure avoidance methods. After TEs get engaged, there is no end to their mission.

Organizational Structure

In most organizations I have worked with, developers and testers exist as part of the same product team. Organizationally, developers and testers

4. The term *dogfood* is used by most software companies in the U.S. to denote internal adoption of software that is not yet released. The phrase "eating your own dogfood" is meant to convey the idea that if you make a product to sell to someone else, you should be willing to use it yourself to find out if it is any good.

report to the same product team manager. One product, one team, and everyone involved is always on the same page.

Unfortunately, I have never actually seen it work that way. Senior managers tend to come from program management or development and not testing ranks. In the push to ship, priorities often favor getting features complete and other fit-and-finish tasks over core quality. As a single team, the tendency is for testing to be subservient to development. Clearly, this is evident in the industry's history of buggy products and premature releases. Service Pack 1 anyone?

Note

As a single team, senior managers tend to come from program management or development and not testing ranks. In the push to ship, priorities often favor getting features complete and other fit-and-finish tasks over core quality. The tendency for such organizational structures is for testing to be subservient to development.

Google's reporting structure is divided into what we call Focus Areas or FAs. There is an FA for Client (Chrome, Google Toolbar, and so on), Geo (Maps, Google Earth, and so on), Ads, Apps, Mobile, and so on. All SWEs report to a director or VP of a FA.

SETs and TEs break this mold. Test exists in a separate and horizontal (across the product FAs) Focus Area called Engineering Productivity. Testers are essentially on loan to the product teams and are free to raise quality concerns and ask questions about functional areas that are missing tests or that exhibit unacceptable bug rates. Because we don't report to the product teams, we can't simply be told to get with the program. Our priorities are our own and they never waiver from reliability, security, and so on unless we decide something else takes precedence. If a development team wants us to take any shortcuts related to testing, these must be negotiated in advance and we can always decide to say no.

This structure also helps to keep the number of testers low. A product team cannot arbitrarily lower the technical bar for testing talent or hire more testers than they need simply to dump menial work on them. Menial work around any specific feature is the job of the developer who owns the feature and it cannot be pawned off on some hapless tester. Testers are assigned by Engineering Productivity leads who act strategically based on the priority, complexity, and needs of the product team in comparison to other product teams. Obviously, we can get it wrong, and we sometimes do, but in general, this creates a balance of resources against actual and not perceived need.

Note

Testers are assigned to product teams by Engineering Productivity leads who act strategically based on the priority, complexity, and needs of the product team in comparison to other product teams. This creates a balance of resources against actual and not perceived need. As a central resource, good ideas and practices tend to get adopted companywide.

The on-loan status of testers also facilitates movement of SETs and TEs from project to project, which not only keeps them fresh and engaged, but also ensures that good ideas move rapidly around the company. A test technique or tool that works for a tester on a Geo product is likely to be used again when that tester moves to Chrome. There's no faster way of moving innovations in test than moving the actual innovators.

It is generally accepted that 18 months on a product is enough for a tester and that after that time, he or she can (but doesn't have to) leave without repercussion to another team. One can imagine the downside of losing such expertise, but this is balanced by a company full of generalist testers with a wide variety of product and technology familiarity. Google is a company full of testers who understand client, web, browser, and mobile technologies, and who can program effectively in multiple languages and on a variety of platforms. And because Google's products and services are more tightly integrated than ever before, testers can move around the company and have relevant expertise no matter where they go.

Crawl, Walk, Run

One of the key ways Google achieves good results with fewer testers than many companies is that we rarely attempt to ship a large set of features at once. In fact, the exact opposite is the goal: Build the core of a product and release it the moment it is useful to as large a crowd as feasible, and then get their feedback and iterate. This is what we did with Gmail, a product that kept its beta tag for four years. That tag was our warning to users that it was still being perfected. We removed the beta tag only when we reached our goal of 99.99 percent uptime for a real user's email data. We did it again with Android producing the G1, a useful and well-reviewed product that then became much better and more fully featured with the Nexus line of phones that followed it. It's important to note here that when customers are paying for early versions, they have to be functional enough to make them worth their while. Just because it is an early version doesn't mean it has to be a poor product.

Note

Google often builds the "minimum useful product" as an initial version and then quickly iterates successive versions allowing for internal and user feedback and careful consideration of quality with every small step. Products proceed through canary, development, testing, beta, and release channels before making it to users.

It's not as cowboy a process as it might sound at first glance. In fact, in order to make it to what we call the beta channel release, a product must go through a number of other channels and prove its worth. For Chrome, a product I spent my first two years at Google working on, multiple channels were used depending on our confidence in the product's quality and the extent of feedback we were looking for. The sequence looks something like this:

- **Canary Channel**: This is used for daily builds we suspect aren't fit for release. Like a canary in a coalmine, if a daily build fails to survive, then it is a sign our process has gotten chaotic and we need to re-examine our work. Canary Channel builds are only for the ultra-tolerant user running experiments and certainly not for someone depending on the application to get real work done. In general, only engineers (developer and testers) and managers working on the product pull builds from the canary channel.

Note

The Android team goes one step further, and has its core development team's phone continually running on the nearly daily build. The thought is that they will be unlikely to check in bad code it if impacts the ability to call home.

- **Dev Channel:** This is what developers use for their day-to-day work. These are generally weekly builds that have sustained successful usage and passed some set of tests (we discuss this in subsequent chapters). All engineers on a product are *required* to pick up the Dev Channel build and use it for real work and for sustained testing. If a Dev Channel build isn't suitable for real work, then back to the Canary channel it goes. This is not a happy situation and causes a great deal of re-evaluation by the engineering team.
- **Test Channel:** This is essentially the best build of the month in terms of the one that passes the most sustained testing and the one engineers trust the most for their work. The Test Channel build can be picked up by internal dogfood users and represents a candidate Beta Channel build given good sustained performance. At some point, a Test Channel build becomes stable enough to be used internally companywide and

sometimes given to external collaborators and partners who would benefit from an early look at the product.

- **Beta Channel** or **Release Channel:** These builds are stable Test Channel builds that have survived internal usage and pass every quality bar the team sets. These are the first builds to get external exposure.

This crawl, walk, run approach gives us the chance to run tests and experiment on our applications early and obtain feedback from real human beings in addition to all the automation we run in each of these channels every day.

Types of Tests

Instead of distinguishing between code, integration, and system testing, Google uses the language of *small*, *medium,* and *large* tests (not to be confused with t-shirt sizing language of estimation among the agile community), emphasizing scope over form. Small tests cover small amounts of code and so on. Each of the three engineering roles can execute any of these types of tests and they can be performed as automated or manual tests. Practically speaking, the smaller the test, the more likely it is to be automated.

Instead of distinguishing between code, integration, and system testing, Google uses the language of *small*, *medium,* and *large* tests, emphasizing scope over form.

Small tests are mostly (but not always) automated and exercise the code within a single function or module. The focus is on typical functional issues, data corruption, error conditions, and off-by-one mistakes. Small tests are of short duration, usually running in seconds or less. They are most likely written by a SWE, less often by an SET, and hardly ever by TEs. Small tests generally require mocks and faked environments to run. (Mocks and fakes are stubs—substitutes for actual functions—that act as placeholders for dependencies that might not exist, are too buggy to be reliable, or too difficult to emulate error conditions. They are explained in greater detail in later chapters.) TEs rarely write small tests but might run them when they are trying to diagnose a particular failure. The question a small test attempts to answer is, "*Does this code do what it is supposed to do?*"

Medium tests are usually automated and involve two or more interacting features. The focus is on testing the interaction between features that call each other or interact directly; we call these *nearest neighbor functions*. SETs drive the development of these tests early in the product cycle as individual features are completed and SWEs are heavily involved in writing, debugging, and maintaining the actual tests. If a medium test fails or

breaks, the developer takes care of it autonomously. Later in the development cycle, TEs can execute medium tests either manually (in the event the test is difficult or prohibitively expensive to automate) or with automation. The question a medium test attempts to answer is, *"Does a set of near neighbor functions interoperate with each other the way they are supposed to?"*

Large tests cover three or more (usually more) features and represent real user scenarios, use real user data sources, and can take hours or even longer to run. There is some concern with overall integration of the features, but large tests tend to be more results-driven, checking that the software satisfies user needs. All three roles are involved in writing large tests and everything from automation to exploratory testing can be the vehicle to accomplish them. The question a large test attempts to answer is, *"Does the product operate the way a user would expect and produce the desired results?"* End-to-end scenarios that operate on the complete product or service are large tests.

Note

Small tests cover a single unit of code in a completely faked environment. **Medium tests** cover multiple and interacting units of code in a faked or real environment. **Large tests** cover any number of units of code in the actual production environment with real and not faked resources.

The actual language of small, medium, and large isn't important. Call them whatever you want as long as the terms have meaning that everyone agrees with.[5] The important thing is that Google testers share a common language to talk about what is getting tested and how those tests are scoped. When some enterprising testers begin talking about a fourth class they dubbed *enormous tests*, every other tester in the company could imagine a systemwide test covering every feature and that runs for a very long time. No additional explanation is necessary.[6]

The primary driver of what gets tested and how much is a very dynamic process and varies from product to product. Google prefers to release often and leans toward getting a product out to users quickly so we can get feedback and iterate. Google tries hard to develop only products that users will find compelling and to get new features out to users as early

---

5. The original purpose of using small, medium, and large was to standardize terms that so many testers brought in from other employers where smoke tests, BVTs, integrations tests, and so on had multiple and conflicting meanings. Those terms had so much baggage that it was felt that new ones were needed.

6. Indeed, the concept of an enormous test is formalized and Google's automation infrastructure uses these designations of small, medium, and so on to determine the execution sequence during automated runs. This is described in more detail in the chapter on SETs later in this book.

as possible so they might benefit from them. Plus, we avoid over-investing in features no user wants because we learn this early. This requires that we involve users and external developers early in the process so we have a good handle on whether what we are delivering hits the mark.

Finally, the mix between automated and manual testing definitely favors the former for all three sizes of tests. If it can be automated and the problem doesn't require human cleverness and intuition, then it should be automated. Only those problems, in any of the previous categories, that specifically require human judgment, such as the beauty of a user interface or whether exposing some piece of data constitutes a privacy concern, should remain in the realm of manual testing.

The mix between automated and manual testing definitely favors the former for all three sizes of tests. If it can be automated and the problem doesn't require human cleverness and intuition, then it should be automated.

Having said that, it is important to note that Google performs a great deal of manual testing, both scripted and exploratory, but even this testing is done under the watchful eye of automation. Recording technology converts manual tests to automated tests, with point-and-click validation of content and positioning, to be re-executed build after build to ensure minimal regressions, and to keep manual testers always focusing on new issues. We also automate the submission of bug reports and the routing of manual testing tasks.[7] For example, if an automated test breaks, the system determines the last code change that is the most likely culprit, sends email to its authors, and files a bug automatically. The ongoing effort to automate to within the "last inch of the human mind" is currently the design spec for the next generation of test-engineering tools Google builds.

7. Google's recording technology and automation-assisted manual testing are described in detail in subsequent chapters on the TE role.

Index

Symbols

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W-X-Y-Z