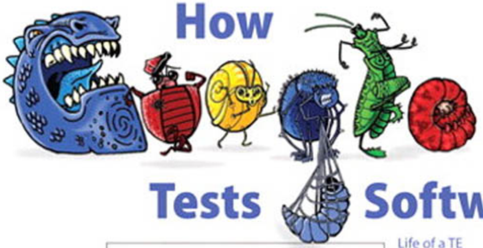
How



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

Better candidates will do even more:

- **Think about scale:** Maybe the return type should be a 64-bit integer because Google often deals with large amounts of data.
- **Think about re-use:** Why does this function count only As? It is probably a good idea to parameterize this so that an arbitrary character can be counted instead of having a different function defined for each one.
- **Think about safety:** Are these pointers coming from trusted sources?

The best candidates will:

- **Think about scale:**
    - Is this function to be run as part of a MapReduce[17] on sharded[18] data? Maybe that's the most useful form of calling this function. Are there issues to worry about in this scenario? Consider the performance and correctness implications of running this function on every document on the entire Internet.
    - If this subroutine is called for every Google query and would be called only with safe pointers because the wrapper does this validation already; maybe avoiding a null check will save hundreds of millions of CPU cycles a day and reduce user-visible latency by some small amount. At least understand the possible implications of full textbook parameter validation.
- **Think about optimizations based on invariants:**
    - Can we assume the data coming in is already sorted? If so, we might be able to exit quickly after we find the first B.
    - What is the texture of the input data? Is it most often all As, is it most often a mix of all characters, or is it only As and spaces? If so, there may be optimizations in our comparison operations. When dealing with large data, even small, sublinear changes can be significant for actual compute latencies when the code executes.
- **Think about safety:**
    - On many systems, and if this is a security-sensitive section of code, consider testing for more than just nonnull pointers; 1 is an invalid pointer value on some systems.
    - Add a length parameter to help ensure the code doesn't walk off the end of the string. Check the length parameter's value for sanity. Null-terminated character strings are a hacker's best friend.

[17] MapReduce is a form of distributed computing where the computation is broken into smaller pieces, categorized by key (mapped), and then rolled up (reduced) by key. See http://en.wikipedia.org/wiki/MapReduce.

[18] Sharding is a form of database partitioning. Horizontal partitioning is a database design principle whereby rows of a database table are held separately, rather than splitting by columns. See http://en.wikipedia.org/wiki/Shard_(database_architecture).

- If there is a possibility the buffer can be modified by some other thread while this function executes, there may be thread safety issues.
- Should we be doing this check in a try/catch? Or, if the calling code isn't expecting exceptions, we should probably return error codes to the caller. If there are error codes, are those codes well defined and documented? This shows thinking about the context of the larger code-base and runtime, and this kind of thinking can avoid errors of confusion or omission down the road.

Ultimately, the best candidates come up with a new angle for these questions. All angles are interesting to consider, if they are considered intelligently.

Note

A good SET candidate should not have to be told to test the code she writes. It should be an automatic part of her thinking.

The key in all this questioning of the spec and the inputs is that any engineer who has passed an introductory programming course can produce basic functional code for this question. These questions and thinking from the candidates differentiate the best candidates from the decent ones. We do make sure that the candidate feels comfortable enough socially and culturally to ask questions, and if not, we prod them a bit to ask, making sure their straight-to-the-code behavior isn't just because they are in an interview situation. Googlers should question most everything without being annoying and by still getting the problem solved.

It would be boring to walk through the myriad of possible correct implementations and all the common mistakes as this isn't a programming or interviewing book. But, let's show a simple and obvious implementation for discussion's sake. Note: Candidates can usually use the language they are most comfortable with such as Java or Python, though that usually elicits some questions to ensure they understand things such as garbage collection, type–safety, compilation, and runtime concerns.

```
int64 Acount(const char* s) {
  if (!s) return 0;
  int64 count = 0;
  while (*s++) {
    if (*s == ‘a’) count++;
  }
  return count;
}
```

Candidates should be able to walk through their code, showing the evolution of pointer and counter values as the code executes with test inputs.

In general, decent SET candidates will do the following:

- Have little trouble with the basics of coding this solution. When doing so, they do not have trouble rewriting or fumbling over basic syntax issues or mixing up syntax or keywords from different languages.
- Show no sign of misunderstanding pointers or allocating anything unnecessarily.
- Perform some input validation upfront to avoid pesky crashes from dereferencing null pointers and such, or have a good explanation of why they are not doing such parameter validation when asked.
- Understand the runtime or Big O[19] of their code. Anything other than linear here shows some creativity, but can be concerning.
- If there are minor issues in the code, they can correct them when pointed out.
- Produce code that is clear and easily readable by others. If they are using bitwise operators or put everything on one line, this is not a good sign, even if the code functionally works.
- Walk through their code with a single test input of A or null.

Better candidates will do even more.

- Consider int64 for counters and return type for future compatibility and avoiding overflow when someone inevitably uses this function to count As in an insanely long string.
- Write code to shard/distribute the counting computation. Some candidates unfamiliar with MapReduce can come up with some simple variants on their own to decrease latency with parallel computation for large strings.
- Write in assumptions and invariants in notes or comments in the code.
- Walk through their code with many different inputs and fix every bug that they find. SET candidates who don't spot and fix bugs are a warning sign.
- Test their function before being asked. Testing should be something they don't have to be told to do.
- Continue trying to optimize the solution until asked to stop. No one can be sure that their code is perfect after just a few minutes of coding and applying a few test inputs. Some tenacity over correctness should be evident.

Now, we want to see if the candidate can test his own code. Convoluted or tricky test code is likely the worst test code in the world and definitely worse than no code. When tests fail at Google, it needs to be clear what the

[19] Big O notation describes how long it takes some function to execute based on the size of the input data. See http://en.wikipedia.org/wiki/Big_O_notation.

test is doing. If not, engineers might disable, mark as flaky, or ignore the failures—it happens, and it's the fault of the SETs and SWEs who wrote and reviewed the code for letting this bad test code into the tree.

SETs should be able to approach testing in a black box manner, operating under the assumption that someone else implemented the function, or in a white box manner, knowing which test cases might be irrelevant given the specifics of their implementation.

In general, decent candidates will do the following:

- Are methodical and systematic. Supplying test data according to some identifiable characteristic (such as string size) and not just random strings.
- Focus on generating interesting test data. Consider how to run large tests and where to get real-world test data.

- Better candidates:
    - Want to spin up concurrent threads executing this function to look for cross talk, deadlock, and memory leaks.
    - Build long running tests, such as spin up tests in a `while(true)` loop, and ensure they continue to work over the long haul.
    - Remain interested in coming up with test cases and interesting approaches to test data generation, validation, and execution.

Example of a Great Candidate *by Jason Arbon*

One recent candidate (who has since proven to be amazing on the job) was asked how he would do boundary testing for a version of this API with 64-bit integers. He realized quickly that it wouldn't be physically possible because of time and space constraints, but for the sake of completeness and curiosity when thinking about this level of scale, thought about how to host at least very large amounts of data for such a test and considered using Google's index of the Web as input data.

How did he validate the answer? He suggested using a parallel implementation and making sure the two produced the same result. He also thought of a statistical sampling approach: What is the expected frequency of A in web pages, and because we know the number of pages indexed, the number computed should be close. This is a Google-like way to think about testing. Even if we don't build these monster tests, thinking about these large solutions usually leads to more interesting or efficient solutions for normal-scale work.

Another thing we interview for is "Googliness," or culture fit. Is the SET candidate technically curious during the interview? When presented with some new ideas, can the candidate incorporate these into her solution? How does she handle ambiguity? Is she familiar with academic approaches to quality such as theorem proving? Does she understand measures of quality or automation in other fields such as civil or aerospace engineering?

Is she defensive about bugs you might find in her implementation? Does she think big? Candidates don't have to be all of these things, but the more, the merrier! And, finally, would we like to work with this person on a daily basis?

It is important to note that if someone interviewing for an SET position isn't that strong of a coder, it does not mean that she will not be a successful TE. Some of the best TEs we've ever hired originally interviewed for the SET position.

An interesting note about SET hiring at Google is that we often lose great candidates because they run into a nontesting SWE or an overly focused TE on their interview loop. We want this diversity in the folks interviewing SET candidates, because they will work together on the job, and SETs are really hybrids, but this can sometimes result in unfavorable interview scores. We want to make sure the bad scores come from interviewers who truly appreciate all aspects of what it takes to be a great SET.

As Pat Copeland says in his forward, there has been and still is a lot of diversity of opinion on SET hiring. Should an SET just be doing feature work if he is good at coding? SWEs are also hard to hire. Should they just be focusing on pure testing problems if they are that good at testing? The truth, as it often is, lies somewhere in the middle.

Getting good SET hires is a lot of trouble but it's worth it. A single rock star SET can make a huge impact on a team.

An Interview with Tool Developer Ted Mao

Ted Mao is a developer at Google but he's a developer who has been exclusively focused on building test tools. Specifically, he's building test tools for web applications that scale to handle everything Google builds internally. As such, he's a well-known person in SET circles because an SET without good tools will find it hard to be effective. Ted is probably more familiar with the common web test infrastructure at Google than anyone in the company.

HGTS: When did you start at Google, and what excited you about working here?

Ted: I joined Google in June 2004. Back then, I only had experience working at large companies like IBM and Microsoft, and Google was the hot startup to work for and they were attracting a lot of talented engineers. Google was attempting to solve many interesting, challenging problems, and I wanted to work on these problems alongside some of the best engineers in the world.

HGTS: You are the inventor of Buganizer,[20] Google's bug database. What were the core things you were trying to accomplish with Buganizer versus the older BugDB?

[20] The open-source version of Buganizer is called Issue Tracker and is available through the Chromium project at http://code.google.com/chromium/issues/list.

Ted: BugsDB was impeding our development process rather than supporting it. To be honest, it was wasting a lot of valuable engineering time and this was a tax paid by every team that used it. The issues manifested themselves in many ways, including UI latency, awkward workflows, and the practice of using "special" strings in unstructured text fields. In the process of designing Buganizer, we made sure that our data model and UI reflected our users' actual development processes and that the system would be amenable to future extension both in the core product and through integrations.

HGTS: Well you nailed Buganizer. It's truly the best bug database any of us have ever used. How did you start working on web-testing automation, did you see the need or were you asked to solve a problem with test execution?

Ted: While working on Buganizer, AdWords, and other products at Google, I consistently found that the web-testing infrastructure we had available was insufficient for my needs. It was never quite as fast, scalable, robust, or useful as I needed it to be. When the tools team announced that they were looking for someone to lead an effort in this area, I jumped on the opportunity to solve this problem. This effort became known as the Matrix project and I was the tech lead for it.

HGTS: How many test executions and teams does Matrix support today?

Ted: It really depends on how you measure test executions and teams. For example, one metric we use is what we call a "browser session"—every new browser session for a particular browser is guaranteed to start in the same state, and thus, a test running in the browser will behave deterministically insomuch as the test, browser, and operating system are deterministic. Matrix is used by practically every web frontend team at Google and provisions more than a million new browser sessions per day.

HGTS: How many people worked on these two projects: Buganizer and Matrix?

Ted: During their peak development periods, Buganizer had about five engineers and Matrix had four engineers. It's always somewhat sad for me to think of what we might have been able to accomplish with a larger, more sustained development team, but I think we did a great job given what we had to work with.

HGTS: What were the toughest technical challenges you faced while building these tools?

Ted: I think that the toughest and often the most interesting challenges for me have always come at design time—understanding a problem space, weighing different solutions and their tradeoffs, and then making good decisions. Implementation is usually straightforward from that point. These types of decisions have to be made throughout the life of a project, and together with implementation, they can make or break the product.

HGTS: What general advice would you give to other software engineers in the world who are working on testing tools?

Ted: Focus on your users, understand their needs, and solve their problems. Don't forget about "invisible" features like usability and speed. Engineers are