# CHAPTER 3 The Test Engineer

Testability and long-term viability of test automation infrastructure is the purview of the Software Engineer in Test (SET). The Test Engineer (TE) plays a related but different role where the focus is on user impact and risk to the overall mission of the software product. Like most technical roles at Google, there is some coding involved, but the TE role is by far the most broad of all the engineers. He contributes to the product, but many of the tasks a TE takes on require no coding.[1]

## A User-Facing Test Role

In the prior chapter, we introduced the TE as a "user-developer" and this is not a concept to take lightly. The idea that all engineers on a product team fit the mold of a developer is an important part of keeping all stakeholders on an equal footing. At companies like Google where honoring "the writing of code" is such an important part of our culture, TEs need to be involved as engineers to remain first-class citizens. The Google TE is a mix of technical skills that developers respect and a user facing focus that keeps developers in check. Talk about a split personality!

TEs need to be involved as engineers to remain first-class citizens. The Google TE is a mix of technical skills that developers respect and a user facing focus that keeps developers in check.

[1] This is a general view. Many TEs perform work more akin to the SET role and write a lot of code. Some TEs perform a role more closely aligned with a release engineer and write little code.

The TE job description is the hardest to nail down as one size definitely does not fit all. TEs are meant as an overseer of all things quality as the various build targets come together and ultimately comprise the entire product. As such, most TEs get involved in some of this lower-level work where another set of eyes and more engineering expertise is needed. It's a matter of risk: TEs find and contribute to the riskiest areas of the software in whatever manner makes the most sense for that particular product. If SET work is the most valuable, then that's what a TE does; if code reviews are the most valuable, then so be it. If the test infrastructure is lacking, then it gets some TE attention.

These same TEs can then lead exploratory testing sessions at some other point in a project or manage a dogfood or beta testing effort. Sometimes this is time-driven as early phase work means far more SET-oriented tasks are needed, and later in the cycle, TE-oriented work is prevalent. Other cases are the personal choice of the TEs involved and there are a number of cases where engineers convert from one of these roles to another. There are no absolutes. What we describe in the following section is essentially the ideal case.

## The Life of a TE

The TE is a newer role at Google than either Software Engineers (SWEs) or SETs. As such, it is a role still in the process of being defined. The current generation of Google TEs is blazing a trail that will guide the next generation of new hires for this role. Here we present the latest emerging TE processes at Google.

Not all products require the attention of a TE. Experimental efforts and early-stage products without a well defined mission or user story are certainly projects that won't get a lot of (or any) TE attention. If the product stands a good chance of being cancelled (in the sense that as a proof of concept, it fails to pass muster) or has yet to engage users or have a well defined set of features, testing is largely something that should be done by the people writing the code.

Even if it is clear that a product is going to get shipped, TEs often have little testing to do early in the development cycle when features are still in flux and the final feature list and scope are undetermined. Overinvesting in test engineering too early, especially if SETs are already deeply engaged, can mean wasted effort. Testing collateral that gets developed too early risks being cast aside or, worse, maintained without adding value. Likewise, early test planning requires fewer TEs than later-cycle exploratory testing when the product is close to final form and the hunt for missed bugs has a greater urgency.

TEs often have little to do early in the development cycle when features are still in flux and the final feature list and scope are undetermined.

The trick in staffing a project with TEs has to do with risk and return on investment. Risk to the customer and to the enterprise means more testing effort and requires more TEs, but that effort needs to be in proportion with the potential return. We need the right number of TEs and we need them to engage at the right time and with the right impact.

After they are engaged, TEs do not have to start from scratch. There is a great deal of test engineering and quality-oriented work performed by SWEs and SETs, which becomes the starting point for additional TE work. The initial engagement of the TE is to decide things such as:

- Where are the weak points in the software?
- What are the security, privacy, performance, reliability, usability, compatibility, globalization, or other concerns?
- Do all the primary user scenarios work as expected? Do they work for all international audiences?
- Does the product interoperate with other products (hardware and software)?
- In the event of a problem, how good are the diagnostics?

Of course these are only a subset. All of this combines to speak to the risk profile of releasing the software in question. TEs don't necessarily do all of this work, but they ensure that it gets done and they leverage the work of others in assessing where additional work is required. Ultimately, TEs are paid to protect users and the business from bad design, confusing UX, functional bugs, security and privacy issues, and so on. At Google, TEs are the only people on a team whose full-time job is to look at the product or service holistically for weak points. As such, the life of a TE is much less prescriptive and formalized than that of an SET. TEs are asked to help on projects in all stages of readiness: everything from the idea stage to version 8 or even watching over a deprecated or "mothballed" project. Often, a single TE even spans multiple projects, particularly those TEs with specialty skill sets such as security, privacy, or internationalization.

Clearly, the work of a TE varies greatly depending on the project. Some TEs spend much of their time programming, but with more of a focus on medium and large tests (such as end-to-end user scenarios) rather than small tests. Other TEs take existing code and designs to determine failure modes and look for errors that can cause those failures. In such a role, a TE might modify code but not create it from scratch. TEs must be more systematic and thorough in their test planning and completeness with a focus on the actual usage and systemwide experience. TEs excel at dealing with ambiguity in requirements and at reasoning and communicating about fuzzy problems.

Successful TEs accomplish all this while navigating the sensitivities and, sometimes, strong personalities of the development and product team members. When weak points are found, TEs happily break the software and drive to get these issues resolved with the SWEs, PMs, and SETs. TEs are generally some of the most well known people on a team because of the breadth of interactions their jobs require.

Such a job description can seem like a frightening prospect given the mix of technical skill, leadership, and deep product understanding required. Indeed, without proper guidance, it is a role in which many would expect to fail. However, at Google, a strong community of TEs has emerged to counter this. Of all job functions, the TE role is perhaps the best peer-supported role in the company. The insight and leadership required to perform the TE job successfully means that many of the top test managers in the company come from the TE ranks.

The insight and leadership required to perform the TE job successfully means that many of the top test managers in the company come from the TE ranks.

There is fluidity to the work of a Google TE that belies any prescriptive process for engagement. TEs can enter a project at any point and must assess the state of the project, code, design, and users quickly and decide what to focus on first. If the project is just getting started, test planning is often the first order of business. Sometimes TEs are pulled in late in the cycle to evaluate whether a project is ready for ship or if there are any major issues before an early "beta" goes out. If they are brought into a newly acquired application or one in which they have little prior experience, they often start doing some exploratory testing with little or no planning. Sometimes projects haven't been released for quite a while and just need some touchups, security fixes, or UI updates—calling for an even different approach.

One size rarely fits all for TEs at Google. We often describe a TE's work as "starting in the middle" in that a TE has to be flexible and integrate quickly into a product team's culture and current state. If it's too late for a test plan, don't build one. If a project needs tests more than anything else, build just enough of a plan to guide that activity. Starting at "the beginning" according to some testing dogma is simply not practical.

Following is the general set of practices we prescribe for TEs:

- Test planning and risk analysis
- Review specs, designs, code, and existing tests
- Exploratory testing
- User scenarios

- Test case creation
- Executing test cases
- Crowd sourcing
- Usage metrics
- User feedback

Of course, TEs with a strong personality and excellent communication skills are the ones who do all these things to maximum impact.

## Test Planning

Developers have a key advantage over testers in that the artifact they work on is one that everyone cares about. Developers deal with code and because that code becomes the application that users covet and that makes profit for the company, it is by definition the most important document created during project execution.

Testers, on the other hand, deal with documents and artifacts of a far more temporal nature. In the early phases of a project, testers write test plans; later, they create and execute test cases and create bug reports. Still, later they write coverage reports and collect data about user satisfaction and software quality. After the software is released and is successful (or not), few people ask about testing artifacts. If the software is well loved, people take the testing for granted. If the software is poor, people might question the testing, but it is doubtful that anyone would want to actually see it.

Testers cannot afford to be too egotistical about test documentation. In the throes of the coding, reviewing, building, testing, rinsing, and repeating cycle that is software development, there is little time to sit around and admire a test plan. Poor test cases rarely achieve enough attention to be improved; they simply get thrown out in favor of those that are better. The attention is focused on the growing codebase and as the only artifact that actually matters, this is as it should be.

As test documentation goes, test plans have the briefest actual lifespan of any test artifact.[2] Early in a project, there is a push to write a test plan (see Appendix A, "Chrome OS Test Plan," for an early Chrome OS test plan). Indeed, there is often an insistence among project managers that a test plan must exist and that writing it is a milestone of some importance. But, once such a plan is written, it is often hard to get any of those same managers to take reviewing and updating it seriously. The test plan becomes a beloved stuffed animal in the hands of a distracted child. We want it to be there at all times. We drag it around from place to place without ever giving it any real attention. We only scream when it gets taken away.

[2] Clearly in the event a customer negotiates the development of a test plan or some government regulation requires it, the flexibility we talk about here disappears. There are some test plans that have to be written and kept up to date!

Test plans are the first testing artifact created and the first one to die of neglect. At some early point in a project, the test plan represents the actual software as it is intended to be written but unless that test plan is tended constantly, it soon becomes out of date as new code is added, features veer from their preplanned vision, and designs that looked good on paper are re-evaluated as they are implemented and meet feedback from users. Maintaining a test plan through all these planned and unplanned changes is a lot of work and only worthwhile if the test plan is regularly consulted by a large percentage of the projects’ stakeholders.

Test plans are the first testing artifact created and the first one to die of neglect.

This latter point is the real killer of the idea of test planning: How much does a test plan actually drive testing activity throughout the entire lifecycle of a product? Do testers continually consult it as they divvy up the features among them so as to divide and conquer the app? Do developers insist the test plan be updated as they add and modify functionality? Do development managers keep a copy open on their desktop as they manage their to-do list? How often does a test manager refer to the contents of a test plan in status and progress meetings? If the test plan is actually important, all of these things would be happening every day.

Ideally, the test plan should play such a central role during project execution. Ideally, it should be a document that lives as the software lives, getting updates as the codebase gets updates and representing the product as it *currently* exists, not as it existed at the start of a project. Ideally, it should be useful for getting new engineers up to speed as they join a project already in progress.

That’s the ideal situation. It’s also a situation few testers have actually achieved here at Google or anywhere else for that matter.

Here are some features we want in a test plan:

- It is always up to date.
- It describes the intent of the software and why it will be loved by its users.
- It contains a snapshot of how the software is structured and the names of the various components and features.
- It describes what the software should do and summaries how it will do it.

From a purely testing point of view, we have to worry about the test plan being relevant while not making its care and feeding such a burden that it becomes more work than it is worth:

- It cannot take long to create and must be readily modifiable.
- It should describe what must be tested.
- It should be useful during testing to help determine progress and coverage gaps.

At Google, the history of test planning is much the same as other companies we've experienced. Test planning was a process determined by the people doing it and executed according to local (meaning the individual team) custom. Some teams wrote test plans in Google Docs (text documents and spreadsheets), shared with their engineering team but not in a central repository. Other teams linked their test plans on their product's home page. Still others added them to the internal Google Sites pages for their projects or linked to them from the engineering design documents and internal wikis. A few teams even used Microsoft Word documents, sent around in emails to the team in a proper old-school way. Some teams had no test plans at all, just test cases whose sum total, we must suppose, represented the plan.

The review path for these plans was opaque and it was hard to determine the authors and reviewers. Far too many of the test plans had a time and date stamp that made it all too clear that they had been written and long forgotten like the sell-by date on that old jar of jam in the back of the refrigerator. It must have been important to someone at some time, but that time has passed.

There was a proposal floated around Google to create a central repository and template for *all* product test plans. This was an interesting idea that has been tried elsewhere, but one clearly contrary to Google's inherently distributed and self-governed nature where "states rights" was the norm and big government a concept that brought derision.

Enter ACC (Attribute Component Capability) analysis, a process pulled together from the best practices of a number of Google test teams and pioneered by the authors and several colleagues in various product areas. ACC has passed its early adopter's phase and is being exported to other companies and enjoying the attention to tool developers who automate it under the "Google Test Analytics" label.

ACC has the following guiding principles:

- **Avoid prose and favor bulleted lists**. Not all testers wish to be novelists or possess the skill set to adequately capture in prose a product's purpose in life or its testing needs. Prose can be hard to read and is easily misinterpreted. Just the facts please!
- **Don't bother selling**. A test plan is not a marketing document or a place to talk about how important a market niche the product satisfies or how cool it is. Test plans aren't for customers or analysts; they are for engineers.

- **No fluff**. There is no length expectation for a test plan. Test plans are not high school term projects where length matters. Bigger is not better. The size of the plan is related to the size of the testing problem, not the propensity of the author to write.
- **If it isn't important and actionable, don't put it in the plan**. Not a single word in the plan should garner a "don't care" reaction from a potential stakeholder.
- **Make it flow**. Each section of the test plan should be an expansion of earlier sections so that one can stop reading at anytime and have a picture of the product's functionality in his head. If the reader wants more detail, he can continue reading.
- **Guide a planner's thinking**. A good planning process helps a planner think through functionality and test needs and logically leads from higher-level concepts into lower-level details that can be directly implemented.
- **The outcome should be test cases**. By the time the plan is completed, it should clearly describe not just what testing needs to be done but that it should also make the writing of test cases obvious. A plan that doesn't lead directly to tests is a waste of time.

A plan that doesn't lead directly to tests is a waste of time.

This last point is crucial: If the test plan does not describe in enough detail what test cases need to be written, then it hasn't served its primary purpose of helping us test the application we are building. The *planning of tests* should put us in a position to know what tests need to be written. You are finished planning when you are at exactly that spot: You know what tests you need to write.

ACC accomplishes this by guiding the planner through three views of a product corresponding to 1) *adjectives* and *adverbs* that describe the product's purpose and goals, 2) *nouns* that identify the various parts and features of the product, and 3) *verbs* that indicate what the product actually does. It follows that testing allows us to test that those capabilities work and the components as written satisfy the application's purpose and goals.

## A is for "Attribute"

When starting test planning or ACC, it is important to first identify why the product is important to users and to the business. Why are we building this thing? What core value does it deliver? Why is it interesting to customers? Remember, we're not looking to either justify or explain these things, only to label them. Presumably the PMs and product planners, or developers, have done their job of coming up with a product that matters in the marketplace. From a testing perspective, we just need to capture and label these things so we can ensure they are accounted for when we test it.

We document the core values in a three-step process called Attribute, Component, Capability analysis and we do so in that order, with attributes as the first target.

Attributes are the adjectives of the system. They are the qualities and characteristics that promote the product and distinguish it from the competition. In a way, they are the reasons people would choose to use the product over a competitor. Chrome, for example, is held to be fast, secure, stable, and elegant, and these are the attributes we try to document in the ACC. Looking ahead, we want to get to a point where we can attach test cases to these labels so that we know how much testing we have done to show that Chrome is fast, secure, and so on.

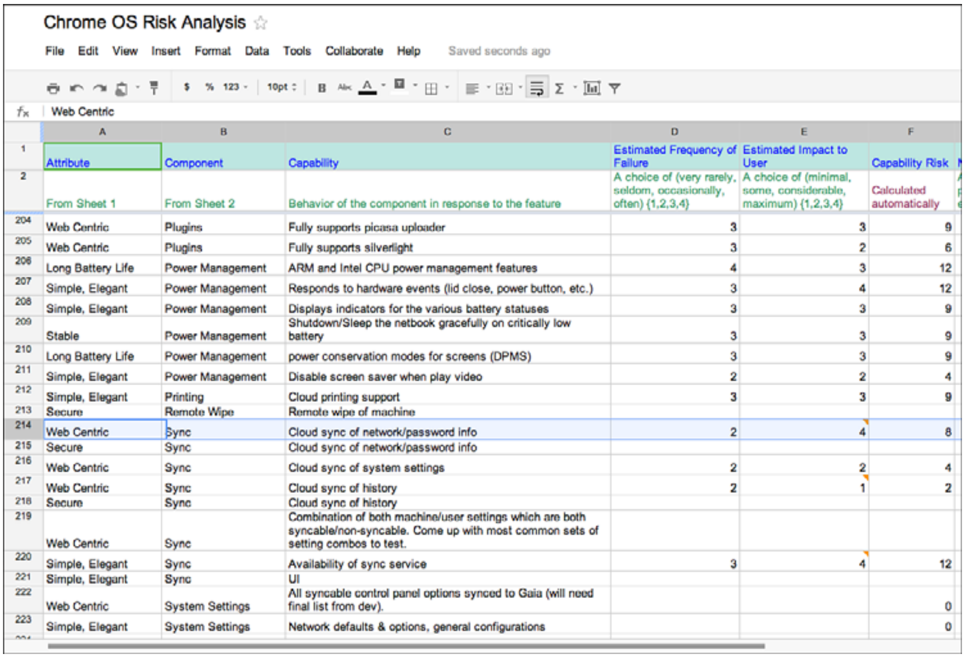
Attributes are the adjectives of the system. They are the qualities and characteristics that promote the product and distinguish it from the competition. Attributes are the reasons people would choose to use the product over a competitor.

Typically, a product manager has a hand in narrowing down the list of attributes for the system. Testers often get this list by reading the product requirements document, the vision and mission statement of the team, or even by simply listening to a sales guy describe the system to a prospective customer. Indeed, we find at Google that salespeople and product evangelists are an excellent source of attributes. Just imagine back-of-the-box advertising or think about how the product would be pitched on QVC, and you can get the right mindset to list the attributes.

Some tips on coming up with attributes for your own projects:

- **Keep it simple**. If it takes more than an hour or two, you are spending too long on this step.
- **Keep it accurate**. Make sure it comes from a document or marketing information that your team already accepts as truth.
- **Keep it moving.** Don’t worry if you missed something—if it’s not obvious later, it probably wasn’t that important anyway.
- **Keep it short**. No more than a dozen is a good target. We boiled an operating system down to 12 key attributes (see Figure 3.1), and in retrospect, we should have shortened that list to 8 or 9.

**FIGURE 3.1** Original Chrome risk analysis.



| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | Attribute | Component | Capability | Estimated Frequency of Failure | Estimated Impact to User | Capability Risk |
| 2 | From Sheet 1 | From Sheet 2 | Behavior of the component in response to the feature | A choice of (very rarely, seldom, occasionally, often) {1,2,3,4} | A choice of (minimal, some, considerable, maximum) {1,2,3,4} | Calculated automatically |
| 204 | Web Centric | Plugins | Fully supports picasa uploader | 3 | 3 | 9 |
| 205 | Web Centric | Plugins | Fully supports silverlight | 3 | 2 | 6 |
| 206 | Long Battery Life | Power Management | ARM and Intel CPU power management features | 4 | 3 | 12 |
| 207 | Simple, Elegant | Power Management | Responds to hardware events (lid close, power button, etc.) | 3 | 4 | 12 |
| 208 | Simple, Elegant | Power Management | Displays indicators for the various battery statuses | 3 | 3 | 9 |
| 209 | Stable | Power Management | Shutdown/Sleep the netbook gracefully on critically low battery | 3 | 3 | 9 |
| 210 | Long Battery Life | Power Management | power conservation modes for screens (DPMS) | 3 | 3 | 9 |
| 211 | Simple, Elegant | Power Management | Disable screen saver when play video | 2 | 2 | 4 |
| 212 | Simple, Elegant | Printing | Cloud printing support | 3 | 3 | 9 |
| 213 | Secure | Remote Wipe | Remote wipe of machine | | | |
| 214 | Web Centric | Sync | Cloud sync of network/password info | 2 | 4 | 8 |
| 215 | Secure | Sync | Cloud sync of network/password info | | | |
| 216 | Web Centric | Sync | Cloud sync of system settings | 2 | 2 | 4 |
| 217 | Web Centric | Sync | Cloud sync of history | 2 | 1 | 2 |
| 218 | Secure | Sync | Cloud sync of history | | | |
| 219 | Web Centric | Sync | Combination of both machine/user settings which are both syncable/non-syncable. Come up with most common sets of setting combos to test. | | | |
| 220 | Simple, Elegant | Sync | Availability of sync service | 3 | 4 | 12 |
| 221 | Simple, Elegant | Sync | UI | | | |
| 222 | Web Centric | System Settings | All syncable control panel options synced to Gaia (will need final list from dev). | | | 0 |
| 223 | Simple, Elegant | System Settings | Network defaults & options, general configurations | | | 0 |

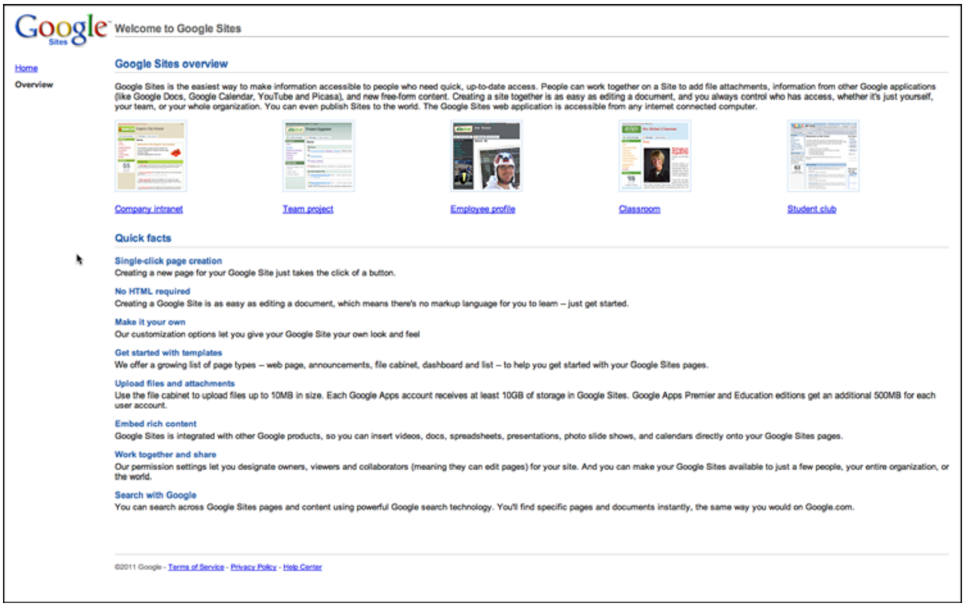
**Note**

Some figures in this chapter are representational and are not intended to be read in detail.

Attributes are used to figure out what the product does to support the core reasons for the product's existence and to surface these reasons to testers so they can be aware of how the testing they do impacts the application's ultimate reason for existence.

As an example, consider the attributes for a product called Google Sites, which is a freely available application for building a shared website for some open or closed community. Sites, as you'll find with many end-user applications, is kind enough to provide most of its attributes for you in its own documentation, as shown in Figure 3.2.

**FIGURE 3.2** Welcome to Google Sites.



Welcome to Google Sites

Home

Overview

**Google Sites overview**

Google Sites is the easiest way to make information accessible to people who need quick, up-to-date access. People can work together on a Site to add file attachments, information from other Google applications (like Google Docs, Google Calendar, YouTube and Picasa), and new free-form content. Creating a site together is as easy as editing a document, and you always control who has access, whether it's just yourself, your team, or your whole organization. You can even publish Sites to the world. The Google Sites web application is accessible from any internet connected computer.

Company intranet · Team project · Employee profile · Classroom · Student club

**Quick facts**

**Single-click page creation**
Creating a new page for your Google Site just takes the click of a button.

**No HTML required**
Creating a Google Site is as easy as editing a document, which means there's no markup language for you to learn – just get started.

**Make it your own**
Our customization options let you give your Google Site your own look and feel

**Get started with templates**
We offer a growing list of page types – web page, announcements, file cabinet, dashboard and list – to help you get started with your Google Sites pages.

**Upload files and attachments**
Use the file cabinet to upload files up to 10MB in size. Each Google Apps account receives at least 10GB of storage in Google Sites. Google Apps Premier and Education editions get an additional 500MB for each user account.

**Embed rich content**
Google Sites is integrated with other Google products, so you can insert videos, docs, spreadsheets, presentations, photo slide shows, and calendars directly onto your Google Sites pages.

**Work together and share**
Our permission settings let you designate owners, viewers and collaborators (meaning they can edit pages) for your site. And you can make your Google Sites available to just a few people, your entire organization, or the world.

**Search with Google**
You can search across Google Sites pages and content using powerful Google search technology. You'll find specific pages and documents instantly, the same way you would on Google.com.

©2011 Google - Terms of Service - Privacy Policy - Help Center

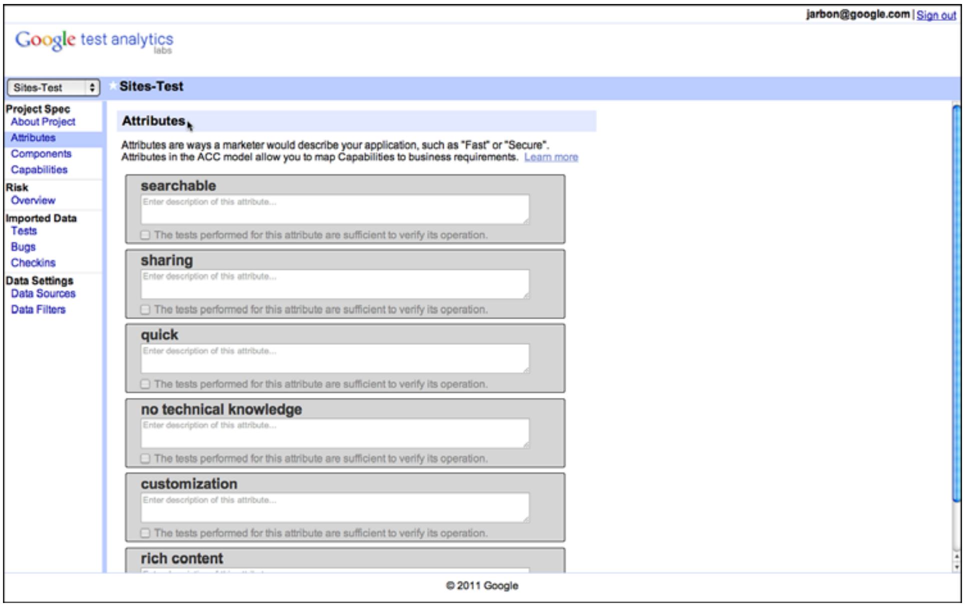
Indeed, most applications that have some sort of Getting Started page or sales-oriented literature often do the work of identifying attributes for you. If they do not, then simply talking to a salesperson, or better yet, simply watching a sales call or demo, gets you the information you need.

Attributes are there for the taking. If you have trouble enumerating them in anything more than a few minutes, then you do not understand your product well enough to be an effective tester. Learn your product and listing its attributes becomes a matter of a few minutes of work.

If you have trouble enumerating attributes in anything more than a few minutes, then you do not understand your product well enough to be an effective tester.

At Google, we use any number of tools for documenting risk; from documents to spreadsheets to a custom tool built by some enterprising engineers called *Google Test Analytics* (GTA). It doesn’t really matter what you use, just that you get them all written down (see Figure 3.3).

**FIGURE 3.3** Attributes for Google Sites as documented in GTA.



Sites-Test

Attributes

Attributes are ways a marketer would describe your application, such as "Fast" or "Secure".
Attributes in the ACC model allow you to map Capabilities to business requirements. Learn more

searchable
Enter description of this attribute...
☐ The tests performed for this attribute are sufficient to verify its operation.

sharing
Enter description of this attribute...
☐ The tests performed for this attribute are sufficient to verify its operation.

quick
Enter description of this attribute...
☐ The tests performed for this attribute are sufficient to verify its operation.

no technical knowledge
Enter description of this attribute...
☐ The tests performed for this attribute are sufficient to verify its operation.

customization
Enter description of this attribute...
☐ The tests performed for this attribute are sufficient to verify its operation.

rich content

## C is for "Component"

Components are the nouns of the system and the next target of enumeration after the attributes are complete. Components are the building blocks that together constitute the system in question. They are the shopping cart and the checkout feature for an online store. They are the formatting and printing features of a word processor. They are the core chunks of code that make the software what it is. Indeed, they are the very things that testers are tasked with testing!

Components are the building blocks that together constitute the system in question. They are the core components and chunks of code that make the software what it is.

Components are generally easy to identify and often already cast in a design document somewhere. For large systems, they are the big boxes in an architectural diagram and often appear in labels in bug databases or called out explicitly in project pages and documentation. For smaller projects, they are the classes and objects in the code. In every case, just go and ask each developer: "What component are you working on?" and you will get the list without having to do much else.

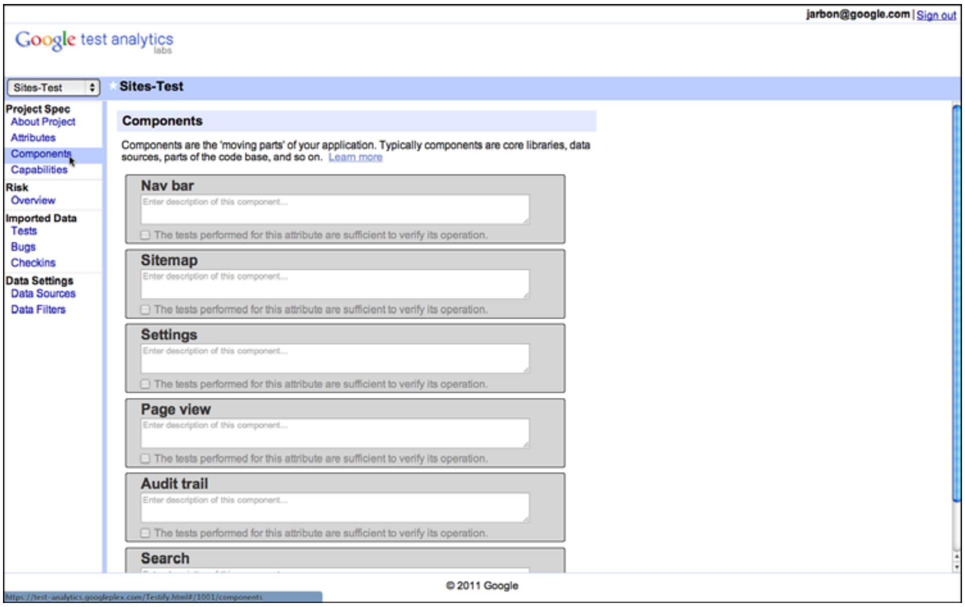
As with attributes, the level of detail in identifying components of the product is critical. Too much detail and it becomes overwhelming and provides diminishing returns. Too little detail, and there's simply no reason to bother in the first place. Keep the list small; 10 is good and 20 is too many unless the system is very large. It's okay to leave minor things out. If they are minor, then they are part of another component or they don't really matter enough to the end user for us to focus on them.

Indeed, for both the attributes and the components, spending minutes tallying them should suffice. If you are struggling coming up with components, then you seriously lack familiarity with your product and you should spend some time using it to get to the level of a power user quickly. Any actual power user should be able to list attributes immediately and any project insider with access to the source code and its documentation should be able to list the components quickly as well. Clearly we believe it is important for testers to be both power users and, obviously, project insiders.

Finally, don't worry about completeness. The whole ACC process is based on doing something quick and then iterating as you go. If you miss an attribute, you might discover it as you are listing the components. Once you get to the capability portion, which is described next, you should shake out any attributes or components you missed earlier.

The components for Google Sites appear in Figure 3.4.

**FIGURE 3.4** Components for Google Sites as documented in GTA.



**Components**

Components are the 'moving parts' of your application. Typically components are core libraries, data sources, parts of the code base, and so on. Learn more

**Nav bar**
Enter description of this component...
☐ The tests performed for this attribute are sufficient to verify its operation.

**Sitemap**
Enter description of this component...
☐ The tests performed for this attribute are sufficient to verify its operation.

**Settings**
Enter description of this component...
☐ The tests performed for this attribute are sufficient to verify its operation.

**Page view**
Enter description of this component...
☐ The tests performed for this attribute are sufficient to verify its operation.

**Audit trail**
Enter description of this component...
☐ The tests performed for this attribute are sufficient to verify its operation.

**Search**

© 2011 Google

https://test-analytics.googleplex.com/Testify.html#/1001/components