

The 10-Minute Test Plan

James A. Whittaker, Google

// Test plans are perhaps the least appreciated of all supporting software development artifacts, so let's spend as little time as possible on them—say, 10 minutes. //



SOFTWARE IS UBIQUITOUS. It runs corporations, governments, and markets, as it has for decades, but it's also central to personal and social communication, entertainment, travel, and leisure. Software is no longer a tool limited to the 40-hour work week; it's a service consumed at work, home, and play. It isn't a stretch to say that users not only depend on software but also fall in love with it. Developing beautiful and engaging software creates an emotional bond between your application and its users.

But how many users have ever fallen in love with a test plan? Of all the supporting artifacts of software development (specifications, designs, product plans, and so forth), the lowly test plan is probably the hardest to love. Test plans are written, reviewed, and often left to rot in place as the hustle and bustle of software development progresses

and shifts the focus—rightly so, in my opinion—to the care and feeding of source code. After all, the source code becomes the product that users covet. Spending a moment more on a test plan than is absolutely necessary is a moment taken away from making the code the best it can be.

Writing voluminous test plans may be a prescribed burden in the life of software engineers working in regulated industries or dealing with safety-critical issues. But for the compressed shipping cycles of modern mobile and Web apps, we need something simpler. A solution that compresses the time spent developing and maintaining a test plan might even have useful takeaways for those who require more rigor and adherence to standards.¹

I See Dead Test Plans

When I first arrived at Google, my ex-


perience suggested that test plans were created with an uncritical eye. They're simply expected. The first question asked when the subject of testing comes up is, "What's the plan?" Then inevitably someone accepts the task of creating it. The process of doing so creates value: writing the test plan forces us to think through testing problems and potential solutions. But the actual value is in the test plan activity and not the test plan artifact, which often sits unused once development starts in earnest. Test planning is an incredibly useful exercise, but the plan itself is of questionable value. Keeping a test plan up to date through all the various requirements, specification, and product changes is simply of too little value to be worth anyone's time.

As someone who's worked in the testing field for much of my career, I wanted to find out why test plans die so prematurely. If test planning is truly of little value, should we even bother? Why grant tenure to a low-value process simply because it's familiar? And if test planning is valuable, how do we get to that value quickly and efficiently without generating an artifact that's not worth maintaining? I wanted to rethink test planning from the ground up.

A theoretical exercise was outside the skill set of my engineering team. Asking them to sit around and brainstorm about why our test plans were dying would have produced scant insight. Instead, I took a more aggressive approach, using a very industry-oriented technique: I gave a group of people who report to me a nebulous task with unrealistic time constraints.

Ten Minutes and No More

I chose my directions carefully: "Build a test plan for Google App Engine; you have 10 minutes." Such a directive has no ambiguity. The peo-



ple I asked to participate in this experiment were used to taking work requests from me. Test planning was an activity they understood. App Engine was a product they understood. As I closed the door, sequestering the team in their conference room, there were no questions to ask.

Ten minutes later, I opened the door and there was, unsurprisingly, no test plan. The excuses were predictable: “We didn’t think you really meant 10 minutes.” “That isn’t enough time to even document our intent.” “Dude, seriously?”

I reset the clock but changed the application: “10 minutes, Chrome Web Store.... Go!”

You get the idea. Every 10 minutes, I changed the application target, but the task remained the same: build a test plan. As the minutes ticked by, my team got the message. They quickly discarded any idea of describing the problem, the application, or finally, anything at all. There wasn’t enough time.

They set aside prose in favor of bulleted lists. They learned to ignore any information that wasn’t relevant to testing—meaning that if it wouldn’t appear in an important test case, ignore it. If it didn’t affect testing in some fundamentally important way, leave it out. By the fifth attempt, they were able to get 80 percent through the test-planning process in 30 minutes or less, documentation included. Not exactly a 10-minute test plan, but a success nonetheless. We boiled test planning down to its very essence.

Attributes, Components, and Capabilities

In all the documentation and notes created during both the false-start and the 80-percent-complete test plans, three requirement categories surfaced.

First, what application *properties*

need verification? App Engine needed to be available, secure, and responsive. Chrome Web Store needed to be intuitive, appealing, and worthwhile. In other words, the adjectives and adverbs that describe why a user would use the service in the first place are an important part of the planning process. They represent the characteristics we’re trying to validate during testing. We must understand the system properties that need verification.

Second, what *parts* of the application need verification? App Engine has components for creating, hosting, and scaling applications. Chrome Web

Store has a data store, discovery engine, and download manager. These are the nouns that name components of the system under test. We must know the parts of the application that require testing attention.

Finally, what *purpose* does the application serve for users? App Engine lets users write code, serve application instances, and scale to meet demand. In other words, the verbs describing what capabilities users receive are the actions that eventually get translated into test cases. We must know what actions the system performs.

We experimented with various combinations of these three properties and settled on a process called ACC: attributes (the adjectives and adverbs), components (the nouns), and capabilities (the verbs), to be documented in that order and, we hoped, in 10 minutes or less. ACC has seven guiding principles:

- *Avoid prose and favor bulleted lists.* Not all testers possess the skills to adequately capture a product’s purpose or testing needs in narrative form. Prose can be hard to read and easy to misinterpret.
- *Don’t bother selling.* A test plan isn’t a marketing document or a place to talk about the importance of a product’s market niche or how cool it is. Test plans aren’t for customers or analysts; they’re for engineers.
- *No fluff.* Test plans aren’t school term projects with specific length expectations. Bigger is not better.

Every 10 minutes, I changed the application target, but the task remained the same: build a test plan.

A plan’s size is related to the size of the testing problem, not the author’s propensity to write.

- *If it isn’t important and actionable, don’t put it in the plan.* Not a single word should garner a “don’t care” reaction from a potential stakeholder.
- *Make it flow.* Each section should expand on earlier sections so that readers can stop at any time and have a clear picture of the product’s functionality. If they need more detail, they just continue reading.
- *Guide a tester’s thinking.* A good planning process helps a tester think through functionality and test needs and so leads logically from higher-level concepts to lower-level details that engineers can implement directly.
- *The outcome should be test cases.* By the time the plan is complete,

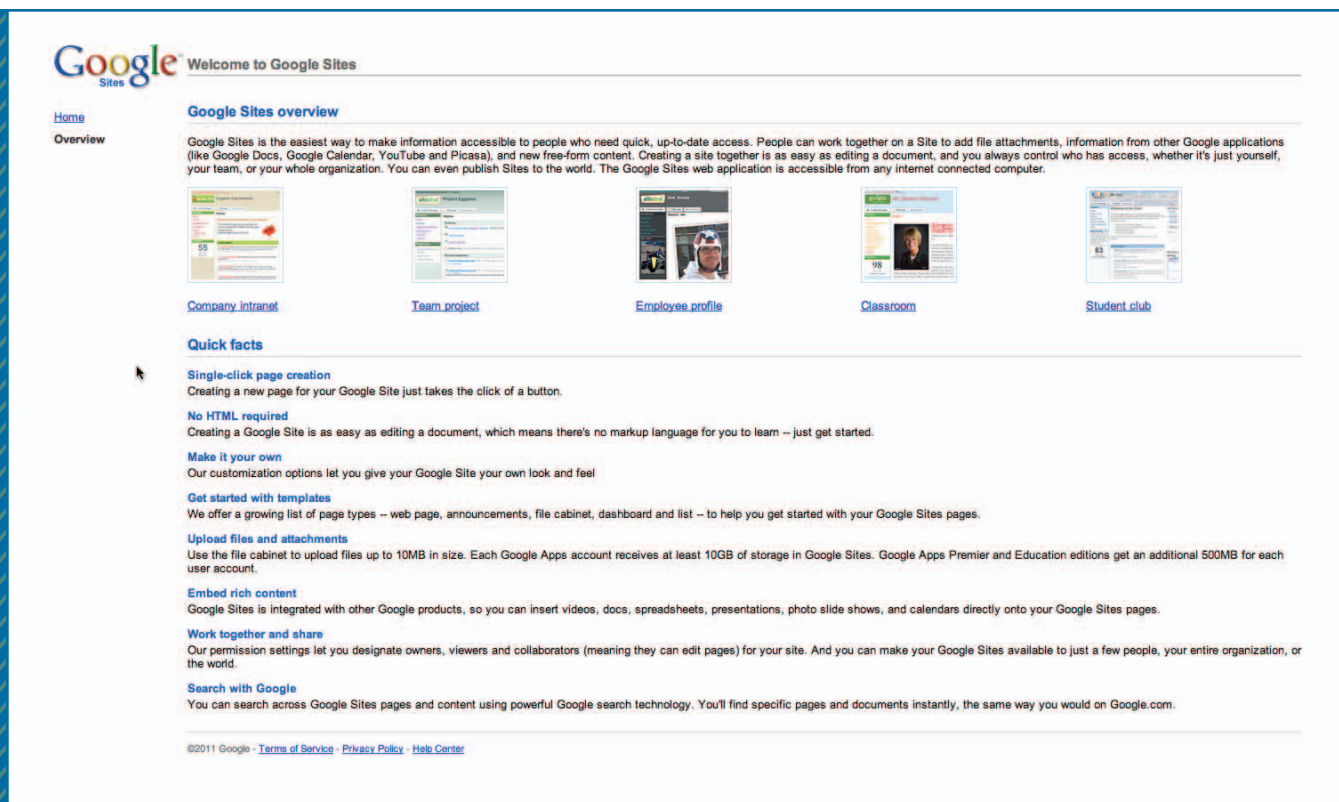


FIGURE 1. “Welcome to Google Sites” screenshot. Most of the product attributes are listed under “Quick facts.” Many end-user applications include similar pages that effectively identify attributes for testing.

it should clarify not just what testing needs to be done but also how to write the test cases. A plan that doesn’t lead directly to tests is a waste of time.

This last point is crucial. If the test plan doesn’t have enough detail to tell us what test cases we need to write, it hasn’t served its primary purpose of helping test the application we’re building. Test planning should put us in a position to know exactly what tests must be written. ACC accomplishes this by guiding the planner through three views of a product: its attributes, components, and capabilities.

A Is for Attributes

Attributes describe 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? We’re not looking to either justify or explain these things, only to label them. Presumably, the product planners have done their job in coming up with a product that will matter in the marketplace. From a testing perspective, we just need to capture and label attributes so we can ensure they’re accounted for when we test the product.

Attributes are the qualities and characteristics that promote the product and distinguish it from the competition. In a way, they explain why people would choose to use your product over a competitor’s. For example, Chrome is designed to be fast, secure, stable, and elegant, so these are the attributes ACC is trying to document. Eventually,

we want to attach test cases to the attributes, so we know how much testing we’ve done to demonstrate their manifestation in the product.

Typically, a product manager will have a hand in narrowing down the list of system attributes. Testers often get this “list” by reading the product requirements document or the team’s vision/mission statement or even by simply listening to a salesperson describe the system to a prospective customer. Indeed, at Google, we find 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, and you’re getting the right mindset to list the attributes.

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

- *Keep them simple.* If it's taking more than a few minutes, you don't understand the product well enough.
- *Keep them accurate.* Make sure they come from documentation or marketing information that your team already accepts as truth.
- *Keep moving.* Don't worry if you missed something. If it's not obvious later that you missed something, it probably wasn't that important.
- *Keep it short.* No more than a dozen attributes is a good target. We boiled Chrome's operating system down to 12 key attributes and, in retrospect, should have shortened that list to eight or nine.

As an example, consider the attributes for the Google Sites product, which is a freely available application for building a shared community website. Google Sites, as you'll find with many end-user applications, is kind enough to give you most of its attributes in its own documentation, as Figure 1 shows in the "Quick facts" list.

Indeed, most applications have some sort of "Getting started" page or sales-oriented literature that will often do the work of identifying attributes for you. If they don't, then talking to a salesperson or, better yet, watching a sales call or demo, will get you the information you need.

At Google, we use any number of tools for documenting ACC, from specification-like documents to spreadsheets to a custom tool built by some enterprising engineers called Google Test Analytics (gTA). Figure 2 shows the attributes for Google Sites as documented in gTA.

C Is for Components

The next enumeration targets are nouns—the component building blocks



FIGURE 2. Google Sites attributes as documented in the Google Test Analytics (gTA) tool. "Searchable," "sharing," "quick," and so forth are the actual attributes we selected to demonstrate in subsequent testing.

that together constitute a system and implement the attributes. Components are the shopping cart and the checkout feature for an online store. They're the formatting and printing features of a word processor. They're the core chunks of code that make the software what it is. Indeed, they're the very things that testers are tasked with testing.

Components are generally easy to identify and often already cast in a design document somewhere. For large systems, they're the big boxes in an architectural diagram and often appear in bug database labels or get called out explicitly in project pages and documentation. For smaller projects, they're the code classes and objects that developers are creating. You'll get the list without having to do much else if you

just ask each developer, "What component are you working on?"

As with attributes, the level of detail in identifying product components is critical. Too much detail becomes overwhelming and brings diminishing returns. Too little detail, and there's simply no reason to bother in the first place. Keep the list small: 10 components are good, 20 are probably too many unless the system is very large. It's okay to leave minor things out: if they're minor, then they're either part of another component or lacking enough end-user value to focus on them.

Indeed, you should be able to enumerate both attributes and components in minutes. If you're struggling to come up with components, then you're not familiar enough with your product and



FIGURE 3. Google Sites components as documented in gTA. “Nav bar,” “Sitemap,” and so forth are blocks of actual code modules. Each of these components is visible from the Sites UI or immediately available in specification documents.

need to spend some time using it to get quickly to a power-user level. 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.

Finally, don’t worry about completeness. The whole ACC process is based on doing something quickly and then iterating as you go. If you miss an attribute, you might discover it as you’re listing the components. ACC’s capabilities process should shake out any attributes or components you missed earlier.

Figure 3 shows how Google Sites components are documented in gTA.

C Is for Capability

Capabilities are the system’s verbs.

They represent the actions the system performs in response to user inputs. Indeed, users choose your software because they want some specific functionality that it provides.

Chrome, for example, has capabilities to render a webpage, play a Flash file, synchronize between clients, and download a document. These capabilities and many more represent the browser’s full functionality. A shopping app, on the other hand, has capabilities to perform a product search and complete a sale. If an application can perform a task, then the task is one of its capabilities.

Capabilities lie at the intersection of attributes and components. Components perform some function to satisfy a product attribute and the result gives

the user a capability. Chrome renders a webpage fast and plays a Flash file securely. If your product does something that isn’t covered by an attribute-component intersection, it probably warrants raising the question of why you’re bothering to include it in the product. A capability that doesn’t serve a core product value sounds like fat that you can trim—either that or an explanation for the capability exists but you don’t understand it. Not understanding your product is unacceptable.

Here are some example capabilities for an online shopping site:

- *Add/remove items to/from the shopping cart.* A cart component intersects with an intuitive UI attribute.
- *Collect credit card and verification data.* The cart component meets the convenient and integrated (with the payment system) attributes.
- *Processes monetary transactions using HTTPS.* The cart component meets the secure attribute.
- *Provide suggestions to shoppers based on the products they’re viewing.* The search component meets the convenient attribute.
- *Calculate shipping cost.* The FedEx integration component meets the fast and secure attributes.
- *Display available inventory.* The search component meets the convenient and accurate attributes.
- *Defer a purchase for a later date.* The cart component intersects with the convenient attribute.
- *Search for items by keyword, stock-keeping unit, and category.* The search component intersects with the convenient and accurate attributes. (In general, we prefer treating each search category as a separate capability.)

Obviously, the capabilities list can be long. If you feel as if you’re listing

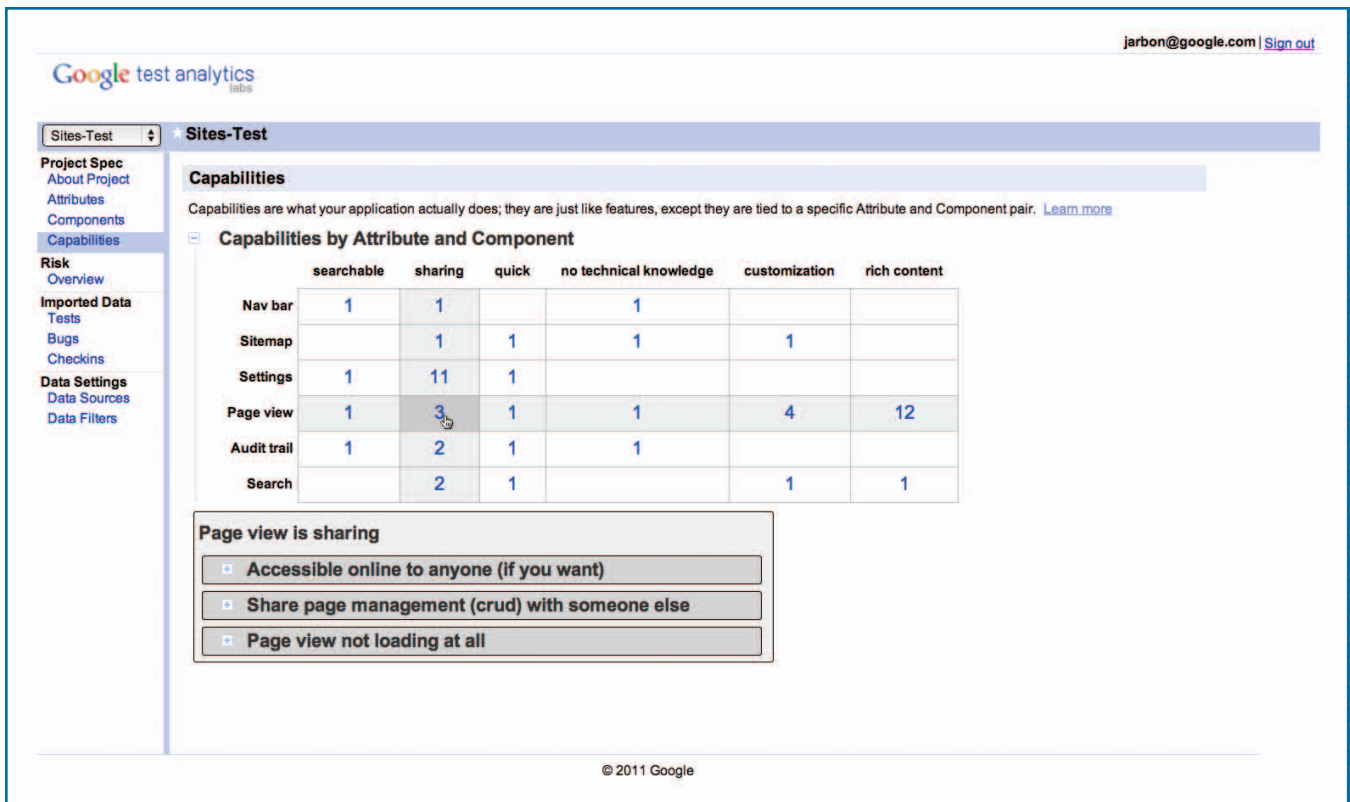


FIGURE 4. Capabilities grid. The numeric value indicates the number of capabilities that a particular component (y-axis) provides to satisfy a particular attribute (x-axis). The higher the number, the more test points for that intersection. Blank entries indicate no attribute-component link and therefore no test requirement.

everything you could test, then you're getting the hang of ACC—to list, quickly and succinctly, the most important system capabilities that need verification to be in working order.

Capabilities are generally user-oriented and written to convey a user's view of what the system does. Whereas brevity rules the attribute or component ACC stages, capabilities should describe everything the system can do. They're far more numerous than attributes and components, and their number increases with an application's feature richness and complexity. Smaller applications I've worked on at Google have dozens of capabilities, and the more complicated systems tend to have hundreds (Chrome OS has over 300, for example).

Testability

Most importantly, capabilities must be testable. They're verbs because they require action on our part—specifically, to write test cases that will determine whether each capability is implemented correctly and whether the user will find the experience useful. This is the primary reason to write them in an active voice.

Capabilities aren't meant to be test cases that contain all the information necessary to run as actual tests. They don't require exact input values and data. If the capability is to let users shop, the test case will specify what they shop for. Capabilities are general concepts of actions the software can take or a user can request. They imply tests and values but aren't tests themselves.

Continuing with the Google Sites example, Figure 4 shows a grid with attributes across the x-axis and components across the y-axis. This is the way ACC links capabilities back to attributes and components. The large number of empty squares is typical because not every component has an impact on every attribute. For example, only some of Chrome's components are responsible for making it fast or secure; the others will have blanks at the attribute intersection points, representing no impact and therefore no need to test this particular attribute-component pair.

Each row or column in the capabilities grid represents a slice of functionality that's related in some fashion. A single row or column is a good way to break the application's functionality

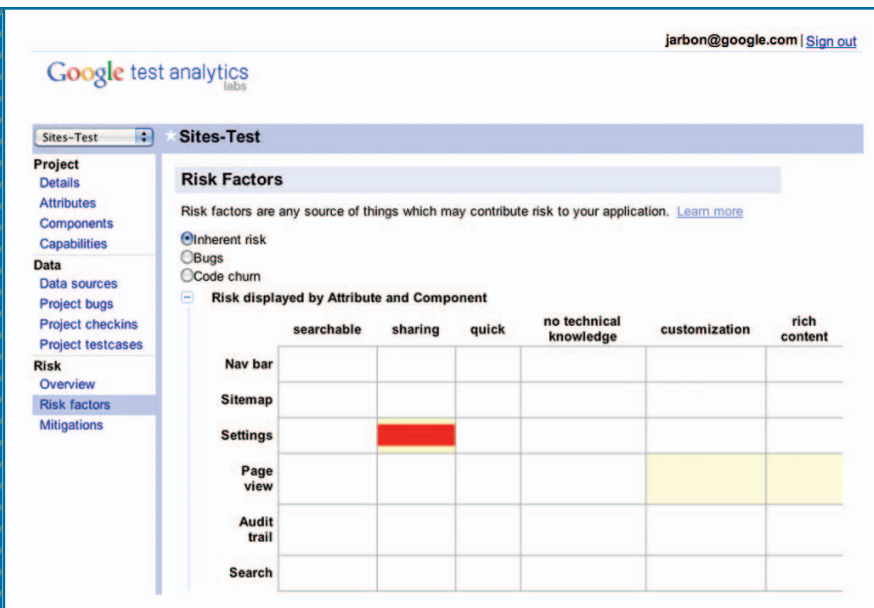


FIGURE 5. Heat map of risk areas per attribute-component pair, for inherent risk, bugs, and code churn. The darker the box, the greater the risk and thus the more testing required. When tests are successfully executed, the dark areas become lighter.

into testable sessions. A test manager might assign each row to a separate test team or have a bug bash to really hit a row or column hard. The grid also reveals targets for exploratory testing and, when each exploratory tester takes a different row or column, helps manage overlap and improve coverage.

The numeric values in Figure 4 represent the number of capabilities provided by the component on that row to satisfy the attribute in that column. The higher the number, the more test points for that particular intersection. For example, the page-view component addresses the sharing attribute in three capabilities:

- Make the document accessible to collaborators.
- Share page-management duties with a collaborator.
- View collaborator position within a page.

These capabilities must be tested for

the page-view/sharing pair. We can either write test cases for them directly or test a combination of capabilities by combining them into a larger use case or test scenario.

Documenting Capabilities

Writing good capabilities requires some discipline. Three properties we've found to be useful include, first, writing a capability as an action that conveys the sense of the user accomplishing some activity, as in the page-view/sharing examples.

Second, a capability should provide enough guidance for a tester to understand the variables involved in writing test cases for the behavior it describes. For example, "Process monetary transactions using HTTPS" requires the tester to understand what types of monetary transactions the system can perform and to define a mechanism that validates whether the transaction occurs over HTTPS. Obviously, there's a great

deal of work to be done here. If you believe that some monetary transactions might be missed by, say, a new tester on the team, then you need to replicate this capability to expose the various transaction types; if not, then the general level of abstraction is good enough.

Likewise, if HTTPS is something the team understands well, this capability is fine as it's worded. Capabilities are supposed to be abstract, so don't fall into the trap of trying to document every detail as a capability. Leave it to the test cases or to the exploratory testers themselves to provide that level of detail. (Leaving such variations to the tester also creates variety in how capabilities are interpreted and cast into actual test cases, which in turn translates to better coverage.)

Finally, a capability should be composable with other capabilities. In fact, a user story² or use case³ (or whatever terminology you may prefer) should be describable in a series of capabilities. If you can't write a user story with only the existing capabilities, then either some capabilities are missing or they're written at too high an abstraction level.

Transforming a set of capabilities into user stories is an optional interim step that can add a great deal of flexibility to testing. In fact, several Google groups prefer more general user stories over more specific test cases when engaging with external contractors or when organizing a crowdsourced⁴ exploratory testing effort. Test cases that are specific can become boring as a contractor executes them over and over, whereas a user story provides enough leeway in deciding specific behaviors to make testing more fun and less prone to mistakes.

Whether your ultimate goal is user stories, test cases, or both, we developed three general guidelines at Google for translating capabilities to test cases:

- *Every capability should be linked to at least one test case.* If the capability is important enough to document, it's important enough to test.
- *Many capabilities require more than one test case.* Variation in the inputs, input sequences, system variables, data used, and so forth require multiple test cases. The attacks in *How to Break Software*⁵ and the tours in *Exploratory Software Testing*⁶ offer guidance on selecting test cases and thinking through data and inputs in ways that are more likely to turn a capability into a test case that finds a bug.
- *Not all capabilities are equal;* some are more important than others.

Once the ACC document is complete, it specifies everything we could test if budget and time weren't a problem. Given that both are major problems, the next step is to associate the capabilities with a risk and distinguish their importance.

Prioritization and Risk

Our 10-minute goal leaves too little time to stack-rank the capabilities to determine the order in which we convert them to tests. A more realistic goal is to assess the relative importance of each capability to the overall mission. We do this with two simple variables: expected frequency of failure and failure impact.

With regard to expected frequency, as a function of complexity, usage rates, and so forth, how often do you anticipate the capability to fail? A capability that establishes a network connection might be expected to fail fairly often. Capabilities that represent the work of a new developer or that depend on flaky external resources may also fail often.

With regard to impact, if the capability does fail, how dire are the consequences to a user? If all the user has to

do is reenter a value, the impact is low. If the user loses data, the impact is high.

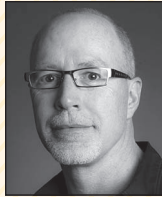
Together, these variables represent the minimum amount of thinking a tester must perform from a prioritization and risk perspective. If we give these two factors numerical weights and sum them, we can group the capabilities into higher- and lower-risk categories and make more organized decisions about test coverage. We can even get views of testing needs based on these values, as Figure 5 shows.

Obviously, regulated or safety-critical software must do far more than what I've prescribed here. However, the core of test planning is about discovering and documenting testing effort, and the 10-minute test plan represents the core activity that must be performed.

A complete ACC is a guide for either performing testing or simply organizing it. It lets you clearly scrutinize each area for the type of testing you want to do—for example,

- automate tests for certain groups of high-risk capabilities;
- write test scripts for specific groups of capabilities that you want to test frequently;
- select capability sets for exploratory testing without any further documentation beyond the ACC; and

ABOUT THE AUTHOR



JAMES A. WHITTAKER was an engineering director at Google before returning to Microsoft as a development manager. His research interests are in developer platforms and application development. Whittaker has a PhD in computer science from the University of Tennessee. He's written several books, including *How to Break Software*, its series follow-ups, and *Exploratory Software Testing*; he also wrote cowrote *How Google Tests Software* (Addison-Wesley, 2012). Contact him at docjamesw@gmail.com.

- outsource or crowdsource certain groups of capabilities.

ACC makes it easy to ensure that each type of testing has minimal overlap or that high-risk areas have purposeful overlap. The idea is to be mindful of all the testing that takes place either by actual testers or by collaborating groups such as beta users or a dog food team.⁷ Then you can map the testing back to ACC to see just how well the actual testing covers the testing surface. 🍷

References

1. *IEEE Std. 829: Software Test Documentation*, IEEE, 1998.
2. K. Beck, *Extreme Programming Explained*, 2nd ed., Addison-Wesley Professional, 2004.
3. A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley Professional, 2000.
4. J. Winsor, "Crowdsourcing: What It Means for Innovation," *Business Week*, 15 June 2009, http://innovbfa.viabloga.com/files/BusinessWeek___Crowdsourcing___What_it_means_for_Innovation___june_2009.pdf.
5. J. Whittaker, *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2002.
6. J. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, Addison-Wesley Professional, 2009.
7. W. Harrison, "Eating Your Own Dog Food," *IEEE Software*, vol. 23, no. 3, 2006, pp. 5–7.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.