What

THREE months worth of tests. ONE month to deliver. Something has to give. When you can't possibly test it all, what do you do? FIVE questions you need to ask when deciding

WHAT DO YOU DO WHEN YOUR TESTING PROJECT IS SQUEEZED for time? When code drops are late? When deadlines are advancing?

Do you play "if only" ? "If only we had automated more tests" or "If only our tests were tougher, so that we didn't have to run so many to shake out the bugs." The problem with the "if only" game is that by the time you play it, it's too late to make changes.

Do you faithfully follow the plan you already have in place and then stop testing when the clock runs out? What if the tests you leave undone are more valuable than the tests you run?

I recommend instead that you purposefully decide what not to test. When project realities change, you just alter your plan accordingly. To do that successfully, you have to plan in a way that supports change, and you have to know *how* to change.

PLAN IN A WAY THAT SUPPORTS CHANGE

I always start with a list of great testing ideas that apply to the project. A test idea is a description of a kind of test that needs to be run. See Figure 1a (on page 32) for a sample list.

Let's look at some of the fields in the tables. *ID* is just a reference number. *Source* is the source of the test idea. There are many sources for these testing ideas; some that I use regularly are shown in Figure 1b.

Description briefly details each idea. Notice that the ideas aren't specific enough to be test procedures, but they are specific enough that a tester easily could flesh them out. We will invest in detailing an idea more specifically if (and only if) we decide to use it. If we defer or reject a testing idea, there is very little benefit to further detailing or elaboration. Imagine the waste of spending several days detailing a test procedure only to decide not to implement the associated testing idea.

In testing, size matters—hence the field *size*. I consider the size of a testing idea to be an estimate of the reasonable amount of effort you would have to invest to satisfactorily implement that idea given the state of the software under test and the skills and competency of the testing team. On a recent project we used a very simple scale: small, medium, and large. *Small* is a testing idea that takes less than ninety minutes to implement. *Medium* is a testing idea that takes a day to implement, and *large* is a testing idea that takes about a week to implement. These time scales assume an experienced tester and software that is relatively bug-free. If test ideas are larger than *large*, we split them or otherwise reorganize them.

The final field summarizes an idea's *importance*. The bottom line is that a test idea is not important if the bug it finds will not be fixed. For example, if we decide not to fix spelling mistakes in final system testing, then any test idea related to spell checking is unimportant. A testing idea *is* important when the information it provides contributes to the decision to deploy or ship the software. In a recent project, we used another simple scale, high, medium, and low—where *high* implied knowledge of the test passing was critical and *low* implied a decision to ship or deploy the software could be made without knowledge of the test result. Anything in between was considered *medium*.

CAPTURE TESTING IDEAS

To fill in the table, the test lead should arrange a structured brainstorming session designed to capture testing ideas. To help people stay focused and on track, it is a good idea to have an experienced moderator run the

Not to Test

BY ROBERT SABOURIN

ID	SOURCE	DESCRIPTION	SIZE	IMPORTANCE
00100	REQ	Test file download feature to all supported platforms		
00200	REQ	Test software installation on all supported platforms		
00600	FM	Test installation on systems without sufficient disk space available		
00700	FM	Test installation on nonsupported operating system		
00800	FM	Test software operation with insufficient system memory		
01100	USAGE	User migrates from recent version to new version of software		
01200	USAGE	User migrates from very old version to new version of software		
01300	USAGE	Typical usage scenario (A)		
01600	QF	Performance of transaction A on specific workstation (same or better than previous release)		

▲ **Figure 1a: This table lists test ideas for a project. The abbreviations in the source column are spelled out in a separate source table, as shown in Figure 1b.** ►

TESTING IDEA SOURCE	COMMENT
Requirements (REQ)	*What should the system do?* What capabilities should it have? What are its performance criteria? Under what constraints does it operate? (And so on.)
Failure Modes (FM)	*What can break and fail?* What environment does the system run in? What data can be wrong, missing, or incorrectly structured? Does the system have to synchronize with other systems or events? Are there timing issues?
Usage Scenarios (USAGE)	*What do people do when they use the system?* How can we model operational workflow? What other systems and manual processes do they use while using our system?
Quality Factors (QF)	*What characteristics must the system possess to have quality?* Performance, usability, maintainability, scalability: the usual suspects.

meeting. Select participants from among those people who will be working on the project. Try to mix up roles. Include some developers, testers, writers, GUI designers, etc. (This session doesn't have to be the only source of testing ideas. Whenever someone thinks of a new one, it can be added to the list.) This group of people might not be able to judge the size and importance of testing ideas, but the test lead can gather that information from important project stakeholders. I like to use a spreadsheet program to help organize my testing ideas, but index cards, with one idea per card, work just as well.

When it's time to decide which tests will not be run, make sure the stakeholders participate. I find groups of three to be best: a Product Manager, a Development Lead, and a Test Lead. Remember, deciding what not to test does not have to be final. We can decide not to implement a testing idea based on the information we have now, but if new information comes our way in the future, we can reconsider using the testing idea. There are no wrong testing ideas. They all have some value even if they are not implemented.

KNOW HOW TO CHANGE

Your list of test ideas will help focus your testing on those bugs that key project stakeholders consider urgent. As the schedule tightens, you can be sure you're not dropping what's important. Even if the schedule isn't tightening, as you test, you can triage testing ideas and decide what to test and what not to test.

There are more questions to ask about test ideas, questions whose answers will help you make such decisions. They fall into two broad categories: Is this the *right* test to run? Is this the right test to run *now?* In several of my recent projects with severe time and budget constraints, we used five specific questions:

1. Are ideas based on credible information?

2. Is the system ready for this test idea?

3. Are there alternative test ideas that could give the same value?

4. Can we combine this test idea with others?

5. Can we choose subsets of this test idea?

ONE

ARE IDEAS BASED ON CREDIBLE INFORMATION?

I use testing ideas based on credible data before I use ideas based on guesses or intuition.

Before we embark on prioritizing a testing idea, it is important to look at the credibility of the source of the idea and of our understanding of the size and importance of it. On large Internet applications many testing ideas come from assumptions of how users are going to be using the systems. An example could be browser types. How many users will be using Internet Explorer, Netscape, Opera, or other browsers when accessing the system? Generally, a value is

provided indicating either an expected absolute value (we expect 10,000 users to be using Opera) or a relative percentage (we expect 30 percent of the users to be using Opera). Sometimes information may look like it is verifiable factual data when it is really a dressed-up wild guess.

So how do you rate credibility? First, gauge the credibility of any fact a test idea is based upon. Next, assign a credibility score—an example scale is 1 to 10, where 10 represents absolute verifiable fact and 1 represents a totally wild guess. When the credibility score is too low, get more information before prioritizing the testing idea. I find other sources, look at past projects, or speak to more peers. In the meantime, leave the test idea out until the credibility of source information is improved.

TWO
IS THE SYSTEM READY FOR THIS TEST IDEA?

In some cases, a testing idea may be ahead of its time. To paraphrase JFK: Ask not (just) whether your test is ready for the system but whether the system is ready for your test.

A recent project I was testing involved interaction between a Web server and client device drivers being downloaded and updated. Three testing cycles were organized on different servers.

We found a lot of bugs on the first cycle of testing. In the second cycle of testing we expected to find fewer bugs—but in fact we found many more bugs. A lot of features worked well on the first cycle but started to fail on the second cycle.

We investigated the problem and discovered that the first cycle of testing was useless. All of the server code needed a rewrite to move to the preproduction servers! We had wasted our valuable testing time. During the first testing cycle the client side may have been ready to test—the server side was not even close.

The moral of the story is that we should have assessed the readiness of the system for testing. We should have skipped a lot of the tests—and done them when the time was right.

THREE
ARE THERE ALTERNATIVE TEST IDEAS?

Testing provides a great deal of information about the status of a software system under development. Each testing idea can provide some useful information. But sometimes a testing idea provides information that we could assess in other ways.

When triaging testing ideas, we should ask ourselves these questions: Is there any other way I can learn the same information about the system? What alternatives are available?

For example, you may want to confirm that the application prints reports correctly on all supported printers. You could develop a series of test cases, which print reports on all the target printers and then, to ensure correctness, compare the printed reports to one another or to a master.

Alternatively, you could implement a test scenario that includes data entry, processing, and report printing, varying the printer for each test scenario. There is some setup required to ensure the printers are connected and configured correctly, but in this case you mimic what a user will typically do with a system—including printing the report. If report printing fails badly on a supported printer, the bug will show up when running the scenario.

In this case, I would focus on the test scenario rather than on creating a customer test suite just for printers. Using the test scenario would be faster and provide more information.

FOUR
CAN WE COMBINE THIS TEST IDEA WITH OTHERS?

When triaging testing ideas, don't reject an idea outright; ask whether it can be combined with another idea. Combining two testing ideas may generate more value for a small increase in cost.

In the 1990s, as graphical user interfaces became more popular, test projects began to include a GUI test plan. The GUI test plan included a procedure to systematically exercise all menus, buttons, and controls, as well as any graphical elements, mouse, or keyboard events. GUI test plans were long and tedious, but they certainly helped us find many important GUI bugs.

Today, GUI technologies are better integrated into development environments and operating systems. Fewer GUI bugs are discovered during testing, but they still happen and are important to address. (They are especially common in Web-based technologies using client side scripted technologies, such as Flash or JavaScript.)

GUI testing can be combined with functional testing if you teach all members of the testing team to vary their navigation techniques and to ensure that certain common errors—identified in some sort of common GUI-error checklist—are not present. Some GUI testing will then be done implicitly every time a tester uses the GUI to perform other functional tests.

FIVE
CAN WE CHOOSE SUBSETS OF THIS TEST IDEA?

A testing idea may be too big. For example, I worked on a project to upgrade old versions of a system to a new version. As one test idea, someone proposed exhaustively testing all possible combinations. The system had five op-

Sometimes information may look like it is verifiable factual data when it is really a dressed-up wild guess. So how do you rate credibility?

erating systems, seven software source versions, two languages, two locales, and two different installation options. To try each case once would require 280 test cases ($5 \times 7 \times 2 \times 2 \times 2$). Each test case would take more than half an hour to run (assuming a pass), so even in an ideal world, we were looking at 140 hours of testing.

As an alternative, a practical subset of the upgrade test case was developed. In the subset, certain test cases that added less information or that exercised conditions unlikely to occur were deleted. The resulting test case required much less time than the exhaustive test idea but provided almost exactly the same meaningful information.

It is even possible to dramatically improve a testing idea by reducing the scope or focus of it. In testing server database applications that require migration from old structures to new structures, you might see this testing idea: Confirm that databases from all previous supported versions can be migrated to the current revision. This idea sounds simple but is, in fact, quite broad. It includes testing that a record with a new field in the new revision can migrate (What happens to the new field?); testing the same thing for a removed field; testing the same thing for a field whose data type has changed; and migrating one sample test database for each supported version.

Analyzing what a testing idea really entails allows you to make better decisions about what to test and what to leave out.

When you first try to introduce the concept of test idea triage you may run into resistance from other members of the testing team.

GET STARTED RIGHT

Now that you understand the process for deciding what not to test, how can you implement it? I recommend you begin with a pilot project. In order to succeed, the expectations of all participants must be in sync. Meet participants individually to explain the purpose of the project and the life of a testing idea. Detail how ideas are collected, sorted, evaluated, triaged, implemented, or rejected. Make sure people know that not every idea they mention will be tested. Make sure key stakeholders know that you cannot test everything (it is impossible), but rather you want to focus the limited testing budget on the most important aspects of the project.

I have applied testing idea triage to several projects in different organizations for more than ten years. When you first try to introduce the concept of test idea triage you may run into resistance from other members of the testing team. You must get buy in before trying to impose it on key team players.

Some testers insist on having a complete test plan before a project starts; a complex change request management system is needed to make the slightest modification. These testers are likely to resist any Agile approach. Are they on the right team? Resolving their resistance before implementing test idea triage will save you a lot of grief. If you cannot change the person's attitude, you may need to pick a different project and convince the skeptics by demonstrating success there.

During the project, baby steps should be the order of the day. Implement test idea collection and triage as a daily part of the project work.

Deciding what to test—and what not to test—can be done with a systematic approach to collecting, evaluating, and triaging testing ideas. My experience dictates that it's better to omit a test on purpose than to skip it for no reason other than "we ran out of time." **{end}**

Test Idea Triage

Test ideas are not static. As testing progresses, new ideas are discovered. As testers and developers start using the application from an end-user perspective, usage, scenario, and interoperation testing ideas are generated.

I urge you to set up periodic test triage meetings. These meetings can start as soon as the testing is underway. Triage does not end until the project is completed.

Test idea triage is similar to bug triage or change control boards. I recommend using a small group representing all important stakeholders. As a rule, the same folks who triage bugs and requirements should also be triaging testing ideas.

In the test triage meeting, you will review benefit, size, and credibility of new testing ideas relative to testing ideas already considered. Do we need more information about this testing idea? How credible is the information available? Should the idea be used or not? Should this new idea be combined with an existing idea? Should the new idea be split up?

My experience is that daily test idea triage meetings work well when run first thing in the morning—as a sort of kickoff meeting for the day. Generally, I recommend bug and requirement triage in midafternoon.

Robert Sabourin (rsabourin@amibug.com) is an author, lecturer, and president of AmiBug.Com, Inc., a firm specializing in software management consulting, teaching, and professional development. Robert is the author of the popular children's book I am a Bug, *which explains what SQA folks really do at work.*

page 35
full-page ad
RadView