White Paper

Dealing With the Time Crunch in Software Testing

By Randall W. Rice, CTAL (Full), CTFL-AT, CMT
Rice Consulting Services, Inc.
1608 SW 113th Pl.
Oklahoma City, OK 73170
405-691-8075

http://www.riceconsulting.com
rrice@riceconsulting.com

Abstract

In my research involving thousands of software testers over the last fifteen years, several challenges are mentioned as critical pain points in nearly every interview and survey. One of the most common complaints from software testers is that, "There is no way we can do a complete test in the time we are given."

The mismatch of scope and time causes stress for software testers and often leaves them in a no-win position. They might work very hard in the short amount of time they have been allotted; only to be blamed at times for not finding certain defects once the software is released to users.

Another impact of time constraints is that schedule delays in development and other project activities can erode the planned test schedule. In those situations, the overall delivery deadline remains firm even when the testers get access to the software later than planned. This schedule crunch is another reason testers feel stress and may be forced to skip some tests.

The truth of the matter is that no matter how much time you have, it is not enough time to test all functions and combinations of functions in most software. In essence, all testing is sampling, so it becomes very important where we draw the boundaries in testing and where we take our samples.

In this article, I explore ways to deal with the time constraints in software testing and still have a reasonable level of test coverage and overall confidence in the quality of the application being delivered.

The Impact of Time Constraints on Testing

The lack of time for testing is a symptom of deeper project, process and organizational issues. However, that doesn't eliminate the impact of the time crunch. The impact is seen in the following areas.

1. Lack of Test Coverage

Test coverage can be seen in a variety of ways: code coverage, requirements coverage, and test case coverage, to name a few of them. If you can't cover enough of these items, it's hard to give a reasonable level of confidence in the software.

2. Lack of Overall Software Quality

If you can't test to the levels needed to assess the quality of the application, the test will be incomplete and defects will be missed. Overall software quality suffers.

The reality is that just because testing stops, software problems (i.e., defects) will still continue. It's just that the users will eventually find them. Sometimes the application is mission critical or safety critical, so to miss finding defects due to lack of test time can be a very high risk.

However, even "complete" testing does not necessarily equate to high quality applications. This seems to be contradictory. After all, why do we perform testing if we can't make the assurance of quality?

What we may deem as "complete" may just be an illusion. There is always one more thing (or millions of things) you can test.

© 2016, Rice Consulting Services, Inc.

3. Poor Morale

It really does kill morale to ask people to do something as challenging as testing, then not give enough time, tools or people to get the job done. Even worse is when the testers get blamed for defects when the time has been cut short, or the amount of work is impossible to complete even in a thousand years.

Test teams that are starved of resources often don't see the possibility of applying what would normally be considered sound testing techniques. The comments are along the lines of, "Oh, we would never have time to document a test plan. We barely have enough time to test as it is."

Never mind the idea that a test plan could actually save time by making sure the resources are used in the most efficient ways, or that a workable scope balanced by risk could be defined – all of which save time.

4. Missed Expectations

Sometimes people have the expectation that exhaustive testing is possible with scarce resources and that testers would just waste time if they had more of it. Or, management believes sometimes just the opposite, "Since we can't test everything, why bother at all?" This is primarily a perception issue.

Many times, project deadlines are established with the expectation that an arbitrary allocation of time will be enough time for testing, even though there is no clarity about which kinds of tests are needed, or even what the scope of testing will be. It is only after the design of the system becomes clear that the scope of the project becomes clear.

So when defects are discovered, the project schedules often slip and so does the management attitude toward the testing effort. These projects are either delivered later than planned, or delivered on-time with high levels of defects.

An Exercise in Priorities

Realizing that you could have all the time you desired and still not have enough time to completely test anything of even moderate complexity, the issue becomes how to define the scope of testing.

Key ways that testing fails are to 1) do the wrong tests, 2) do the right tests in the wrong ways and 3) fail to do enough of the right tests. So, in the third way mentioned, lack of time is a root cause. In the first two ways, it is possible to leverage the time available for testing with efficient test design and test execution, combined with reviews and inspections.

If we recognize the fact we can't test everything, the challenge then becomes to pick the right things to test. Picking what to test is like packing a suitcase. You only have some much room and so much weight allowance to work with. First, you pack the big and important things, like the main clothes you need. Next, you pack the smaller important items, like toiletries. Finally, if you have room and weight, you can pack your fuzzy slippers.

A key question is "How do we prioritize?" Some might say by risk. Others might say "by what we getting paid or pressed the most to deliver." It is important to note that criticality is not the same as risk. Risk is a potential loss while criticality is importance. Sometimes risk and criticality has a direct relationship, such as a function that is safety-critical. However, in other cases, a function may be critical to certain users yet pose no potential negative impact.

© 2016, Rice Consulting Services, Inc.

Yet there is more to efficiency than designing and prioritizing the right tests. You also have to design and perform them efficiently.

Going back to the suitcase analogy, how you pack the suitcase makes a big difference in how much you can pack in it. Some people are masters at packing a suitcase. They know how to roll their clothes, how to fill small spaces and so forth.

Likewise, testers need an efficient process for testing. One example of this is using combinatorial techniques to get the most coverage from the fewest number of test cases. The National Institute of Standards and Technology (NIST) has a wonderful free tool called ACTS (Advanced Combinatorial Testing System) which generates test cases based on combinatorial test design.[1] The advantage is that you get high levels of test coverage with a minimal number of tests (Figure 1).

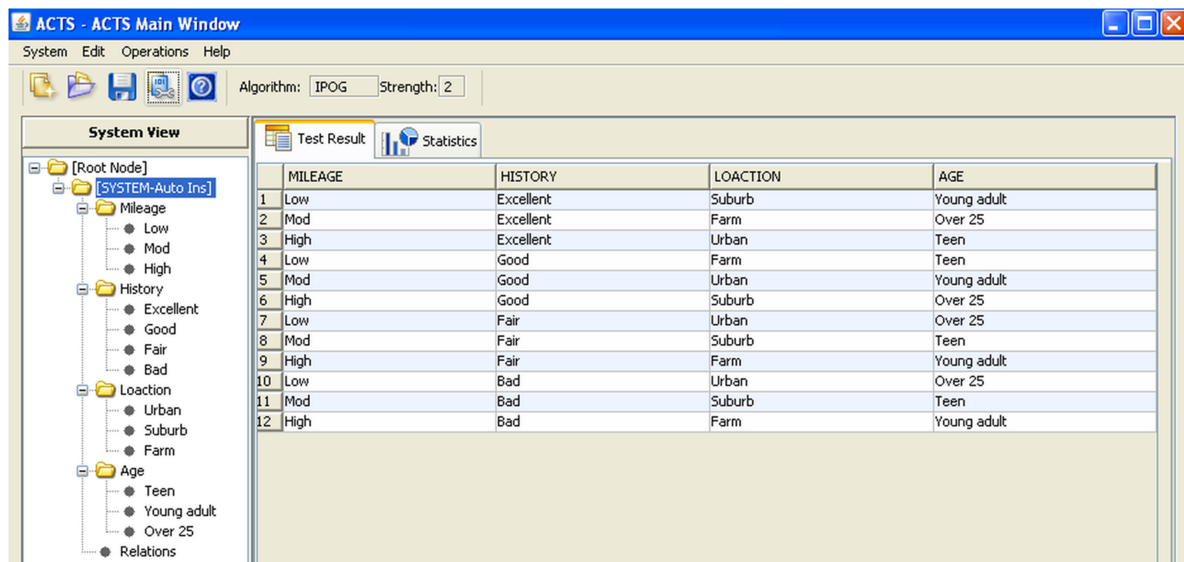

Figure 1 - ACTS Combinatorial Test Design Tool

In this example of determining automobile insurance rates based on risk factors, there are seventy-two possible condition combinations. After applying the pairwise algorithm, we are left with twelve cases. While this is a great reduction in the total number of test cases, care must be taken to review the tests and supplement them with tests that may be important, but not generated by the tool. You will also need to manually define the expected results, which requires time.

What is Pairwise Testing and Why is it Helpful?

There have been over twenty research projects[2] that have sought to learn why defects are missed in testing but found by users and customers in actual use. In all of these studies, the conclusions are the same – that the vast majority of missed defects in testing are due to the failure to test all pairs of conditions.

Perhaps you have experienced times when you tested multiple functions in an isolated way and observed that Function A worked, Function B worked, and Function C worked. But then, perhaps after the release of the software, it was discovered that Functions A and B failed when performed in combination. Those are the types of failures we are trying to find and fix before release.

Going back to the 1930's, manufacturers learned that mathematical methods such as orthogonal arrays and Latin squares could be applied as a technique to organize tests (or "design of

© 2016, Rice Consulting Services, Inc.

experiments) to achieve the most efficient way to test all pairs of conditions. In the 2000's this idea caught on in the software testing community as a way to test related conditions as well. However, using orthogonal arrays and Latin squares is the long way to achieve this kind of test design. A much faster and scalable way to design pairwise tests is through the use of pairwise algorithms. The great advantage is that algorithms can be implemented in tools, such as the ACTS tool described above.

The great news is that we don't have to test all combinations, since we have already seen that is an impossible effort in many cases. Our first concern is to test all pairs of conditions, then add other conditions as needed. And…that kind of test design can be performed by tools!

What About Test Estimation?

Test estimation is inherently tricky because it is an approximation. Some people are very challenged in providing estimates of the testing effort due to lack of knowledge about what is to be eventually tested. In some cases, the problem of test estimation boils down to poor estimating methods. By "poor estimating methods" I mean that an organization is not handling test estimates in a way that deals with the facts and risks as they really are. Instead, they become overly optimistic or just pull a number out of thin air for the testing effort.

In those cases where the product is already developed, such as Commercial-off-the-shelf (COTS) applications, test estimates can still be difficult if you don't have access to the software yet. It is also a surprise to management sometimes that the time needed to test a COTS product may actually be a multiple of the time and cost needed to acquire and install the product.

In estimation, accuracy is not so much the issue as is getting the recipient of the estimate to accept your estimate, which you honestly believe as the true effort and time needed for testing.

Let's say you work hard to create an accurate test estimate for a project. You estimate the test to take five people three weeks to complete. Your manager may assume there is some padding in your estimate, so she "readjusts" your estimate to be two weeks for testing with the same five people. In this example, the accuracy of the original estimate really doesn't matter. When it's all said and done, you and your team are still expected to do three weeks of testing in two weeks time!

However, the next time you have to provide an estimate, you will remember that estimate reduction and will likely inflate your initial estimate because you know it will be cut. This is how the dysfunction around estimation is propagated.

That is the dark side of test estimation. There is a better way but it requires a healthy, eyes-open attitude on everyone's part to re-adjust the estimate when necessary or make adjustments in scope and resources.

Another point of awareness and agreement is that an estimate is not completely accurate – it is a best approximation based on experience and other factors.

The problem with highly precise estimates is that they often assume things will go well and without delays. The reality is seldom that trouble-free.

To deal with the unexpected occurrences that cause actual test times and resources to differ from their estimates, contingencies are needed. A helpful way to think of contingencies is in the form of reserves.

© 2016, Rice Consulting Services, Inc.

Reserves are simply more people, tools, and other things available in case of unforeseen events. Contingencies are your "Plan B" which describe your options should a certain risk materialize.

Some people might think of reserves as "padding" an estimate, but my experience is that reserves and contingencies are needed as a safety net for when schedule crunch occurs. It is very important not to squander reserves in time-wasting activities. For example, if your team spends forty hours on work that could easily be done in ten hours, that is an inefficient use of time and is not the purpose of reserves.

Dealing With the Time Crunch

Here are a few ways to deal with short test timeframes.

1. Get Good at Prioritizing

Unfortunately, most of the discussion around prioritizing testing is all about risk. While risk is a key concern, it is not the only way to prioritize testing. Risk-based testing is a sound practice in many cases, but there are risks, even to risk-based testing. One such risk is that our risk assessment may be totally wrong due to factors we cannot foresee.

In addition, risk-based testing becomes ineffective when everything is a high risk. You will need another way to prioritize tests in that situation. To prioritize means to determine what is important. But the big question is "Important to whom?"

Priorities for testing are often a reflection of the stakeholders, such as project management, senior organizational management, customers and users. One big trap we fall into is thinking that the priorities given to development and testing groups are the real priorities.

Testers are not in charge of the priorities of a project, and therefore, not really in charge of the priorities of testing. Even though a tester may write the test plan and set the test schedule, they may not reflect the true project and stakeholder priorities. Like the traveller who packs someone else's bag (don't really do that – you can get in trouble for it!), you may include the important things to you, but not to the other person. It is all about who owns the bag.

You can prioritize based on:

- Mission need – What solves a key problem or delivers needed functionality right away?
- Stakeholder needs – Which products and features do users and other stakeholders value most?
- Risk – Where might problems most likely appear, and where might they carry the most impact?
- Management directive – Ultimately, management sets project priorities. However, on occasion, someone in management decides his or her pet project or feature is most important regardless of business value or risk. Therefore, it becomes a high priority in testing.
- Experience – Where have we seen problems in the past that we don't want to repeat?
- Sampling – There are many ways to take samples in testing, such as random samples, customer samples, and so forth. The idea is that if you find a concentration of problems in one area, there will likely be others in that same area. This is known as "defect clustering."

2. Management Needs to Understand The Tradeoffs

The management influence on schedules and priorities drives much of the testing time challenge. Many organizations are schedule-driven in which the software delivery schedule takes precedence over other decisions.

© 2016, Rice Consulting Services, Inc.

There are four key factors in the time challenge:

- Resource levels
- Schedules
- Workload balance
- Process efficiency

In many cases, deadlines are necessary and important. Management, however, must allocate the right number of resources to get the job done whether it is in development, testing or anything else. At the time of this writing, we have been through an economy where the theme in organizations has been to learn how to do more with fewer resources. That includes people, tools and hardware.

Gerald Weinberg writes about “The Law of Raspberry Jam”. This law says, “The wider you spread something the thinner it gets.”[3] Just like spreading jam on toast, you can have such a small amount that after spreading it over a piece of toast, you can hardly taste it at all.

Senior management needs to understand and embrace this concept when defining their expectations of what can be accomplished with given resources. There comes a point when project resources can be spread so thin that you can’t even tell any difference was made at all.

3. Optimize the Testing Process

Many of the perceived time problems in testing may be due to poor workload balance and inefficient testing approaches. It is management’s job to fix inefficient or ineffective processes because people in the trenches don’t have the authority or control to make major process changes.

People at the grassroots level may have the opportunity to work smartly, but sometimes that is thwarted by management’s decisions to do things that are inefficient and ineffective.

Here are three ways to optimize the testing process. By doing these things, you perform testing more efficiently, which takes less time and often finds more defects as compared to non-optimized test methods.

Optimization Technique #1 - Apply Sampling Techniques

Going back to the idea stated earlier in this article that all testing is sampling, it can be very helpful to learn how to sample for defects.

Testers face a “needle in the haystack” problem. We are looking for something very small in something very large. I am intrigued at how some gold miners deal with a similar challenge, and how those techniques can be applied to software testing and finding defects.

Like testing, gold mining is also a very expensive undertaking. Even a small operation can cost thousands of dollars per day to move layer upon layer of dirt to get close to finding gold. Unfortunately, there are no signs that read, “Dig here for gold.”

One interesting and effective technique to know where to dig for gold is to take soil samples by drilling. That is the best way to see a) if there is any gold in the immediate soil and b) how dense the gold concentration may be, c) how deep the gold is, and d) the quality of the gold. If the soil sample shows gold twenty feet or more below the surface, it will probably cost too much to remove the dirt above it.

© 2016, Rice Consulting Services, Inc.

Like in sampling for gold, in testing, we can sample in various areas of a system with functional testing, we can sample lines of code with structural testing, we can perform random tests either manually or with tools, or we can sample based on risk, just to name a few possibilities.

Sampling can help us find those pockets or clusters of defects. This can lead us to the best places to focus testing for the little time we have. Finding high concentrations of defects in software is like striking gold in that you get a very high return on investment for testing.

An effective and traditional form of sampling is to use equivalence classes (or partitions).

"In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way. Equivalence partitions (or classes) can be found for both valid data, i.e., values that should be accepted and invalid data, i.e., values that should be rejected. Partitions can also be identified for outputs, internal values, time-related values ((e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing). Tests can be designed to cover all valid and invalid partitions. Equivalence partitioning is applicable at all levels of testing.

Equivalence partitioning can be used to achieve input and output coverage goals. It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing."[4]

The equivalence partitioning technique has been presented in many of the classic books on software testing, such as *The Art of Software Testing* by Glenford Meyers (First Edition) and *A Practitioner's Guide to Software Test Design* by Lee Copeland.

In addition, you can view a video on equivalence partitioning at https://youtu.be/GEM-LKTggm4.

Optimization Technique #2 - Apply the Pareto Principle

The Pareto Principle (the eighty/twenty rule) that says you can get the majority of value from the minority of people, time or effort. In fact, my experience is that it is possible to get over ninety percent of the value of some efforts in a very short time frame.

As an example, once I was asked to perform an eight-week test in two weeks. So, the client and I worked together to focus on key areas and skip minor areas. The test design was based on critical user workflows and modular tests that yielded many more defects than expected.

While we realized that some features would not be tested, we made the best use of the limited time. We also made sure senior management understood the risks of a scaled-down test.

Optimization Technique #3 – Use Combinatorial Methods

As mentioned earlier, tools such as ACTS and model-based tools can help greatly by intelligently combining test conditions into the most efficient set of tests. If you choose not to use a combinatorial tool, even a basic decision table can help reduce the number of tests without sacrificing coverage.

4. Agile Methods Can Help

I take great care with how I portray agile methods to avoid promoting agile techniques as the solution for all software project problems.

When applied pragmatically, agile methods can be a great help in getting the right things done quickly. When applied haphazardly or in the wrong cultures, agile can lead to disaster faster.

© 2016, Rice Consulting Services, Inc.

The good aspect of agile is that people learn to minimize the meetings, the documentation and the project overhead to focus on deliverables. The challenging part of agile is to keep the knowledge we really need because of the lesser emphasis on documentation. Agility is all about dealing with change gracefully and focusing on the right things - do that, and you'll be fine.

5. The Right Tools Can Help

Tools can save time when used in the right ways. Test design tools such as ACTS (Figure 1) can save huge amounts of time as compared to designing tests manually. The generated tests can then be imported into test execution tools to automate the mundane tests and help in regression testing – given that you don't have to spend large amounts of time on tool implementation and maintenance issues.

The time saved by using tools is often the result of an existing test tool implementation effort that is well-designed and has some history behind it. Initial tool efforts typically take more time because you are learning as you implement. In other words, thinking that getting and using a tool will save time immediately is likely not going to work out as expected. That is because it takes time to learn the tool, but more importantly, it takes time and understanding to know which tests to implement and how to implement them.

6. Know When You Are Over-testing

Whether it is your status report, test plan, or anything else, there comes a time when the finish point needs to be declared and then move on. It's like the old saying that "projects are ninety-nine percent done forever."

Seth Godin says, "Ship often. Ship lousy stuff, but ship. Ship constantly."[5] I don't concur with *all* of that (like the shipping of lousy stuff), but I do get the sentiment. The main point is to overcome the resistance of things that keep us from making progress and to deliver something of value.

How about this instead? **Ship often to the *right* people with the quality they need.** Some people are happy to get new features, warts and all. They will give you great feedback and will still be raving fans.

Many times, perception is reality. If a software project is delivered with obvious defects, users don't care who is to blame. They experience frustration and want the problems fixed "now." It's the old maxim that says, "There's never enough time to do the job right, but there's always time to do it again."

As an example, an update to Apple's iOS8 mobile operating system caused major problems for users in that they were unable to make phone calls with an iPhone after the update was installed. Apple recalled the update very quickly, but not without reputation damage. For the first time I can recall, a major news outlet identified the QA Manager of a defective product by name.[6]

It's not clear why the iOS8.0.1 defects were not found – whether it was a lack of test time or some other root cause. The result, however, shows what happens when defects go "big time." Blaming the QA Manager and testers is not the solution and is not helpful, since testers usually don't make the decision to release software.

© 2016, Rice Consulting Services, Inc.

Plan of Action

These are not sequential steps, but rather ways to implement some of the ideas in this article.

1. Lay a foundation of expectations at the management level that exhaustive testing is impossible and that the thinner testing resources are spread, the less effective they become. If your estimates differ greatly from those already set for you, be able to justify why you need more time and resources for testing. Risk is a good way to balance this discussion.
2. Get good at identifying where the risk is in the things you test.
3. Learn how to sample products to find where defects may be clustering.
4. Define a workable scope of testing. If you set the scope of testing too large, you won't finish in time. If you set it too small, you won't get the confidence levels you need for deployment.
5. Optimize your tests.
6. Work with project managers to build reasonable reserves and contingency plans.
7. Create a risk-based test planning and reporting process.

Summary

Like any project activity, testing takes a certain amount of time and resources to perform if it is to measure an accurate level of confidence and quality. Exhaustive testing is impossible due to the very high numbers of possible tests required to test all combinations of conditions. By combining dynamic testing with reviews and inspections throughout a project, the effect is often a time and cost savings.

However, it makes a big difference in the time crunch as to how testing is prioritized, designed and performed, as to whether time and resource constraints are dealt with in an effective way.

Good estimates are helpful, but even the best efforts at estimation may fall short of being accurate. For those times, you need some reserves in time and resources as a contingency. You also need a "Plan B" for those times when the time crunch gets really bad.

There are no magic solutions to the time crunch challenge. However, with some expectation management and careful test selection and optimization, you may just find enough time to perform the level of testing that gives an acceptable level of confidence to stakeholders.

About the Author

Randall W. Rice, CTAL (Full), CTFL-AT, CMT

Randall Rice is a leading consultant, author, and speaker in software testing and software quality. He is a popular speaker at international software testing conferences and is co-author with William E. Perry of the book, *Surviving the Top Ten Challenges of Software Testing* and *Testing Dirty Systems*. Randall also serves on the board of directors of the American Software Testing Qualifications Board (ASTQB).

Rice Consulting Services, Inc.
1608 SW 113th Pl
Oklahoma City, OK 73170
Phone: 405-691-8075
E-mail: rrice@riceconsulting.com

© 2016, Rice Consulting Services, Inc.

References

1. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html
2. http://www.pairwise.org
3. Weinberg, Gerald, “The Secrets of Consulting”, New York: Dorset House Publishing, 1985
4. ISTQB Foundation Level Syllabus (V2011), http://www.istqb.org
5. http://99u.com/tips/6249/Seth-Godin-The-Truth-About-Shipping
6. http://www.bloomberg.com/news/2014-09-25/apple-s-iphone-software-snafu-has-links-to-flawed-maps.htm

Resources

Copeland, Lee, A Practitioner’s Guide to Software Test Design, Artech House Publishers, 2004
Jones, Capers and Bonsignour, Olivier, The Economics of Software Quality, New York: Addison-Wesley, 2012
Myers, Glenford, The Art of Software Testing, 1979
Perry, William E. and Rice, Randall W, Surviving the Top Ten Challenges of Software Testing, New York: Dorset House Publishing, 1997
Weinberg, Gerald, Perfect Software, New York: Dorset House Publishing, 2008

© 2016, Rice Consulting Services, Inc.