Rapid Testing for Rapid Maintenance

Version 1.2, 4/25/19, by James Bach (first published in TASSQuarterly Magazine, 9/06)[1]

A correspondent writes:

"I have a test management problem. We have a maintenance project. It contains about 20 different applications. Three of them are bigger in terms of features and also the specs that are available. I am told that these applications had more than 1-2 testers on each of these applications. But in this maintenance project we are only 6-7 testers who are responsible to do the required testing. There will be a maintenance release every month and what it will deliver is a few bug fixes and a few CRs. What those bugs and CRs would be is not known in advance. Could you please suggest how to go about managing such kind of assignment?"

Okay, this is what I would call a classic rapid testing situation: lots of complexity, not a lot of time or people.

In tackling this problem, first I would analyze the context. Many images leap to mind when I hear words like "application" and "maintenance" and "CR" (that means change request, right?), but these images may be mistaken. I typically refer to a diagram called the Context Model (https://www.satisfice.com/download/rapid-software-testing-context-model) to help me think this through. In this case, the factors foremost in my mind are the following:

Consider your mission. What specifically do your clients expect from you? Do they need you to find important bugs quickly, or are there also other requirements such as the use of certain practices and tools, or the production of certain documentation? *(This is important for two reasons: it may be that there is no way for you to achieve the mission in your situation, in which case you'll need to renegotiate it; also, clarity of mission helps you avoid doing anything that isn't necessary to do. In your situation, you don't have the luxury of putting your process on the cruise control of unexamined assumptions.)*

Consider the quality goal. How important is the product? What if you miss a bug? Are the customers of this product very sensitive to problems, or are they tough computer geeks who don't mind a little crash once in a while? Do these products have the capacity to harm anyone if they fail? Apart from the general "level of quality" you are testing for, have you considered the different *types* of quality (reliability, usability, compatibility, performance, etc.)? *(This is important because higher risk justifies and demands a more expensive and meticulous testing approach.)*

Consider your team. Are your testers interested and able to do exploratory testing under pressure? Do they understand the products already or do they need to climb the learning curve rapidly? Also, are there any people not on the testing team who might be

[1] Thank you Mike Kelly and Cem Kaner for indispensible comments!

able to help test, such as tech support or documentation people? Even friendly users in the field might be able to help. *(This is important because I suspect that this situation will call for some creative and intensive testing by skilled testers. Testers who prefer to be spoon-fed rote test cases will probably be miserable. Furthermore, the situation is not necessarily easier if you are merely maintaining existing tests and test infrastructure that someone else has handed you— because you need to master those materials, and you may need to throw them out as obsolete or inadequate.)*

Consider your test lab and materials. Do you have equipment, tests, test data, automation, or anything else that will help you retest the products over time? For instance, do you have a test coverage outline, which is much more flexible than a set of test cases? *(This is important because if you have existing tests or test documentation, then you have to figure out if it's actually helpful. Possibly the old tests weren't very good. Maybe the existing automation is broken and not worth fixing. Take stock of your testing infrastructure.)*

Consider the overall testability of the products. Are these applications conceptually easy to test or difficult to test? (see my Heuristics of Testability document for more ideas on this, at https://www.satisfice.com/download/heuristics-of-software-testability) Can the programmers make them easier to test by adding logging, or scriptable interfaces? How much is there to test? Are the products modular, such that a problem in one part won't necessarily affect other parts? *(This is important because testability is critical to the consistent success of meagerly staffed test project.)*

Consider systemic obstacles to rapid testing. Is there anything about the applications or your project that poses a predictable obstacle to testing? For instance, if they are mainframe-based batch processing systems, there may be a fixed and irreducible delay between starting a test and finishing it. If your programmers are working 10,000 miles away, or if the build process takes 48 hours, then the turnaround time to deal with problems is not going to be good. *(This is important because you need to set expectations and negotiate agreements to mitigate the effects of these obstacles.)*

Consider stability. Are these applications brittle or supple? Is it likely that the maintenance process will create more problems than it fixes? Are there robust unit tests, for instance? Are components used that don't change and that you trust so much they can be considered as "already tested"? *(This is important because when programmers are working with a stable code base and when they do reasonable unit tests, the probability of a bad bug reaching you is lower, and it will require less effort for you to reach a level of comfort about the status of the product during your testing.)*

Consider the cohesiveness of the product line. Are these twenty applications completely separate, or do they interact and share data or components? Is there a flow of data among the individual applications? *(This is important because a cohesive, highly integrated set of applications can be tested together and against each other. The output of one may validate the output of another. The flipside of that is the increased chance of a fix in one causing a problem in the other.)*

Consider the availability of good oracles. How easy is it to validate that the outputs of a product are correct, or at least, sane? Will a bad problem also be an obvious problem, or is it possible that bad problems can occur in testing, but not be noticed by the tester? *(This is important because you may need to invest in tools, training, or reference materials to assure that you will spot a problem that does, in fact, occur.)*

Consider the development process. How and when will decisions be made about what to fix and what to change? When will you get the new builds? Can you get detailed information about specifically what has changed? Will testing have a voice in this? Are you in good communication and reputation with the programmers? Can you review the testing done by developers to see whether you can skip or minimize that sort of testing in your own strategy? *(This is important because considering yourself as part of a combined quality creating and witnessing effort helps you optimize your work. Also, you can test the product better when fixes are made with testing requirements in mind. You won't be blindsided if you are involved in those discussions.)*

These considerations are part of understanding the general lay of the testing situation. As a result of thinking these things through, I'm sure I would have in mind some issues to raise with management and the programmers. In any case, the following is my first impression based on your question. You can consider this my default mental template for dealing with this kind of test project, subject to amendment based on what I discover from learning about the issues, above:

- The solution is probably not heavily documented manual test procedures. Those are expensive to produce, expensive to maintain, and do little to help test a complex product. They are favored by managers who don't understand testing and large consulting companies who get rich by exploiting the ignorance of said managers. See almost any military software test plan for an illustration of this sad principle. See any test plan afflicted with *Sarbanes-Oxley* syndrome.
- If your products have a rich user interface, and if that interface is subject to change, then automation is probably not going to be a big part of your solution. It is a sisyphusian task of scripts breaking and being repaired again, and the automated checks you create would be pretty simplistic. However, if I did have a programmer available who enjoyed automation, I would look for low cost, simple, quick uses of tools to help the testing. That could mean creating a tool to create test data, or to do some relatively simple user simulation. If the system has an API you are in much better shape, and I would make use of that. See my paper on this for details (https://www.satisfice.com/download/a-context-driven-approach-to-automation-in-testing
- The solution probably does involve vigorous and targeted interactive testing that is especially exploratory. This means that you do not script tests in detail, in advance of performing them. Certainly you prepare to test the products, but your preparation is primarily in the form of preparing test data, test platforms, test tools, concise test strategy and test outline documents, and learning what you can about each product.

- Locate a calendar. Lay out the milestone dates. What do those dates mean? Are they release to manufacturing or final customer ship dates? Work backwards from those dates. What is the last moment that you can receive an acceptable build and still render a decent test report so that your clients can decide to release it on time? A test cycle is the set of activities you perform from the time you get a build to the time you feel you have fulfilled your testing mission with that build. Now, consider thinking in terms of four kinds of test cycle: full, incremental, spot, and emergency. A full test cycle is what you do when the product comes to you after being destabilized with disruptive changes. You need to estimate how long that takes for each application. An incremental test cycle is what you do when the product comes to you with minor or carefully targeted changes. That should take much less time. A spot cycle is what you do when a developer wants your team (or someone on your team) to take a quick informal look at something he's done, prior to "official" testing of a "formal" build. An emergency cycle is what you do when a fix must be made at the last minute, or when there is otherwise very little time to react. Look at the calendar and try to get an idea for how many test cycles you will be facing per release. When will they probably be? What delays will accompany them? Are changes going to dribble in, or will most of them be made all at once?
- As for your test cycle strategy, you need to develop the equivalent of a two-minute drill for each product; a well-organized test cycle. When a fix or fixes come down the pipe, you should have someone testing that specific change within a few minutes, and you should have preliminary test results within a few minutes to an hour after that. You achieve this by the approach of testing by skilled testers. Assign the products to the testers such that each tester has a few products that they are required to master. Everyone should be up to speed on each product, but for each product there should be a responsible tester who coordinates all testing effort. A responsible tester must be reasonably skilled, so depending on your team, you may be the only responsible tester who supervises the other supporting testers (who may in turn be part-timers or developers pitching in to help).
- …or maybe fixed obstacles to rapidity make a "two-minute drill" impossible. In that case, you still need to find ways to minimize potential delays, perhaps by acquiring the equipment to make long test runs parallel instead of serial, or by piling more value into each test activity through improved preparation.
- If your products are not easy to install, you need to work on that problem. You can't afford to waste hours after the build just trying to get the products up and running. Remember: two-minute drill. Hup hup. When a product is released to your team, they should start exploring it like snakes on plane.
- Make a set of product coverage outlines that address all your products. A product coverage outline is literally an outline of the structures, functions, and data that you might need to cover when testing those products. Start with something no more than, say, two pages long per application. Use Notepad, Excel, or some other very low formatting method of documenting it. Exotic formatting just slows you down. Use these outlines to plan and report your testing on the fly. (Use the Heuristic Test Strategy Model, available at

https://www.satisfice.com/download/heuristic-test-strategy-model, to help produce these outlines. An example outline for a game called “Putt Putt Saves the Zoo” can be found in the appendices to my RSTE class, at https://www.satisfice.com/download/rst-appendices.)

- Make a list of risks for each application. By risks, I mean, what kind of bugs would be most worrisome if they were to creep into those products? For instance, for Microsoft Excel, I would think "math error" would be high on the list. I would need to have tests for mathematical correctness. (The Heuristic Test Strategy Model will help here, too, and you will find two specific articles about identifying risks in software in the appendices to my RSTE class.)
- Make a one or two-page outline of your test strategy. This is a list of the kind of test activities you think need to be done to address the major risk areas. Be as specific as you can, but also be brief. Brevity is important because you need to go over this strategy with your clients, and if it's just a page or two long, it will be easy to get a quick review. (See examples of test documentation and notes in the appendices of my RSTE class.)
- In producing your test strategy, look for test tools, techniques, or activities that might be able to test multiple products in the same activity. If your products are part of an integrated suite, this will be much easier. I would focus on scenario testing as a principle test development technique and given that this is a maintenance project, I suspect that parallel testing (whereby you test a product against its earlier version in parallel) will also be a bread-and-butter part of the strategy.
- Seek variety and freshness in your testing. A risk with maintenance testing, especially under pressure, is that you lapse into following the same paths in the same way with the same data, cycle after cycle. This makes testing stale and minimizes the chance that you will find important bugs. Remember, you can’t test everything, so that’s why we need to take fresh samples of product behavior each time, if that’s feasible. Yes, you need to cover the same aspects of the product, again and again, but try to cover them with different tests. Use different data and do things in a different order. It also helps to use paired exploratory testing (two people, one computer, testing the same thing at the same time). Pairing injects a surprising amount of energy into the process…
- … But although variety is important, it may be important for some tests to be repeated each test cycle in pretty much the same way. This can be useful if it’s difficult to tell whether a complicated set of output is correct. By keeping some tests very controlled and repetitive, you can directly compare complicated output from version to version, and spot bugs in the form of subtle inconsistencies in the output. Hence, you may need to prepare some baseline test data and test lab infrastructure and preserve it over time.
- Brief your management and developers as follows: *"I want to serve you by alerting you to every important problem in these products before those problems get into the field. I want to do this without slowing you down. If I'm going to do a good job with such a small staff, then I need to know about fixes and changes as*

early as possible. I need to be involved in these decisions so that I can let you know about possible difficulties in retesting. If you are careful about the way to change the product, and if you share with me details about the nature of each fix, my team can test in a much more targeted, confident fashion. In general, the more testable this product is, the more quickly I can give you a good report on the status of each change, so keep that in the back of your mind as you go. Meanwhile, I commit to giving you rapid feedback. I will do my best to keep testing out of the bottleneck."

- You must develop skilled testers in your team if you don't already have them. I'm talking about tactically skilled testers: people who are comfortable with complexity, understand what test coverage is and what oracles are; people who can design tests to evaluate the presence of risk; people who can make observations and reason about them. To support this, I have videos on YouTube. I have lots of written training materials on my site, too. See also the various articles on my blog, or for that matter, see my book *Lessons Learned in Software Testing*.
- If there isn't already a bug triage process in your group, establish one. I suggest that you run those meetings, unless the program manager insists. You need to get good at making the case for bug fixes. I have prepared a cheat sheet to help test managers with this, you can find it as an appendix to my process evolution article at https://www.satisfice.com/download/process-evolution-in-a-mad-world, but it's also included in the RSTE class appendices.
- Establish a testing dashboard either online or, as I prefer, on a whiteboard. It should be something that expresses, at a glance, your testing status, and does so in a way to answers the most important management questions. See my presentation on a Low Tech Testing Dashboard on my site at https://www.satisfice.com/download/low-tech-testing-dashboard

As you work through these ideas and activities— and you will revisit them frequently throughout your project— I recommend that you keep an issues list. The issues list is your repository of concerns and obstacles that you must resolve with your clients. Your clients want a service from you, and you want to provide that service. That means if anything threatens your ability to provide that service, you must raise those issues in a timely manner and on a regular basis. The resolution of an issue may be that your client accepts more risk; or perhaps they will provide you with more testers. These issues are arising more or less every day, however, and your job as a test manager will be to know what they are, explain them in specific, compelling language, and shepherd them to resolution.

Yes, it's a lot to think about, being a leader of a rapid test project. That's probably why they put you in charge of it… or maybe because they have no idea what they were doing. Either way, how you meet this challenge will affect how you gain credibility in your project. The more credibility you have, the more you can influence your situation for the better.