Dealing with a Test Automation Bottleneck

https://www.stickyminds.com/article/dealing-test-automation-bottleneck

By **Michael Stahl** - July 1, 2019
Share URL

Summary:
The test team uses the test automation system to execute thousands of test cases because … why not? The tests are running automatically, for free, so there is no incentive to improve test efficiency. Just run them all! But eventually, as more and more tests are added, the system becomes overloaded. Test runs are delayed and you get a bottleneck. Don't throw more money—or new systems—at the problem; do this instead.

It's usually difficult to get management approval for recruiting more people. It is much easier, on the other hand, to get approval to spend money, especially when you work at a successful company with a healthy budget. Not to say that you use money as wall-to-wall carpeting, but if you can justify the expense by a positive ROI analysis, in most companies there is a good chance the request will be approved.

However, this can actually have some negative side effects, at least in the testing world.

Let's imagine that we built a wonderful test automation system. The system receives a list of tests to run and executes them in parallel on a large number of computers. The system is so good that it encourages testers to automate more tests, as doing so clears up the testers' time to do some real engineering work, like think of new and interesting test cases, develop tools that allow deeper testing, and participate actively in reviews. Besides, it's much more fun to write new code or develop a script than to execute the same test for the seventeenth time.

The test team uses the automated system to execute thousands of test cases—even on the daily test cycle—because … why not? The tests are running automatically, for free, so there is no reason to take the risk of missing a possible regression. Just run them all! It even improves the ROI of the automated system.

The system is indeed lightning-fast, but eventually, as more and more tests are added, it becomes overloaded. Test runs are delayed and the test automation system becomes a bottleneck. At a certain point, the program managers fail to get test results on time, and they start raising flags—and hell.

Everyone realizes we must go over all the test cases and define priorities: Which test case must be executed each day? What's enough to run once a week? Which tests are inefficient and could be made to run significantly faster?

But with thousands of test cases in the system, such prioritization or optimization of effort is a significant investment, and no one has the time to do it. Everyone is swamped with executing tests that are still manual, investigating failures, or testing possible bug fixes. If only we had some time to work on this! If only we could recruit a few junior engineers to do execution and free the senior engineers to improve the existing tests!

Recruiting new people is out of the question. But there is a solution that, in one fell swoop, will reduce the test cycle time by 50 percent and solve the problem: Double the number of computers controlled by the automation system. This is practical and it's easy to prove the ROI, if only by showing that version release time will be shortened by half the test cycle

time. And besides, it's easier to get approval to spend money than to get an open req. It also takes much less time to achieve results: A new person must learn the product, understand the technology, and find out how to operate the coffee machine. A new computer only needs a power line and network connection to start executing.

So we buy more computers and connect them to the system, and indeed, the bottleneck miraculously disappears … only to reappear a few months later, when more tests are added to the system.

One of the reasons for this phenomenon is the **tragedy of the commons**. The principle, according to Wikipedia, describes "a situation in a shared-resource system where individual users acting independently according to their own self-interest behave contrary to the common good of all users by depleting or spoiling that resource through their collective action." In an organization-wide test automation system where each team can submit unlimited test cases for execution, the principle explains why no team has an incentive to become more efficient.

Let me illustrate with an example.

A certain project has five test teams that share a common test automation system. The system, as expected, is overloaded and delays test cycle schedules. Each team understands that it should invest effort in reducing test time, but something is blocking their taking action.

Assume that if each team were given sole use of the system, running each team's test would take twelve hours apiece. Since all the teams are sharing the test system, the overall duration of a test cycle would be sixty hours. Team A decided to tackle the overload problem by reducing their test cycle time by 50 percent, to six hours. They improved the test time of all tests, removed redundant tests and merged others to save setup time, and did needed modifications to make their tests robust and save time lost for false failures. Eventually, they met their goal.

The overall test cycle now takes fifty-four hours. So, for the superhuman effort done by team A, the overall reduction in test cycle time was 10 percent. Since the other teams continue business as usual, the time saved by team A will quickly be consumed by new, inefficient tests added by the other teams.

This reality means that no team has an incentive to invest in improving their test efficiency.

Another problem that excessive testing brings is an increase in the number of false failures. Any time we run a test cycle, some tests will fail—not due to a product problem, but to some problem in the test itself, the test environment, or the automation system. It could be a hard drive crash, an overheated motherboard, or a flaky connection. The number of such cases depends on the skill level of the testers developing the automated tests and on the robustness of the test framework, but there always is some level of incorrect failure "noise."

In many cases, a rerun of the test passes. And while the percent of bogus failures may stay constant, the more tests we load to the system, the more false fails will occur, with the associated cost of engineering investigation and correction time. This further reduces the availability of resources to work on test efficiency improvements.

I have witnessed this process taking place not only in testing, but also in DevOps. Each check-in of code triggers a set of continuous integration (CI) tests. CI runs on a central, automated system, and the tendency of test time to get longer happens there as well.

It happens naturally because, as more functionality is added to the product, more tests must be added to the CI test suite. It can also happen when a new project is added in parallel to existing legacy projects. Whatever the reason, once the system is overloaded, delays start to occur. CI results that took only a few minutes when first implemented now take half an hour. The developers become accustomed to doing check-in before lunch. A few months later, lunchtime is not enough anymore. The developers get used to doing check-in at the end of the day and letting CI tests run at night. A few months later … You get the idea. And once again, DevOps solves the problem by buying more computers to execute CI tests.

Once this cycle starts, it is very hard to stop it. The daily pressures and the need to make fast improvements mean that throwing money at the problem is almost always the most logical solution. Every now and then some teams manage to hold a blitz of improvements that makes a difference for a few months, but then the problems return.

Since it is so hard to fix the problem after it's started, prevention is a better approach. Here are three prevention measures you should consider:

1. Invest effort in making sure only robust tests are added to the automation system. Brittle tests will later fail intermittently during runs, wasting engineering time in investigations. This means that every automated test must pass a certain quality bar before being admitted to the system.

   One approach is to create a checklist that each automated test must pass. This checklist should be realistic and short. If the list is too long, it will not be used. (Many years ago I wrote a checklist that took three days to complete; I will let you guess how much use it saw. Half an hour is a much better goal.) Once a workable checklist is in place, it will influence the design and implementation stages. Since test developers know that eventually they will be evaluated against the checklist, many of the checklist requirements will be taken care of during the test implementation phase.

2. Be aware of the impact of a long test time, and invest thought and effort to reduce the time of each test, even when it seems unnecessary.

   When I tested Wi-Fi cards, almost all the tests started with a "Connect to the network" step. To ensure that a connection was achieved, we had a CheckConnection() library function, and almost every test used it. The function guaranteed a good connection by transferring a file back and forth between the Wi-Fi client and the network. To the engineer who designed it, it was a trivial check, with a runtime that looked very reasonable: 15 seconds. For one test, this is indeed not a problem, , but when used by a thousand tests cases, this routine alone was responsible for four hours of test time in each test cycle. Another example is adding a wait() to the test code to give the system time to finish some activity or to stabilize. If a wait time of one second is enough, a wait time of ten seconds, happening again and again, will eventually buy itself a few more computers.

   The best time to streamline a test case is during the test development stage, when the test engineer knows all the considerations and details of the test case. Later, it is much harder to go over all the tests, one by one, and look for possible improvements.

3. Define up front how much test time each test team gets on the system. This targets the tragedy of the commons and can be implemented both as a preventative measure and as an effective route to take when experiencing a bottleneck problem. Practically, this is done by

allocating test machines to each team.

Going back to the previous example, if the system has thirty computers, each of the six teams would get five computers and decide how to use this resource. For team A, the efficiency effort now pays back big time because their test cycle only takes half the time of the other teams. To start with, it looks great in the reports. Additionally, each saved hour is now at team A's disposal to execute new tests. It creates a situation where each team is incentivized to improve their test efficiency.

So the next time you are tempted to solve a problem by throwing money at it, stop for a minute and ask yourself if you are a victim of the tragedy of the commons. Would spending money really get to the root cause of the problem, or would it make better sense to spend effort in making processes more efficient?