SUPPORTING CONTINUOUS TESTING WITH AUTOMATION

BY **BAS DIJKSTRA**

We're living in a world that requires software development organizations to continuously deliver value to their customers. As a result, software development teams need rapid feedback on software quality and its business value. In order to keep up with this demand, many organizations have abandoned their traditional testing efforts in favor of a more flexible, risk-reducing approach, including adopting test automation.

However, teams currently implementing test automation fail to support the constant assessment of quality and business value—a process known as continuous testing.

Let's take a closer look at why automation efforts fall short and what can be done to improve the situation and make test automation the key ingredient for continuous testing.

A Primer on Continuous Testing

Continuous testing can be defined as subjecting every build of an application to a specific set of tests that assess and report on the quality and the business risks for that build, in every environment that build passes through, often including production (figure 1).

Putting the term continuous aside for the moment, these tests should contribute to rapid feedback about the value that the product provides to someone who matters. This is consistent to the definition of quality coined by Jerry Weinberg and adapted by James Bach and Michael Bolton. [1]

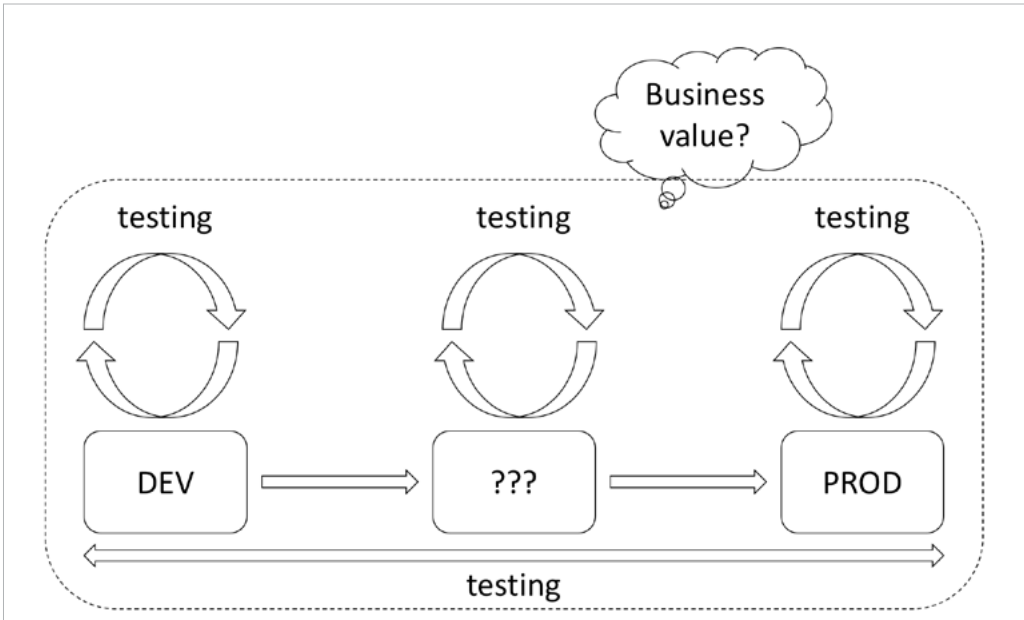

Figure 1: Depiction of continuous testing

Introducing test automation as part of the overall testing approach seems like a straightforward choice, and, indeed, automation can potentially provide considerable value to teams implementing continuous testing. Too often, though, it falls short of promises made and expectations set. Test automation takes too long to run, is hard to maintain, and fails to provide the required insight into application quality. It can mislead its users and stakeholders if the technology results in time and money wasted, applications being released into production without confidence, or bugs going undetected. This can cause automation efforts to be shelved.

In my opinion, there are three reasons test automation fails to deliver in support of continuous testing. Let's look at each of them in detail.

Reason 1: Tests Are Unfit for Continuous Testing

If your automated tests are meant to support continuous testing, they should be able to run continuously or on demand. Whether you're running them once a day or once a minute, your tests should be ready to go and provide the required feedback whenever you need it. Unfortunately, this is not always the case.

To help you make sure that your automated tests are better fit for continuous testing, I use the acronym FITR as a reminder of the qualities to look for when assessing automated tests: Your tests should be focused, informative, trustworthy, and repeatable.

Focused: Having focused tests means that your tests should be tied as closely as possible to the piece of business logic or functionality being tested.

One prime example where tests fail to meet this requirement is when teams write user interface tests. Using a tool like Selenium WebDriver can verify the workings of business logic that's exposed to the user interface through an API, but this can lead to unnecessarily long feedback loops, as UI-driven tests tend to be comparatively slow in execution and can give "shallow" feedback. The further away a test is from the logic it is verifying, the harder it is to pinpoint what, exactly, is wrong when a failure is encountered. The same principle applies to using checks to verify logic that operates at the API level that can be more effectively tested with unit tests.

In short, always make sure that your tests operate at the right level and with the right scope. You should dive deeper into your application under test to find the most efficient way to test anything and everything.

Informative: In order to quickly gauge the effect that a change has on the business value your application provides, your tests should communicate its intent and its result in a clear and unambiguous manner. This minimizes the time needed to investigate failures or to update tests that reflect changes in the application under test.

To improve the way your tests communicate their intent, practice good programming principles like DRY ("don't repeat yourself") and YAGNI ("you ain't gonna need it"), as well as useful naming conventions to make your code readable. Keep your code as close to self-documenting as possible. If it makes sense, consider adding a behavior-driven development library, such as Cucumber or SpecFlow, to document the expected behavior that your tests are verifying.

To improve the way your tests communicate their result, make sure that the target audience is correctly identified for the reports and logging created by your tests. You must also ensure that information requirements are fulfilled and that the audience can make

the appropriate decisions based on these reports. Remember that your target audience can be a person (yourself, another tester, or a product owner) and a machine (such as a build server or deployment pipeline).

Trustworthy: Especially for teams that are practicing continuous delivery or even continuous deployment, automated tests are often the only means used to gauge software quality before it is put into production. This means software development teams need to be able to rely on the outcome of these tests in order to be confident that their product retains a certain degree of quality at all times.

Therefore, a word of caution is in place. Automated tests can trick you with false positives when tests fail but there's no actual defect in the application. Automated tests can also incorrectly report false negatives when tests pass and allow a defect that should have been caught before deployment into production.

To help prevent deceptive automation, review your tests periodically. Fix tests that cause problems, and don't hesitate to delete tests. Experiment with techniques such as mutation testing to help assess the quality of unit tests.

Repeatable: Continuous testing requires that you're able to run your tests on demand. However, there are a number of factors that can prevent you from running your tests over and over again with the click of a button. The two main culprits are test data and dependencies in test environments.

When you're running integration or end-to-end tests, it's highly likely that your test triggers an interaction between different components of your system or between your system and third-party dependencies. This often requires test data to be synchronized between these components. For example, if you're trying to create an insurance policy for a car and part of the application process is a license plate validity check, you'll need to make sure that a license plate number exists in the service each time it is entered. When you're running tests frequently, tests like this can require a lot of test data, and your test data strategy should be able to support that.

A lack of synchronized test data is just one of the ways in which test environments can be a bottleneck when it comes to implementing continuous testing. Critical dependencies required for integration and end-to-end testing can be difficult to access for other reasons, too. Real-world dependencies must be considered, including requiring access fees for third-party services and shared access to mainframes or web services.

One possible approach to dealing with these test environment issues is the implementation of stubbing, mocking, or service virtualization. These are techniques that can be used to simulate the behavior of these critical yet hard-to-access dependencies, thereby removing bottlenecks that are in the way of creating repeatable, on-demand test automation.

Reason 2: Too Many or Too Few Tests

Continuous testing is about continuously gaining insight into the quality and business value of an application using the shortest possible feedback loop. This means there is a trade-off between adding more tests, test cycles, and test types. Increasing test coverage can conflict with the need to get the information required to make decisions (to deploy or not to deploy) as fast as possible.

For every test they are about to include in their continuous testing efforts, software development teams should ask themselves, "Do we really need the information that this test provides every time we build our software?" From a software testing perspective, it is easy to "add another test, just to be sure," but teams should remember that every test added has a negative impact on the length of the feedback loop. From a business value perspective, it might be tempting to cut testing efforts relentlessly in order to minimize the time to production, but this comes with a cost of its own in terms of lower test coverage. It's imperative that for every test that's not added to the continuous testing cycle, teams ask themselves, "Are we OK with not having the information provided by this test for every build?" Performing certain types of tests periodically may be a useful strategy to reach a good trade-off between risk, quality, and speed, but perhaps not on every build.

Too much testing results in unnecessarily long feedback loops and a loss in time to market. Too little testing results in poor insight into application quality before it is put into production, as well as a higher risk of defects finding their way to the customer. As with so many things in life, there is no one right answer or approach that works in all cases. Using the above considerations, find the trade-off that works best for your situation, learn from the outcomes, and continuously monitor and improve your testing efforts.

Reason 3: Not Everything Can Be Automated

Contrary to popular belief, not all testing activities can be automated. I'd like more people to view automation as something that supports testing, rather than something that replaces the things testers do when they test. [2]

Even though automation is likely to be an essential part of most continuous testing efforts for its ability to give rapid feedback, it should be considered a bad idea to rely solely on automation. It cannot paint a complete picture about product quality. On the other hand, there are likely more opportunities for automation to support your continuous testing efforts than you might think. Consider using continuous testing in test data generation, monitoring, log analysis, continuous performance testing, or service virtualization—all forms of automation that use tools for testing. Adopting more innovative test automation coverage can help you achieve your continuous testing goals.

Automation can be an invaluable asset toward a successful adoption of continuous testing, but we should definitely see it in a more realistic light. By adopting the advice in this article, you should be able to get the most out of your automation and your continuous testing efforts. [BSM]

bas@ontestautomation.com

CLICK FOR THIS STORY'S REFERENCES