# Five Tips for Prioritizing Regression Test Automation

It is all too common in DevOps circles to live by the creed: "automate everything." There can be little doubt that automated regression testing is a powerful tool, and when used correctly, can provide an extremely stable automated testing suite, off which can be built drastic and volatile tests. However, regression test automation can be a devilish scoundrel, as poorly designed tests can lead to lousy requirements coverage, wasted development time, or in the worst of cases, broken builds.

In this article, we'll explore five tips to help you prioritize your automated regression testing to ensure the most critical tests are running first and foremost. In doing so, we'll also see a few testing scenarios where automated testing may not be the best solution, and instead, where the delicacy and adaptability of humans via crowdtesting may be favorable.

## 1. Focus on Customers and Revenue

One of the simplest prioritization techniques to implement is the focus on test which are critical to customers. Rather than relying on complex algorithms or metrics to prioritize your test suite, this technique simply requires one or two individuals to go through the test list and evaluate each test case with one thought in mind: **How impactful is this test case for the customer?** A [customer-allotted priority](http://research.ijais.org/volume6/number7/ijais14-451081.pdf) ensures that test priorities remain customer-oriented, by aligning the prospective needs and wants of the customer with the prioritization of the tests within the test suite.

Once customer needs have been established and worked into test ordering, the next logical step is to focus on revenue priority throughout the test suite. Arranging tests based on their likely impact on incoming revenue is a simple, yet powerful, technique that forces the test suite to automatically hone in on the most mission-critical test cases, order to get a fully-functional and well-rounded product out the door at the end of the development life cycle.

## 2. Maintain a Lean Test Suite

Regression testing inherently requires continuous testing throughout the software development life cycle in order to quickly detect faults and implement necessary fixes. As the size of your test suite expands, it becomes an absolute necessity to frequently remove tests from the test suite, lest the suite become too bloated, slow, and unwieldy.

Finding the right balance between the overall requirement coverage and the speed of the test suite can be challenging, so often it is useful to implement a minimization technique to help you keep your test suite as lean and mean as possible. Using a minimization technique, such as the [greedy algorithm](https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/PASTE05.pdf), allows you to easily reduce the size of your test suite in a logical and heuristic manner, by identifying tests which cover the broadest spectrum of faults, and then loosely prioritizing tests within the suite based on that coverage.

Maintaining a smaller test suite is particularly important when dealing with automated testing suites, since regression testing often demands that the test suite is executed with every build and for every change made to the project. While ignoring this requirement and splitting the test suite into smaller chunks is a viable options to reduce execution time, that can potentially cause it's own set of problems if a particular test is delayed too long into development or for two many subsequent builds. Once that test does finally come around again, it may shed light on faults that were unseen before and require *even more* regression than if the full suite had been used in the first place.

## 3. Favor Tests Which Avoid the UI

It is no doubt possible to create automated tests which aim to test the UI and user functionality of the product. Many libraries exist to help developers and QA teams automate testing at the interface level, across a wide range of languages. However, testing at the UI level is largely a practice that should come as a last resort in terms of priority within automated regression testing. By its very nature, the UI is extremely volatile, as often, even when the functionality that is behind the UI is still working correctly, minor changes in the UI itself can cause test failures left and right. Therefore, even in the most well-designed test suites, UI tests will frequently fail during automated regression testing, and require much further analysis or even manual testing to resolve.

Instead, automated regression tests should focus on low-level layers of the product, such as unit testing, the API, and service integrations. Allow the automated power of computers to handle more of the rapid, low-level testing, while allowing humans, via crowdtesting or in-house QA, deal with interface testing. Humans are great at finding "fit-and-finish" issues and they can easily adapt to small changes in the interface, whereas computers and automated tests tend to find it extremely challenging.

## 4. Use Metrics

While there's no perfect prioritization method that fits the needs of every project nor every team, you should make every effort to implement practices which take advantage of common, measurable metrics within the project. Commonly use metrics like the [Average Percentage of Faults Detected](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.378.915&rep=rep1&type=pdf) (APFD) can be utilized to prioritize your test cases, but it's important to analyze the possibilities available to you -- particularly those metrics most relatable to your project -- to find the best solution.

If you can afford the overhead, one metric you can measure to truly help is Total Statement Coverage, which aims to measure how much of your actual code, line-by-line or statement-by-statement, is covered by the test suite. For obvious reasons, using this metric can be extremely cumbersome and requires a great deal of effort on behalf of developers and testers alike to ensure the full code base is covered.

The logical step up from there, and a metric that is far more commonly used, is Total Functional Coverage, which measures the quantity of functions in the code base which are in some way covered by the test suite. Even as a step up from Statement Coverage, it is not always feasible (nor necessary) to build tests which cover every function, so it is common to implement this metric by also tacking on some form of prioritization of your functionality to begin with. Deciding which functions (and therefore what functionality) is most critical, and assigning those higher scores to then use for ordering the tests in the test suite, is a tried and true practice.

It can also be beneficial to step away from the code with your metrics, and instead use a Fault Severity metric to help prioritize your tests. With this technique, each fault that is unearthed during the development life cycle is assigned its own severity rating from low to high. With that simple metric in hand, you're then able to easily prioritize test cases based on the requirements and severity level of the faults which those tests cover.

## 5. Prioritize Unchanging Tests

It's all too common for customers (or even developers) to try to make too many rapid changes to the code base within too short a timespan, which invariably leads to massive volatility, flakier tests, and higher test failure rates. Worst of all, if tests are flaky and are frequently failing, it's often worse to have that flaky test in place than simply having no test there at all. The sheer existence of screwy tests with volatile results requires a wasted time investment from developers to locate the issue and find a solution, all the while sunk costs are climbing and forward momentum on the project is slowing down. In the most dire scenarios, such tests can be build-breakers, preventing the next iteration from moving forward as scheduled.

For these reasons, it's best to prioritize automated tests that handle code and functionality that rarely changes, or when they do change, are subject to fairly minor alterations. By prioritizing these tests higher in the test suite, it generates a strong backbone of automated testing off which to expand. As more and more volatile tests are added to the lower priority levels of the suite, the team can be more confident in those tests which remain at the top of the priority to continue to perform as expected and not cause any breakages along the way.

For test cases which *do* deal with constant changes, it's then often less time investment and lower cost to utilize human intervention. By employing crowdtesters to the most unstable tests and components of the product, there's a nice balance of capabilities, where automated tests are handling the stuff they're good at, while humans are integrated to deal with the volatile and subtler tests.

**SOURCES**

* http://www.sciencedirect.com/science/article/pii/S1877050916001514
* http://research.ijais.org/volume6/number7/ijais14-451081.pdf
* https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/PASTE05.pdf
* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.378.915&rep=rep1&type=pdf