# How Much Regression Testing is Enough?

By most definitions, the term regression simply means *the act of reverting back to a previous state.* Extrapolating a bit on this, in the world of software development and software quality assurance, it stands to reason that regression testing means *the act of testing previously tested states of software to ensure nothing has broken in the interim*. When a piece of software presents a new bug which wasn't present before, but has now appeared following a new change or update, that requires some form of regression testing to duplicate and eventually resolve this issue.

Unfortunately, once proper regression testing practices are in place for any given project, the sheer volume of regression testing that occurs as the size and scope of a project grows can become daunting. It's all too easy for the software quality assurance team to fall into a never-ending chasm of regression testing, where tests are performed on the entire history of bugs that have been discovered up to that point, which is a quantity often in the thousands for larger projects.

In this article, we'll examine a bit more about the practices of proper regression testing, as well as explore how to determine when enough is enough, or when the brakes should be put on your project's regression testing practices a bit.

## The Purpose of Regression Testing

In the world of software quality assurance there exists a term known as software regression, which simply refers to a software bug which causes some kind of unintended non-functionality when a change is made to the system, such as a patch or new release. There are three typical categories of software regressions:

* **Local**: When a new bug is located in the same software component that was updated.
* **Remote**: When a new bug is located in a different software component than the one that was updated.
* **Unmasked**: When the bug already existed, but had no effect prior to the update.

The overall purpose of regression testing is to easily and effectively uncover all possible software regressions, whether they were newly created (local or remote), or previously undiscovered (unmasked).

Furthermore, regression testing comes with a variety of common practices, depending on how thorough and detailed the process needs to be:

* **Retest All**: As the name implies, this method of regression testing is one where the entirety of the system is re-tested. In most cases, the vast majority of this re-testing is performed by automated tools, though often a full retest is simply not feasible for larger systems or as the release count climbs.
* **Regression Test Selection**: Instead of testing everything, this technique allows the testing team to utilize a representative selection of tests that will approximate testing of the entire system, but with far less required time or overhead.
* **Test Case Prioritization**: When specific test cases are in place, it is often preferable to prioritize the testing of said test cases, ahead of those which lower priority. Typically it is best to prioritize test cases which will impact both current and future versions of the software.

## How Much Regression Testing is Enough?

As with nearly every process during the entire software development life cycle of a project, the most precious resource of all is simply **time**. While it may just be a colloquialism at this point to to exasperatedly say, "There's just never enough time in the day," as lookers-on casually roll their eyes, there's a certain degree of truth to this idea, particularly in the realm of software quality assurance. Even for the simplest of projects, finding enough time to properly test every aspect of the software can be difficult if not impossible in most situations.

For example, imagine you were performing tests on a very simple application called Addr (which omits the letter e to be hip, of course). This application simply consists of two text boxes and asks the user to input any two numbers, at which point Addr will magically add the numbers together and output the result. Incredible!

Let's also assume each text box can hold a number no larger than 20 characters in size. This means the user can enter any two numbers up to a maximum value of about 99 quintillion (or 1020-1) to be added together. The question then becomes, how does the software quality assurance department properly test this very simple application?

The primary issue here is: Are there any values that can be entered into the boxes that cause a failure or produce a bug of some kind? In most cases, developers will have placed constraints on the input boxes to prevent non-numeric text from being entered, as well as numbers that are larger than the 20-character limit, and so forth. Even so, testing for those special cases is fairly quick and easy. The real problem comes in identifying which of the trillions of possible number entries might cause an issue.

The first most obvious answer is to use automated testing, which is very common in regression testing as well, and would allow the computer itself to enter all the possible numbers in both boxes and compare output to the expected result, as well as identify any potential bugs. [Computers are very fast](https://computers-are-fast.github.io/), so depending on the testing suite in use, the system it runs on, and so forth, it is likely the computer is capable of testing some five hundred million possible combinations **every second**.

Unfortunately, even at that blazingly fast speed of automated testing, it would still take about 6,338 years for a computer to test every combination. We could also arguably reduce our testing time in half, since we're using two possible entry boxes but likely only need to care about testing each unique pair of numbers one time, so we could split our potential combinations entirely in half, but that still requires about three thousand years. Maybe we can somehow afford to use the services of a huge cloud computing platform to perform our testing calculations, but spreading across even a **thousand machines** means we're still looking at a solid three years of testing.

Now to make things really crazy, imagine we *did* perform this full test, and everything was successful, but suddenly version 1.1 of Addr was released, and it's time for regression testing. Can we assume the functionality still works properly, or do we need to perform our testing once again to ensure nothing was broken during the update?

While this example is for a very simple project, it illustrates just how daunting it can be to find enough time to properly perform adequate regression testing, even on the simplest projects. In most cases -- and certainly so in this example of the near-pointless Addr application -- the best solution is to use a partitioning strategy to help manage the proper balance of regression testing.

Partitioning strategies are simply means for software quality assurance engineers to take a **representative sample** of all possibilities for a problem or test, and make the logical assumption that the results from this sample are characteristic of the entire set as whole.

With a partitioning strategy in place for the Addr application, we know what the maximum possible number is, so we might have the testing suite grab a two completely random numbers from the set of possibilities and add them together and test the results. We can do this a ridiculous number of times, as even performing this test with a set of 100 billion random pairs of numbers, we'd still only need a little over three minutes to run the automated test. Granted, this would leave a huge number of possible numbers outside the scope of our test, but it strikes a nice balance when performing regression testing that is often necessary to ensure a good mixture of speed and thoroughness.

**SOURCES**

* https://en.wikipedia.org/wiki/Regression\_testing
* https://en.wikipedia.org/wiki/Software\_regression
* https://computers-are-fast.github.io/