# Test Smarter, Not Harder

## Introduction: Complexity Leads to Futility

Imagine we are developing a web page for customizing a laptop purchase.

If you've never configured a laptop online before, take a look at Dell's "customize it" page for an entry level laptop. The web page presents eleven questions to the user that have from two to seven responses each. The user has to choose from two options in the first control, two in the second, and so on. The user has seven possible choices for the last control.

When we look at all of the controls combined, the user has to make (2,2,2,2,2,3,2,2,3,4,7) choices. This is a simple configuration problem. The number of possible laptop configurations that could be requested by the user is the product of all of the choices. In this very simple page, there are 32,256 possibilities. At the time of this writing, the page for customizing Dell's high-end laptop has a not dissimilar set of controls, with more choices in each control: (3,3,3,2,4,2,4,2,2,3,7,4,4). The user of this page can request any of 2,322,432 different laptop configurations! If Dell were to add one more control presenting five different choices, there would be over ten million possible combinations!

Creating a test suite that tries all two million combinations for a high-end laptop could be automated, but even if every test took one tenth of second to run, the suite would take over 64 hours! Dell changes their product offerings in less time than that.

Then again, if we use a server farm to distribute the test suite across ten machines we could run it in about 6 hours. Ignoring the fact that we would be running this type of test for each customization page Dell has, 6 hours is not unreasonable.

Validating the two million results is where the really big problem is waiting for us. We can't rely on people to manually validate all of the outputs – it is just too expensive. We could write another program, which inspects those outputs and evaluates them using a rules-based system ("If the user selects 1GB of RAM, then the configuration must include 1GB of RAM" and "The price for the final system must be adjusted by the price-impact of 1GB of RAM relative to the base system price for this model.")

There are some good rules-based validation tools out there, but they are either custom software, or so general as to require a large investment to make them applicable to a particular customer. With a rules-based inspection system, we have the cost of maintaining the rules. The validation rules are going to have to be updated regularly, as Dell changes the way they position, configure, and price their laptops.

Since we aren't Dell, we don't have the scale (billions of dollars of revenue) to justify this level of investment. The bottom line for us is that we can't afford to exhaustively test every combination. Dell's shareholders require them to grow their business, and these configuration pages are the vehicle by which Dell generates billions of dollars in revenue. They have to test it. The cost of errors (crashes, lost sales, mis-priced items, invalid combinations of features) is too high. With this level of risk, the cost of not testing (the cost of poor quality) is extremely high.

## We Can't Afford to Test It

I was able to attend a training session with Kent Beck a few years ago. I was also honored to be able to enjoy a large steak and some cold beer with him that night after the training. When asked how he responds to people who complain about the cost of quality, Kent told us he has a very simple answer: "If testing costs more than not testing then don't do it."

I agree. There are few situations where the cost of quality exceeds the cost of poor quality. These are situations where the needed infrastructure, test-development time, and maintenance costs outweigh the expected cost of having a bug. (The "expected cost" is the likelihood (as a percentage) of the bug manifesting in the field, multiplied by the cost of dealing with the bug.)

The techniques described in this article are designed to reduce the cost of quality, to make it even less likely that "not testing" is the best answer.

## Just Test Everything, It's Automated!

Two "solutions" that we have to consider are to test nothing and to test everything. We would consider testing nothing if we can't afford to test the software. When people don't appreciate the complexities of testing or the limitations of automated testing, they are inclined to want to "test everything." Testing everything is much easier said than done.

Have you ever been on a project where the manager said something like, "I demand full testing coverage of the software. Our policy is zero tolerance. We won't have bad quality on my watch."?

What we struggle with here is the lack of appreciation for what it means to have "full coverage" or any other guarantee of a particular defect rate.

There are no absolutes in a sufficiently complex system–but that's ok. There are statistics, confidence levels, and risk-management plans. As engineers and software developers, our brains are wired to deal with the expected, likely, and probable futures. We have to help our less-technical brethren understand these concepts–or at least put them in perspective.

We may get asked, "Why can't we just test every combination of inputs to make sure we get the right outputs? We have an automated test suite–just fill it up and run it!"

We need to resist the urge to respond by saying, "Monkeys with typewriters will have completed the works of Shakespeare before we finish a single run of our test suite!"

## Solving the Problem

There are a lot of applications that have millions or billions of combinations of inputs. They have automated testing. They have solutions to this problem. We just finished discussing how impractical it is to test exhaustively, so how do companies test their complex software?

In the rest of the article, we will explore the following approaches to solving the problem.

* Random sampling
* Pairwise testing
* N-wise testing

We will also explore the impact that changing the order of operations has on our testing approach, and the methods for testing when the sequence matters.