DIOMIDIS SPINELLI’S “Differential Debugging” article on locating defects (Tools of the Trade column, IEEE Software, Sept./Oct. 2013) prompted a few thoughts on the process of predicting defects.

**Exponential Decay**
Defect detection is generally exponential with time and effort—most defect removal (and knowledge acquisition, which is much the same thing) obeys an inverse exponential law. In testing, finding the first few bugs is usually pretty easy. As each bug is fixed, finding the next bug is harder and usually takes more time. Figure 1 shows that the resulting defect rate/time is an exponential decay curve of the (simplified) form

\[
\text{rate} = a \cdot e^{-\frac{\text{time}}{b}}.
\]

I’ve used this approach in a wide range of applications. In the 1990s, a large telecommunications company used it in a calculation that told us when to stop testing. When testing hits the long tail of the exponential and levels out, little more can be learned through continuing testing in this mode.

**Calibration**
Individual projects can vary quite remarkably, but often, general trends allow us to calculate the values of \(a\) and \(b\) and also each variable’s natural variation. The result tends to be a good guideline for predicting testing effort and duration (with a few caveats).

**Caveats**
Several extremes of defects can occur in systems.

**Linear Defects**
At one end of the defect-detection spectrum, every defect could be linearly dependent on another. This causes defect masking, where owing to the presence of one defect you can’t get to the point of finding the next. So you must fix the first defect, to get to the next defect, and so on. This can make the testing and debugging process very long indeed and linear with respect to time.

**Independent Defects**
At the other extreme, all defects could be entirely independent of each other. This would make defect detection a Markovian process and, in its purest state, would result in a well-defined probability distribution that would result in the exponential decay in Figure 1. In practice, defects exhibit a mixture of this linearity and independence.

**Fixing Defects Causes Defects**
Unfortunately, finding and fixing a defect can cause another defect either through error-masking or simply through a bad fix. This typically has a linear effect on defect fixing since some percentage of fixes go bad. This can often be modeled by simply using a percentage of bad fixes in the formula.

**Existential Defects**
Even more unfortunately, systems can contain existential defects, where a single defect stops the whole thing from working (thereby masking everything, good and bad). I once worked on a system that, as far as I could determine, had a single defect—the system would not load. Of course, since it wouldn’t load, any other defects were both invisible and irrelevant. So there are levels of defect severity: some defects are really important and others not so much. So predicting defect detection should also take into account that some defects just aren’t very important.

**Prediction**
A defect detection prediction mechanism should compensate for all of these. Real systems are somewhere between extremes, but we can make some reasonable assumptions about where that would be, track the performance against those assumptions, and calibrate the equation based on real data.

**More Caveats**
That said, actual defect detection data is usually very noisy, often with a find-a-defect/fix-a-defect periodic cycle. Defect detection is clearly dependent upon the intrinsic quality of the
product, but a threshold exists where if the product is sufficiently bad, testing and debugging is useless or never-ending. So the “real” equation can go nonlinear at this threshold. We could spend time trying to isolate the threshold, but the effort is better spent fixing the process that created the bad product in the first place.

**Second-Order Ignorance**

Testing (and to a smaller extent, debugging) is a *third-order ignorance* process that converts *second-order ignorance* (2OI; what we don’t know we don’t know) into *first-order ignorance* (1OI; what we do know we don’t know). We predominantly test to find out if there’s something in the system about which we’re unaware. We can’t explicitly craft a test for a condition that will fail unless we already know the condition will fail, which means we already know it’s wrong (1OI), which means we should go fix it—thus converting it to *zero-order ignorance* (0OI; what we do know we do know)—rather than test it.

So, we end up having to develop tests for something we aren’t looking for. A scientist can’t develop an experiment for something she isn’t looking for, and testing is in the same boat.

For really rough testing prediction, I’ve actually seen people use a simple multiplier—they estimate what they really think it will take to test a system and just multiply by 2. There’s some sense in that; it often takes two passes to get something right: pass one to figure out what you don’t know (2OI → 1OI) and pass two to figure out the right answer (1OI → 0OI). In any event, predicting testing effort and duration isn’t easy and it tends to be quite variable, but it’s not impossible.

*Phillip G. Armour*