Testing, probably more than any other activity in software development, is about *discovery*.

never occur in the real world, but which we think might expose a so-far unknown limitation in the system. None of these are guaranteed to throw a defect. In fact, nothing in testing is guaranteed, since we don't really know what we are looking for. We are just looking for something that tells us we don't know something. Often this is obvious, as when the system crashes; sometimes it is quite subtle.

The Dual Hypotheses of Knowledge Discovery

The Dual Hypotheses of Knowledge Discovery [3] are:

- We can only discover knowledge in an environment that contains that knowledge.
- The only way to assert the validity of any knowledge is to compare it to another source of knowledge.

The first hypothesis shows us why, sometimes, we cannot test for and detect defects in the lab. If we cannot duplicate, in sufficient detail and with sufficient control, the situations that will occur in the customer's environment when we release the software, we cannot expose these defects. Of course, the customer's environment, not being subject to this limitation, usually has no difficulty in quite publicly demonstrating our lack of knowledge.

The second hypothesis demonstrates the paradox of testing: if I have sufficient knowledge about what is wrong with my system I can create a robust set of test cases and results that will show if there is anything I don't know. But if I do have sufficient knowledge about what I don't know, I must *a priori* know it, which means I have already exposed my ignorance and therefore I don't need to test at all. Testing, it seems, is effective only if we don't need to do it, and is not very effective when we do need to do it.

Paradoctoring the Paradox

How can we effectively address this situation? Our testing heuristics of boundary value analysis and equivalence partitioning help. They point us to the locations of high-density knowledge within our system. We are most likely to make mistakes where complex knowledge is clustered. Where things are complicated we usually understand them less and our ignorance (read defects) is usually higher.

But there is another aspect to consider. I have found good testers have a "nose" for testing. They experience a kind of intuition that tells them what to test and how. A simple example of this in operation can be shown in the layering or sequencing of tests. At the beginning of *The Art of Software Testing*, Myers suggests a self-test to determine your test effectiveness (his phrase). It involves establishing a set of tests for a trivial program that accepts inputs to be used to predict whether the values, if numeric, will describe the sides of an equilateral, an isosceles, or a scalene triangle. Presumably, the program will also indicate if the input values cannot make a triangle at all for some reason. A standard (correct or valid) input test case might be to use the numbers 3, 4, and 5. These numbers, representing the lengths of the sides of a triangle, would produce a right-angled scalene triangle. Assuming the program works well for this input, would it be better to next execute a test for the number set 3, 5, 4? Or how about the number set 4, 5, 6 or 6, 4, 5?

As shown in the table here, the choices in this case are between changing only the order of the inputs, only their values or changing both order and value at the same time. Is it "better" to