THE PRINCE OF TESTERS

A Salute to My Teacher, Gerald M. Weinberg

I knew he wouldn't be able to resist. That's why I set up just 10 feet from Jerry's booth. His keynote speech had gone well, and now he was sitting at a "Meet the Experts" table. Conferees drifted by to ask questions and get a bit of free consulting, but I wanted Jerry's attention for myself. Since I didn't want to ask him to shirk his duty to the conference, my plan was to entice him to shirk *without* having to ask.

So, I challenged another software tester to solve a problem—one that I often pose as part of my coaching practice. It's a card game that resembles the classic Mastermind game, with some twists that make it an open-ended opportunity for experimentation. This is relevant to training testers because each test of a real software product is a kind of experiment. The two of us made a production out of moving a table over toward where Jerry sat, getting chairs, and setting out the playing cards and note paper. I explained the rules and the tester got started.

It was only a few minutes before a shadow suddenly loomed over us.

"This looks interesting." Jerry asked, hovering, having abandoned his booth, leaving petitioners to wander in the wilderness a little while longer.

"Oh. Hi, Jerry. Nice to see you. We're just playing a little testing game, here. You might like it. It's designed to exercise the skills of experiment design, conjecture and refutation, and observation. Tom's goal is to discover which cards I like and which cards I don't. Maybe..." I shrugged elaborately, "You have an idea or two to contribute?"

"Okay, tell me how it works."

"Exx-cellent! Here is a pack of ordinary playing cards. What you do is select four cards and show them to me. That's called a 'showing'. If I like a card in your showing I take it and you don't get it back. If I don't like it, I leave it. You repeatedly show me sets of cards until you think you know the principle I'm using to select the cards. When you've figured that out, show me five cards and declare it as your "final showing". If I take each of those five cards, you win."

"Sounds easy enough. Here's my showing." He placed three cards on the table.

"I need to see four cards at a time, Jerry," I said. "Do you want to hear the rules again?"

"There *are* four cards," he replied with comic indignation. Then he pointed at a napkin on the table. "I call that a card."

"No, that is not a card."

"Tell me more about that. It looks like a card to me. It's made of paper. It's flat."

"It has to be a playing card."

"Well then..." He scribbled a crude drawing of a Jack of Diamonds on the napkin. "Now there are four cards."

"No, I meant a *real* playing card. From your deck."

"Ohhhh. Well, let me see." He picked up the cards and selected four fresh cards from his deck, placing them face down on the table. "Like any of those?"

I reached to turn them face up. "Interesting," mused Jerry as he hastily gathered the cards before I could see them and take any. "You seem to need to see the faces of the cards. That tells me that your algorithm probably has something to do with the semantics of the card faces. Or it could be just your habit. Say, is there another deck around here?"

"Ah, not that I know of, Jerry. Try to focus."

"I just thought if you don't normally look at the backs of the cards, I would slip you a duplicate card from another deck and see what happens. It's not like you would know. But, never mind. Here's another showing." He slapped five cards down in quick succession.

"Is that your final showing?"

"No, it is not."

"Then you must show me only four cards."

"But, James, there *are* only four cards."

"I think I see five cards on the table, Dr. Weinberg."

He gathered the five cards and held up the leftmost. "That's a joker. I didn't know that you would consider it a card. Now I do."

If you're not used to it, Jerry's manner of play might seem silly to you, even exasperating. But to me, it was wonderful. Notice how he did not enter the game with strong assumptions about what the rules meant, or how they mattered. He knew that playing the game "straight" would probably cost him some cards, so he began with several strategies to probe the structure of the problem without much risk to his cards. In so doing, he opened himself to learning about many more potential variables than does a more typical player of the game, at a lower cost. He was practicing the first responsibility of a tester.

Knowing That Things Could Be Different

It was Jerry himself who introduced me to that first responsibility, some years before. I had been visiting him for a week of conversation and writing work, and we were heading up his driveway on one of our walks, when abruptly he asked "What's the first responsibility of a tester?"

My head was immediately brimming with candidates. "I don't think I can pick just one. How about five?"

"I'll tell you what it is," he said, waving me off. "The first responsibility of a tester is to explore and challenge the constraints of the situation. Kind of like how you just challenged the constraints of my question. A tester is someone who knows that things could be different. However they appear, however they really are, whatever we feel about them, it all could be different than it seems, or different tomorrow than it is today. Testers need to be inquisitive; to shake things up a bit, so that people can see things in a different light. Now here's another question for you: What do you do if you don't have enough time to find every bug in a program?"

"That's a trick question. No one ever has the time to find every bug in a program. But that's not our goal. Our goal usually is to find every important bug."

"And if there isn't time to do that?"

"Well, that's why we use our risk analysis to guide us. We put our efforts where they are most needed."

"And if your risk analysis is wrong?"

"That's why we also use a non-risk-oriented test strategy for some of the testing. For instance, we might use a coverage-oriented test technique to scan for unknown risks."

"And if you aren't given enough time to do that well? Or maybe you have the time and you do it badly?"

"I suppose we just keep trying, and we also try to learn from our mistakes."

"True, but you might not learn the right things from your mistakes. Do you see what I'm getting at, James?" I shrugged expectantly and he continued. "Beyond any clever strategy you try, there is something further that you need: you need a philosophy of acceptance."

"A philosophy of acceptance? We just accept failure?"

"You accept reality. That's a tester's job. Your strategies will occasionally fail, no matter what you do. While a tester's first responsibility is to refuse to accept the apparent constraints of the situation, a tester must ultimately accept that some ambiguities and constraints may never be identified or resolved. It's an awkward field. If you wanted solid ground to stand on, you chose the wrong vocation."

I had to smile at that. "Must be why I like it, Jerry. Solvable problems are so tedious."

Testing By Ignoring Testing Problems

Jerry Weinberg is not much known as a tester. He is famous more as a programmer, a teacher of programmers, a systems thinker, a teacher of systems thinkers, a teacher of technical leaders, a writer, a teacher of writers, and very prominently in recent years: a teacher of teachers. Still, I consider Jerry to be one of the great testers in the history of computing. His work profoundly influences my practice.

As I see it, the reason Jerry is not recognized as a tester is simple. Bluntly put, sometime around 1972, the testing field was co-opted by technocrats and process enthusiasts. Their passions were honorable, but those passions tended to marginalize the systemic, social, and psychological aspects of testing (you might say these are "the icky parts").

Sure, there was a lot of testing going on in the sixties, but the testing field had not yet defined itself, as such. The testing literature of the time described the activity as part of programming. In the IFIPS conference proceedings of the mid-sixties, rarely are dedicated testers or testing teams mentioned. Testing was often not distinguished from debugging. It was not until 1972 that the first book dedicated to testing was published, *Program Test Methods*, edited by Bill Hetzel. The book is essentially the proceedings of the Computer Program Test Methods Symposium held at the University of North Carolina at Chapel Hill.

As I analyze it, the Chapel Hill approach pursued two threads of development to make testing more reliable and manageable:

1. The technocratic thread: Develop tools and techniques to systematically discover certain kinds of defects reliably (based on certain assumptions about the availability of complete specifications, sufficient time, sufficient tester skill and knowledge, availability of source code, a product that

was not too complex, teams of people who follow methodical rules of behavior, and competent management).

2. The process control thread: Develop procedures and documents that humans would follow to discover certain defects reliably (based on certain assumptions about the availability of complete specifications, sufficient time, sufficient tester skill and knowledge, availability of source code, a product that was not too complex, and competent management).

And lo! Testing would enter its overproduced disco phase without ever having had its Summer of Love. Almost every testing book and academic testing paper since Hetzel has taken the same basic techno-procedural approach expressed in Program Test Methods. Except for a handful of anachronisms (such as no mention of brand name test tools), the material in Hetzel's compilation would not raise eyebrows in the popular testing conferences today. That's how far we *haven't* come as a field.

The prime assumption behind the two threads is that testing is an ordinary technical problem. This assuption is invalid. And not just a little invalid. Especially in retrospect, it's clear that most of the testing problem (the social and the learning aspects of it) was ignored in order to make the remaining bit seem tractable. What we now know is that we can indeed find certain kinds of bugs reliably using formulaic techniques and tools, but many, many other kinds of bugs cannot be found or prevented that way. We also know that writing detailed test procedures doesn't scale. It's too slow and expensive, and scripted testing simply doesn't find many bugs.

Even at the time, Hetzel and others acknowledged that they were making a leap of faith.

Consider these snippets from the section written by Hetzel called "Principles of Computer Program Testing":

"First we consider the problem of testable specifications. The common practice now is of course to leave much to implicit understanding. The resulting problems are well known. Ambiguity and misunderstanding are the cause of almost as many errors as logical coding... Ideally, we would like the setting of the specifications to be independent of the testing process. In actual practice, the specification set is continually refined throughout program development and testing. Such feedback is simply forced by the imperfection of written specification methods."[1]

Notice that Hetzel saw the problem, but interpreted it as one of specification methodology.

But what if the people involved don't know what they want at the start of a complex project? What if they just don't understand the implications of what they want? If that were true, then better specification methods wouldn't help. Instead, testing needs to be an open-ended investigative process. The purpose of testing would be to help people who have varying desires and perspectives, come to a better understanding of what the product is and what it should be.

Never mind that. Improving specification methods seems pretty doable to Hetzel:

1 Hetzel, William C. 1973. Program test methods. Prentice-Hall series in automatic computation. Englewood Cliffs, N.J.: Prentice-Hall.

"It seems inevitable to me that reliable software will demand automatic examination. This in turn means unambiguous testable specification languages must evolve to allow descriptions about program behavior and the environments in which they operate."[2]

...but he's not sure...

"Unambiguous specification languages and new logical reduction schemes which might make the goal of practical exhaustive testing a reality are far from available. In general, the methodology, techniques, and theory of system testing are entirely inadequate."[3]

Thirty-six years later, unambiguous specifications languages *are* available. Almost nobody uses them. Pick up the nearest specification, and you are likely to find it badly written and out of date. In fact, the Agile development trend has aggressively moved *away* from detailed and unambiguous specifications. Yet, the Chapel Hill attitude is still being touted in large organizations, still consuming enormous resources, and still falling far short of the promise.

For testing to progress as a discipline, it must first acknowledge and address the fundamental problem of testing: How can we observe, interact with and learn about, an arbitrarily complex and volatile artifact of technology, created by people who don't fully understand what they are doing, created for people we may never meet, that may be used in places and in ways we may not anticipate, so as to learn about important problems it may have before it's too late and without driving our company out of business in the process?

Let me come back to Jerry. As I was reading Hetzel's book, clucking to myself, I stumbled across an unexpected passage contained in a section written by Fred Gruenberger. He is criticizing the poor quality of testing advice given in programming textbooks, and then this, "I have found only one text treatment that merits endorsement: the section of “Preliminary Testing” in Leed's and Weinberg's *Computer Programming Fundamentals*."[4]

It turns out that Jerry wrote about testing in 1961! He wrote about it as an intellectually active process. "The testing of a program, properly approached, is by far the most intriguing part of programming. Truly, the mettle of the programmer is tested along with the program. No puzzle addict could experience miraculous intricacies and subtleties of the trail left by a program gone wrong... Testing out a program is seldom a step-by-step procedure. We normally must circle around, repeating ourselves, encompassing more of the total program each time we make a pass through it."[5] This passage anticipates what would come to be called exploratory testing, many years later. Notice how this passage focuses on tester, not testing artifacts or tools. That’s an unbroken thread throughout Jerry’s work.

He also provided a simple, compelling heuristic for test design, "Test the normals, test the extremes, test the exceptions", [ibid.] along with a vivid example of what happened on a bank software project where that guideline was not applied.

---

2 ibid.

3 ibid.

4 ibid.

5 Leeds, H. D., and Weinberg, G.M. Computer Programming Fundamentals. McGraw-Hill, New York, 1961

Jerry has been serious about testing for a long, long time. He was one of the earliest advocates of independent software testing teams. In 1961, when IBM held the Bald Peak conference to discuss software development methodology, some 400 programmers were invited. At the time that represented most of the programmers in the world. Jerry's group was the only one at that conference with a dedicated test team. (Bald Peak was also the event that introduced the notorious term "beta" into the lexicon of software development.)

Once I looked into it, I found other early works of Jerry's that show his way of thinking. Three of these publications, all developed during the sixties, depict what testing can be, how to think about it, and how it might be studied. The assumptions and approach he takes with them is radically different from that of the Chapel Hill Symposium.

Introduction to General Systems Thinking, 1974

This book was a long time in the writing. It developed out of problem solving classes Jerry taught in the 60's.

Jerry calls general systems thinking the science of simplification. It is concerned with the study of complex and open systems in general, rather than with any specific system. The aim of general systems thinking is to help us understand the general relationships between systems and our simplified models of them, so that we can better observe, learn about, interact with and build specific systems.

This is exactly what testers do! This is the critical element of test design. Unlike the Chapel Hill approach, which simplified the testing problem essentially by defining all the hard parts of testing (such as learning what matters and what doesn't) as someone else's job, Jerry's idea was to teach people how to approach an arbitrarily complex situation without fear, and to be a part of the simplification process. There is very little mention in the Chapel Hill material of how to think about an open-ended testing situation, or how to develop skills as a tester. Jerry deals directly with that problem.

General systems thinking is essentially the logic of software testing. Jerry uses a running example, in the book, of three different people trying to analyze and describe a mysterious invention. I consider *Introduction to General Systems Thinking* to be the first true textbook on foundational skills of testing. Still, it is virtually unknown by the heirs of Chapel Hill.

Experiments in Problem Solving, Ph.D. dissertation, 1965

Unlike almost all the other famous names in software testing, Jerry's Ph.D. is in the field of psychology. Jerry studied how people solve problems. His research involved what I would call a problem in exploratory testing. He showed his subjects a running program and they had to predict how it would behave. Jerry approaches the problem like a good social scientist, embracing the complexity and subtleties of the situation:

"There is no way to be certain that all subjects share the experimenter's view of what the problem is, and the more complex the problem we are dealing with, the less chance there is of anything being shared among the subjects. Why not give up the futile attempts to force the subjects into a Procrustean bed and design experiments which measure their differences, not conceal them? To take this approach, we must

abandon the idea of measuring 'success', at least in any fixed way, because each of the subjects—having a different view of what the task is—is working toward a different measure of success."[6]

He approached his research as an exploratory study, but the precision and richness of his measurements allowed for rigorous analysis, too:

"One of the frightening things about the richness of this set of experiments is the way new insights keep turning up each time the 10,000 or so bits of each experiment are recombined in some new manner. How many times in the history of science have discoveries been left hidden for want of the right point of view, as when Uranus appeared on the photographs of several astronomers before its 'discovery'?"[7]

Jerry's research, completed the year before I was born, continues to inform the development, in my own community, of experiments and experiential learning exercises that help us develop better testers and better heuristics for testers.

Natural Selection as Applied to Computers and Programs, 1970

In this paper[8], originally written in 1967, Jerry shows how a program can be viewed as an adaptive system that is bound by the dynamics of Darwin's great hypothesis. The upshot of this, in modern terms, is a criticism of scripted regression testing. Jerry warns that when a system that continues to change or is in a changing environment is subjected to a fixed set of tests, it will inevitably over-adapt to those tests, leading to a higher probability of severe and surprising failures in the field. This will happen regardless of how good those tests were, at the time they were first designed.

Jerry presents testing, in this paper, as a dynamic pursuit, rather than a problem of producing a set of fixed artifacts.

The Father of Software Testing

When I started researching this piece, I intended to set out my case for why Jerry Weinberg should be considered the father of software testing. My reading convinced me that he's *not* the father of what most people know *today* as software testing. You might say that the testing baby was stolen at Chapel Hill. Rather, along with a few others, such as physicist Richard Feynman, and philosophers David Hume, Sextus Empiricus, and Socrates, I would call Jerry Weinberg a prince of testers.

Though it may take many years and a revolution in our educational system, I believe someday his approach will come to dominate the field. It deserves to, because it works. I'm talking about his

---

6 Weinberg, Gerald Marvin. 1965. Experiments in Problem-Solving. Dissertation (Ph.D.)--The University of Michigan, p. 234, http://www.satisfice.com/articles/weinberg.pdf

7 ibid., p. 45

8 Weinberg, Gerald M., 1970, "Natural Selection as Applied to Computers and Programs." General Systems 15

experiential methods of teaching, his insistence on dealing with the world in all its complexity, and his application of psychology to technical projects.

I can't resist a final quote from one of Jerry's papers. It describes an experiment where two instructors attempt to demonstrate by example what their students believed was impossible:

"[This paper] is about the way people write programs, and teach others to write programs. We believe that programming is a practical subject, not a mathematical one, and must be taught by instructors who are prepared to demonstrate how the principles they espouse may be put into action. We believe that 'structured programming' does not mean some rigid set of mathematical rules imposed on programmers, but an attitude about programming that says you can always improve if you only examine the way you currently do things. If, through exercises such as these, frankly discussed with our students, we can make them program self-consciously, we shall have succeeded as teachers."[9]

Jerry has done the same with testing. In so doing, he has inspired me and many others to continue his work.

9 Plum, T. W. and Weinberg, G. M. 1974. Teaching structured programming attitudes, even in APL, by example. In Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education SIGCSE '74. ACM, New York, NY, 133-143.