30

Repeatable Proof

3. I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.

Does that sound unreasonable to you? Does it sound unreasonable to be expected to prove that the code you've written actually works? Allow me to introduce you to Edsger Wybe Dijkstra.

Dijkstra

Edsger Wybe Dijkstra was born in Rotterdam in 1930. He survived the bombing of Rotterdam and the German occupation of the Netherlands, and in 1948, he graduated high school with the highest possible marks in math, physics, chemistry, and biology.

In March of 1952, at the age of 21, and just nine months before I would be born, he took a job with the Mathematical Center of Amsterdam as the Netherlands' very first programmer.

In 1957, he married Maria Debets. In the Netherlands at the time, you had to state your profession as part of the marriage rites. The authorities were unwilling to accept "programmer" as his profession. They'd never heard of such a profession. So he settled for "theoretical physicist."

In 1955, having been a programmer for three years, and while still a student, he concluded that the intellectual challenge of programming was greater than the intellectual challenge of theoretical physics, and as a result chose programming as his long-term career.

In making this decision, he conferred with his boss, Adriaan van Wijngaarden. Dijkstra was concerned that no one had identified a discipline or science of programming, and that he would therefore not be taken

seriously. His boss replied that Dijkstra might very well be one of the people who would make it a science.

In pursuit of that goal, Dijkstra was compelled by the idea that software was a formal system, a kind of mathematics. He reasoned that software could become a mathematical structure rather like Euclid's Elements—a system of postulates, proofs, theorems, and lemmas. And so he set about to create the language and discipline of software proofs.

Proving Correctness

Dijkstra realized that there were only three techniques we could use to prove the correctness of an algorithm: enumeration, induction, and abstraction. Enumeration is used to prove that two statements in sequence, or two statements selected by a boolean expression, are correct. Induction is used to prove that a loop is correct. Abstraction is used to break groups of statements into smaller provable chunks.

If this sounds hard, it is. As an example of just how hard this is, I have included a simple Java program for calculating the remainder of an integer, and my handwritten proof of that algorithm below it.[1]

---

1. This is a translation into Java of a demonstration from Dijkstra's work.

```
public static int remainder(int numerator, int denominator) (
  assert(numerator > 0 && denominator > 0);
  int r = numerator;
  int dd = denominator;
  while(dd<=r)
    dd *= 2;
  while(dd != denominator) {
    dd /= 2;
    if(dd <= r)
      r -= dd;
  }
  return r;
}
```

Don't worry. I don't expect you to read my proof. I just want you to get a feel for what such a proof entails.

I think you can see the problem with this approach. Indeed, this is something that Dijkstra complained bitterly about:

"Of course I would not dare to suggest (at least at present!) that it is the programmer's duty to supply such a proof whenever he writes a simple loop in his program. If so, he could never write a program of any size at all."[2]

---

2. [SP72].

Dijkstra's hope was that such proofs would become more practical through the creation of a library of theorems, again similar to Euclid's Elements.

But Dijkstra did not understand just how prevalent and pervasive software would become. He did not foresee in those early days that computers would

outnumber people and that vast quantities of software would be running in the walls of our homes, in our pockets, and on our wrists. Had he known, he would have realized that the library of theorems he envisioned would be far too vast for any mere human to grasp.

And so Dijkstra's dream of explicit mathematical proofs for programs has faded into near oblivion. Oh, there are some holdouts who hope against hope for a resurgence of formal proofs, but their vision has not penetrated very far into the software industry at large.

But while the dream may have passed, it drew something deeply profound in its wake. Something that we use today, almost without thinking about it.

Structured Programming

In the early days of programming, the 1950s and 1960s, we used languages like FORTRAN. Have you ever seen FORTRAN? Here, let me show you what it was like.

```
        WRITE(4,99)
99      FORMAT(" NUMERATOR:")
        READ(4,100)NN
        WRITE(4,98)
98      FORMAT(" DENOMINATOR:")
        READ(4,100)ND
100     FORMAT(I6)
        NR=NN
        NDD=ND
1       IF(NDD-NR)2,2,3
2       NDD=NDD*2
        GOTO 1

3       IF(NDD-ND)4,10,4
4       NDD=NDD/2
        IF(NDD-NR)5,5,6
5       NR=NR-NDD
6       GOTO 3

10      WRITE(4,20)NR
20      FORMAT(" REMAINDER:",I6)
        END
```

This little FORTRAN program implements the same remainder algorithm as the earlier Java program.

Now, I'd like to draw your attention to those `GOTO` statements. You probably haven't seen statements like that very often.

The reason you haven't seen statements like that very often is that nowadays we look upon them with disfavor. In fact, most modern languages don't even have `GOTO` statements like that anymore.

Why don't we favor `GOTO` statements? Why don't our languages support them anymore? Because in 1968, Edsger Dijkstra wrote an article for the *Communications of the ACM*. The editor, Niklaus Wirth, thought it was so profound that he bypassed the normal review process and published it as a letter to the editor in the March issue. He gave it the title "Go To Statement Considered Harmful."

Why did Dijkstra consider the `GOTO` statement to be harmful? It all comes back to the three strategies for proving a function correct: enumeration, induction, and abstraction.

Enumeration depends upon the fact that each statement in sequence can be analyzed independently, and that the result of one statement feeds into the next.

It should be clear to you that in order for enumeration to be an effective technique for proving the correctness of a function, every statement that is enumerated must have a single entry point and a single exit point. Otherwise, we could not be sure of either the inputs or the outputs of a statement.

What's more, induction is simply a special form of enumeration, where we assume the enumerated statement is true for some x, and then prove by enumeration that it is true for $x + 1$.

Thus, the body of a loop must be enumerable. It must have a single entry and a single exit.

`GOTO` is considered harmful because a `GOTO` statement can jump into or out of the middle of an enumerated sequence. `GOTO`s make enumeration

intractable, making it impractical to prove an algorithm correct by enumeration or induction. So, in order to keep code provable, Dijkstra recommended that it be constructed out of three standard building blocks.

- **Sequence:** Depicted as two or more statements ordered in time. This represents nonbranching lines of code.
- **Selection:** Depicted as two or more statements selected by a predicate. This represents `if` / `else` and `switch` / `case` statements.
- **Iteration:** Depicted as a statement repeated under the control of a predicate. This represents a `while` or `for` loop.

Dijkstra knew[3] that any program, no matter how complicated, can be composed of nothing more than these three structures, and that programs structured in that manner are provable.

---

3. Because of the work of Böhm and Jacopini. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM* 9, no. 5 (May 1966): 366–371.

He called the technique *structured programming*.

Why is this important if we aren't going to write those proofs? If something is provable, it means you can reason about it. If something is unprovable, it means you cannot reason about it.

And if you can't reason about it, you can't properly test it.

Functional Decomposition

In 1968, Dijkstra's ideas were not immediately popular. Most of us were using languages that depended upon `GOTO`. So the idea of abandoning `GOTO` or imposing discipline on `GOTO` was abhorrent.

The debate over Dijkstra's ideas raged for several years. We didn't have an internet in those days, so we didn't use Facebook memes or flame wars. But we did write letters to the editors of the major software journals of the day.

And those letters raged. Some claimed Dijkstra to be a god. Others claimed him to be a fool. Just like social media today; except slower.

But in time, the debate slowed, and Dijkstra's position gained more and more support. Until nowadays, most of the languages we use simply don't have a `GOTO`.

Nowadays, we are all structured programmers, because our languages don't give us a choice.

We all build our programs out of sequence, selection, and iteration. And very few of us make regular use of unconstrained `GOTO` statements.

An unintended side effect of composing programs from those three structures was a technique called *functional decomposition*.

Functional decomposition is the process whereby you start at the top level of your program and recursively break it down into smaller and smaller provable units. Indeed, this is the reasoning process behind structured programming. Structured programmers reason from the top down through this recursive decomposition into smaller and smaller provable functions.

This connection between structured programming and functional decomposition was the basis for the structured revolution that took place in the '70s and '80s.

People like Ed Yourdon, Larry Constantine, Tom DeMarco, and Meilir Page-Jones popularized the techniques of structured analysis and structured design during those decades.

Test-Driven Development et al.

Test-driven development (TDD), the red → green → refactor cycle, is functional decomposition. After all, you have to write tests against small bits of the problem. That means that you must functionally decompose the problem into testable elements.

That means that every system built with TDD is built from functionally decomposed elements that conform to structured programming. And that means that the system they compose is provable.

And the tests are the proof. Or, rather, the tests are the *theory*.

The tests created by TDD are not a formal mathematical proof, like Dijkstra wanted. In fact, Dijkstra is famous for saying that tests can only prove a program wrong; they can never prove a program right.

This is where Dijkstra missed it, in my opinion. Dijkstra thought of software as a kind of mathematics. He wanted us to build up a superstructure of postulates, theorems, corollaries, and lemmas.

Instead, what we have realized is that software is a kind of science. We validate that science with experiments. We build up a superstructure of theories based upon passing tests. Just like all other sciences do.

Have we proven the theory of evolution, or the theory of relativity, or the Big Bang theory, or any of the major theories of science? No. We can't prove them in any mathematical sense.

But we believe them, within limits, nonetheless. Indeed, every time you get into a car or an airplane, you are betting your life that Newton's laws of motion are correct. Every time you use a GPS you are betting that Einstein's theory of relativity is correct.

The fact that we have not mathematically proven these theories correct does not mean that we don't have sufficient proof to depend upon them, even with our lives.

That's the kind of proof that TDD gives us. Not formal mathematical proof; but experimental empirical proof. The kind of proof we depend upon every day.

And that brings us back to the third promise in the Oath:

I will produce, with each release, a quick, sure, and repeatable demonstration that every element of the code works as it should.

Quick, sure, and repeatable. *Quick* means that the test suite should run in a very short amount of time. Minutes instead of hours.[4]

4. James T. Kirk, *Star Trek II: The Wrath of Khan*.

Sure means that when the test suite passes, you know you can ship.

Repeatable means that those tests can be run by anybody at any time to ensure that the system is working properly. Indeed, we want the tests run many times per day.

Some may think that it is too much to ask that programmers supply this level of proof. Some may think that programmers should not be held to this high a standard. I, on the other hand, can imagine no other standard that makes any sense.

When a customer pays us to develop software for them, aren't we honor bound to prove, to the best of our ability, that the software we've created does what that customer has paid us for?

Of course we are.

We owe this promise to our customers, our employers, and our teammates. We owe it to our business analysts, our testers, and our project managers.

But mostly we owe this promise to ourselves. For how can we consider ourselves professionals if we cannot prove that the work we have done is the work we have been paid to do?

What you owe, when you make that promise, is not the formal mathematical proof that Dijkstra dreamed of; rather, it is the scientific suite of tests that covers all the required behavior, runs in seconds or minutes, and produces the same clear pass/fail result every time it is run.