TEX and Software Engineering

Victor Eijkhout

August 2004

(Quotes by Knuth in this chapter taken from [1].)

1 Extremely brief history of TEX

Knuth wrote a first report on TEX in early 1977, and left it to graduate students Frank Liang and Michael Plass to implement it over the summer. Starting from their prototype, he then spent the rest of 1977 and early 1978 implementing TEX and producing fonts with the first version of METAFONT.

TEX was used by Knuth himself in mid 1978 to typeset volume 2 of The Art of Computer Programming; it was in general use by August of 1978. By early 1979, Knuth had written a system called Doc that was the precursor of WEB, and that produced both documentation and a portable Pascal version of the source; the original program was written in Sail.

In 1980 Knuth decided to rewrite TEX and METAFONT. He started on this in 1981, and finished, including producing the five volumes of *Computer and Typesetting*, in 1985.

2 TEX's development

2.1 Knuth's ideas

Inspecting the work of his students, Knuth found that they had had to make many design decisions, despite his earlier conviction to have produced 'a reasonably complete specification of a language for typesetting'.

> The designer of a new kind of system must participate fully in the implementation.

Debugging happened in about 18 days in March 1978. Knuth contrasts that with 41 days for writing the program, making debugging about 30% of the total time, as opposed to 70% in his earlier experience. The whole code at that time was under 5000 statements. The rewritten TEX82 runs to about $14\,000$ statements, in 1400 modules of WEB.

He considered this his first non-trivial program written using the structured programming methodology of Dijkstra, Hoare, Dahl, and others. Because of the confidence this gave him

in the correctness of the program, he did not test TEX until both the whole program and the fonts were in place. 'I did not have to prepare dummy versions of non-existent modules while testing modules that were already written'.

By mid 1979, Knuth was using TEX, and was improving TEX 'at a regular rate of about one enhancement for every six pages typed'.

Thus, the initial testing of a program should be done by the designer/implementor.

Triggered by a challenge of John McCarthy, Knuth wrote a manual, which forced him to think about TEX as a whole, and which led to further improvements in the system.

The designer should also write the first user manual.

'If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.'

Knuth remarks that testing a compiler by using it on a large, real, input typically leaves many statements and cases unexecuted. He therefore proposes the 'torture test' approach. This consists of writing input that is as far-fetched as possible, so that it will explore many subtle interactions between parts of the compiler. He claims to have spent 200 hours writing and maintaining the 'trip test'; there is a similar 'trap test' for METAFONT.

2.2 Context

Software engineering is not an exact science. Therefore, some of Knuth's ideas can be fit in accepted practices, others are counter. In this section we will mention some schools of thought in software engineering, and see how Knuth's development of TEX fits in it.

2.2.1 Team work

The upshot of the development of TEX seems to be that software development is a one-man affair. However, in industry, programming teams exist. Is the distinction between TEX and commercial products then that between academic and real-world?

Knuth's ideas are actually not that unusual. Programming productivity is not simply expressible as the product of people and time, witness the book 'The Mythical man-Month'. However, some software projects are too big, even for one *really* clever designer / programmer / tester / manual writer.

Dividing programming work is tricky, because of the interdependence of the parts. The further you divide, the harder coordination becomes, and the greater the danger of conflicts.

Harlan Mills proposed that software should be written by groups, where each group works like a surgical team: one chief surgeon who does all the real work, with a team to assist in the more mundane tasks. Specifically:

- The Chief Programmer designs the software, codes it, tests, it, and writes the documentation.
- The Co-Pilot is the Chief Programmer's alter ego. He knows all the code but writes none of it. He thinks about the design and discusses it with the Chief Programmer, and is therefore insurance against disaster.

- The Administrator takes care of the mundane aspects of a programming project. This can be a part-time position, shared between teams.
- The Editor oversees production of documentation.
- Two Secretaries, one each for the Administrator and Editor.
- The Program Clerk is responsible for keeping records of all code and the test runs with their inputs. This post is also necessary because all coding and testing will be matter of public record.
- The Toolsmith maintains the utilities used by the other team members.
- The Tester writes the test cases.
- The Language Lawyer investigates different constructs that can realize the Chief Programmer's algorithms.

With such teams of 10 people, coordination problems are divided by 10. For the overall design there will be a system architect, or a small number of such people.

Recently, a development methodology name 'Extreme Programming' has become popular. One aspect of this is pair programming: two programmers share one screen, one keyboard. The advantage of this is that all code is immediately reviewed and discussed.

2.2.2 Top-down and bottom-up

Knuth clearly favours the top-down approach that was proposed by Nicklaus Wirth in 'Program Development by Stepwise Refinement' [2], and by Harlan Mills, who pioneered it at IBM. The advantage of top-down programming is that the design of the system is set from the beginning. The disadvantage is that it is hard to change the design, and testing that shows inadequacies can only start relatively late in the development process.

Bottom-up programming starts from implementing the basic blocks of a code. The advantage is that they can immediately be tested; the disadvantage is that the design is in danger of becoming more ad hoc.

An interesting form of bottom-up programming is 'test-driven development'. Here, first a test is written for a unit (a 'unit test'), then the code. At all times, all tests need to be passed. Rewriting code, usually to simplify it, with preservation of functionality as defined by the tests, is known as 'refactoring'.

2.2.3 Program correctness

The Trip test is an example of 'regression testing': after every change to the code, a batch of tests is run to make sure that previous bugs do not reappear. This idea dates back to Brooks; it is an essential part of Extreme Programming.

However, the Trip test only does regression testing of the whole code. TDD uses both Unit tests and Integration tests. A unit is a specific piece of code that can easily be tested since it has a clear interface. In testing a unit, the code structure can be used to design the tests. Integration testing is usually done as Black Box testing: only the functionality of an assemblage of units is known and tested, rather than the internal structure. One way of doing integration testing is by 'equivalence partitioning': the input space is divided into

classes such that within each classes the input are equivalent in their behaviour. Generating these classes, however, is heuristic, and it is possible to overlook cases.

On the opposite side of the testing spectrum is program proving. However, as Knuth wrote in a memo to Peter van Emde Boas: ‘Beware of bugs in the above code; I have only proved it correct, not tried it.’

Contents

References

[1] D.E. Knuth. The errors of TEX. *Software Practice and Experience*, 19:pages = 607–681.

[2] Niklaus Wirth. Program development by stepwise refinement. *Comm. ACM*, 14:221–227, 1971.