The Fumble Programmer

Roderick Chapman

Protean Code Limited

Bath, UK[1]

Abstract *This paper reflects on the need for formality, discipline and humility in programming. Starting with the work of Turing, Dijkstra and Humphrey, this paper goes on to cover our experience with the Personal Software Process, formal programming with SPARK, and the impact of putting the two together.*

1 The Humble Programmers

I would like to open with something of a retrospective on three "Humble Heroes" of software engineering. From these works, a theme emerges that will be built on later.

2.1 Turing (1947)

Turing's lecture to the London Mathematical Society in February 1947 (Turing 1947) is perhaps not as well-known as his more famous work on the halting problem and Enigma, but is something of a tour-de-force. It describes, in some detail, the design of the Automatic Computing Engine (ACE), including treatment of:

- Memory (using Mercury delay lines)
- I/O (punch-cards)
- Control Structures (including identification of the need for conditional loops)
- Arithmetic (including the design of a binary full-adder circuit)
- Programming.

The last of these topics is of greatest interest here. Over the years, I have seen a few choice quotations from this section, but reading the whole thing reveals that Turing had a deep insight into the problems of programming. Not bad, considering that the machine hadn't even been built at the time!

The first show-stopper comes early:

[1] rod@proteancode.com

> "The programming should be done in such a way that the ACE is frequently investigating identities which should be satisfied if all is as it should be."

In other words, Turing has identified the need for invariant assertions to be checked continuously at runtime in a program. He goes on to note that the number of programs that might be written is rather enormous:

> "It will be seen that the possibilities as to what one may do are immense."

And advises:

> "One of our difficulties will be the maintenance of an appropriate discipline, so that we do not lose track of what we are doing."

Today the "appropriate discipline" might have a new-fangled name like "methodology" or "process" but the message is clear. The entire history of software engineering might be written in terms of the rise and fall, success and failure of various "disciplines" from structured programming, to object-oriented programming, agile methods and whatever is currently fashionable.

Turing goes on to predict the rise of automation in software production, noting that the programming process will become automated

> "...as soon as any technique becomes at all stereotyped..."

He notes that "The Masters" (i.e. the programmers) won't like this development at all, since their "Master" status will be undermined by the very machine that they've created. With (possibly) his tongue in his cheek, Turing warns that

> "They may be unwilling to let their jobs be stolen from them in this way. In that case they would surround the whole of their work with mystery and make excuses, couched in well chosen gibberish, whenever any dangerous suggestions were made."

In short, and in today's language, "Hero/Ninja/Rock-Star Programmers" are prone to choosing notations and approaches that protect their job and "hero status", not necessarily to the long-term benefit of the project or business.

Turing goes on talk about the languages that we'll need to facilitate the programming task. He quickly identifies that the machine is, essentially, a high-speed moron:

> "The machine interprets whatever it is told in a quite definite manner without any sense of humour or sense of proportion. Unless in communicating with it one says exactly what one means, trouble is bound to result."

Finally, he notes that the programming notation should be "as unambiguous as possible", essentially identifying that the programming language should be a *formal* language.

In summary, by early 1947, Turing had identified that programming needed to be a *disciplined* activity and would need *formal* and *unambiguous* languages to succeed.

We might note that it didn't quite work out that way. Let's jump forward 25 years to 1972.

2.2 Dijkstra (1972)

Dijkstra's 1972 Turing Award lecture "The Humble Programmer" contains the well-known and often-cited note about how testing shows "the presence of bugs, but not their absence." Once again, though, a full read-through provides further insights that are worth noting for their relevance and, in retrospect, how well we've (not) done in heeding his words.

The paper opens with a damning critique of FORTRAN and PL/1. He doesn't mince his words when it comes to FORTRAN:

> "The sooner we can forget that FORTRAN ever existed, the better, for as a vehicle of thought it is no longer adequate: it wastes our brainpower, is too risky and therefore too expensive to use. FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands of programmers to our past mistakes."

This might seem mildly amusing and invoke a slightly smug feeling that, as predicted, we got over it and moved on. On the other hand, globally replace "FORTRAN" with "C" in this text, read it again, and you might not be so amused. Writing in 1972, of course, Dijkstra could not have predicted the development of C from BCPL and its eventual and pernicious influence over programming language design.

The lecture contains at least four more highlights that I'd like to emphasize:

Quality is Free. If this sounds familiar, then it probably is. "Quality is Free" is the title of a book on engineering quality management by Philip Crosby from 1977. Dijkstra beats Crosby (and the entire Lean engineering movement) to the punch:

> "Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper."

Note that he says "avoiding" not "detecting" putting the emphasis on *prevention* of defects, not just smarter ways to find them after the event.

Dijkstra wants a theorem prover. He notes that "A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program." (Echoes of Turing's "appropriate discipline" here). He goes on:

> "These rules are of two kinds…Those of the first kind are easily imposed mechanically, viz, by a suitably chosen programming language. For those of the second kind...it seems to need some sort of automatic theorem prover for which I have no existence proof."

I wonder what Dijkstra would make of today's tools like SPARK 2014 (SPARK 2017), Frama-C, the Eiffel Verification Environment, Dafny, Coq, Isabelle/HOL and many more?

Constructive Verification. Dijkstra warns against trying to construct a proof of a program after the program is developed, tested and declared "finished." Instead, he recommends:

"On the contrary: the programmer should let the correctness proof and the program grow hand in hand."

Our experience with SPARK over years supports this observation. My colleagues at Praxis and Altran UK have long advocated for a constructive verification approach, but focused (somewhat pragmatically) on proof of key properties (such as "type safety") rather than complete proof of all functional behavior. Secondly, we have seen spectacular failures of the alternative "retrospective" mode of verification, where unsound, incomplete and annoyingly slow static verification remains the norm.

The Call for Humility. Finally, Dijkstra has strong words for the "Hero Programmers" and their "clever tricks":

"The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague."

2.3 Humphrey (1994)

In 2004, I was fortunate enough to attend Personal Software Process (PSP) training at the SEI in Pittsburgh, taught by Watts Humphrey – the founder of the Software Process research group at the SEI.

PSP had caught our attention through a US Department of Homeland Security funded study in secure software development. The eventual report cited both the "Correctness by Construction" (CbyC) approach of Praxis, and the PSP. Both approaches were reporting delivered defect densities in the vicinity of 0.1 defects per thousand logical lines of code, but with noticeably different approaches. CbyC had a technical emphasis on formal methods and the use of strong, automated verification and disciplined use of languages like SPARK. PSP, on the other hand, appeared to be entirely technology-neutral and placed a firm emphasis on personal discipline. The obvious question was: What would happen if we put CbyC and PSP together? Would we get another order of magnitude improvement? Fortunately, the SEI team were equally curious about CbyC, so a "training swap" was arranged, where I trained in PSP, and Watts and some of his colleagues came over to Bath to attend the SPARK course.

The PSP training involves learning a strict measurement regime to assess your own performance, based on the observation that hardly any programmers know how to measure themselves, and have no idea of how much time they are actually wasting on defect detection and repair. The course involves writing eight programs, adding disciplines as you go along, such as measuring, estimating, and personal design and code reviews. Program 1 is critical – the process required is "Whatever you normally do, but measuring time and defects." This forms a baseline for your own performance – essentially Program 1 acts as a control experiment against which your later performance on the course can be compared.

Of course, I used SPARK to write my programs, since using SPARK (and all the verification tools) *was normal* for me working on a critical software project. My lasting impressions of the course are as follows:

- It's the best training I've ever had in software engineering, but also (more generally) in "personal quality management" – how to get my own stuff right. Worryingly, this came nearly 10 years after joining Praxis/Altran, and I had never been taught anything like this at university.
- Almost all the "software engineers" on my training course were unaware of basic statistics, me included. The measurement and estimating system embodied in PSP requires a basic understanding of statistics (linear correlation, prediction internals, statistical significance and so on), and the programming exercises involve re-implementing these functions from scratch, so you end up *really* understanding them.
- Everyone improves. Even the "gurus" and "hero programmers" have room for improvement, once they discover their most wasteful practices.
- PSP places great (and very effective) emphasis on human-driven personal review of designs and code, using personally generated checklists, but the SEI team had greatly undervalued the potential for automated static verification tools like SPARK. Their impression was formed from experience with retrospective tools like "lint" which exhibit a worryingly high rate of false-negatives (where a true defect is missed and goes unreported) and false-positives (where a possible defect is reported when, in fact, none exists.) The SEI team decided that such tools were not trustworthy. They had never seen anything like SPARK – a system with an unflinching goal of soundness in all things. I ended up re-designing my own personal process, placing the use of the SPARK tools *before* personal review, and adjusting later activities and checklists to match.

Watts' later book on the PSP (Humphrey 2005) is sub-titled "A Self-Improvement Process for Software Engineers."[2] It includes a notable paragraph, coming at the end of section 8.3 on the economics of software quality:

> "The PSP data show that, for every defect type and every language measured, defect-repair costs are highest in testing and during customer use. Anyone who seeks to reduce development cost or time must focus on preventing or removing every possible defect before they start testing."

This repeats what Dijkstra and Crosby had to say in the 1970s – Quality is Free – except that the PSP team have data from over 1000 engineers taking the PSP training to back up the point.

[2] The earlier 1994 edition was sub-titled "A Discipline for Software Engineering" – there's that "d word" again...

2.4 The D word is?

The “D Word” is “Discipline”. In talking about SPARK and PSP for more than ten years, I have noticed a pronounced negative reaction to its use. The word has 12 definitions (9 nouns and 3 verbs) on dictionary.com so is rather unfortunately overloaded. Compare two of the noun forms:

- “Punishment inflicted by way of correction and training.”
- “A set or system of rules and regulations.”

To reduce confusion, I have taken to referring to the first “being hit with a stick” connotation as “Discipline with a Big D” and the second (rather more positive) definition as “Discipline with a little d”. I can’t help noticing that Turing, Dijkstra and Humphrey all use the word with the second meaning, hoping that programmers will follow a well-defined “system of rules” rather than being repeatedly punished for their bugs.

3 The Most Basic Discipline

In trying to persuade customers to adopt systems like CbyC and SPARK, the absence of one crucial discipline in software is worryingly common: individuals and teams don’t (really) measure their performance. The sales meeting usually goes like this:

Customer: “OK... if we spend £X on SPARK tools and training, how much money will we save?”
Me: “You tell me – what’s your pre-test defect density for your last project?”
Customer: “Errm... <silence>”
Me: “OK...how much money are you wasting on defects discovered in test?”
Customer: “<more silence...>”
Me: “In which process phase are you introducing the most expensive defects?”
Customer: “<awkward silence...>

And so on... Without measurement of time (and therefore money) wasted and basic data about defects, there is little chance of improvement, especially if real money has to be invested “up front” to make it happen.

The PSP measurement regime is remarkably simple. The key points are accounting for time as “productive” or “waste” rigorously, and the classification of defects according to their *point of introduction*. The latter information tells you where to go back and fix the process to prevent the defect happening again. Similarities with Toyota’s Lean Production System are not a fluke.

In the PSP training, Watts illustrated the point with a sporting metaphor: “Imagine you’re a 100m sprinter training for the Olympics, but you haven’t got a tape measure or a stopwatch. How well do you think you’ll do?”

4 Why bother with Formal?

Having established the need for discipline and measurement in software development, let’s return to the “F Word” in all its glory. A simple question: Why bother with formal methods at all?

Experience from a long line of “Formal” projects at Praxis and Altran, including SHOLIS (King et al. 2000), the MULTOS CA (Chapman and Hall 2002), and NATS iFACTS (Chapman and Schanda 2014) suggests that the central advantage lies in the *thought process* and *communication* that are facilitated by formal methods, not the technology or “proving” activities at all.

Take, for example, the process of constructing a formal functional specification for a non-trivial system. The very act of *trying* to write it down formally causes you to get stuck and ask hard questions. This is not surprising – in going from informal (natural language and people) to a formal notation, it’s no wonder that we find flaws in the source material. Most commonly, formalization reveals:

Ambiguity. You realize that some natural language text or spoken statement has more than one potential meaning and (more often than not) different customers and stakeholders have firm beliefs about which potential meaning is “the right one”. Conflict management skills are essential here.

Contradiction. Usually easy to resolve. One user says X, while another says Y and they can’t both be true.

Incompleteness. You find a “gap” in the natural language requirements, where the behaviour of the system is undefined. Formalization is good at spotting these, since we aim for a “complete” specification that covers all possible system states and (sequences of) inputs. Enormous effort gets spent here in resolving error handling cases. This is something of a hot topic, since almost all security-related “attacks” on computer systems are related to lack of specification of error handling behaviour for ill-formed input data.

Resolution of these issues requires *talking to the customer*, not just a masochistic exercise in writing lots of mathematics. There are other important benefits in using formal notations, such as programming languages:

Semantic consistency. If a formal language only has exactly one meaning, then you’d expect that one meaning to be apparent to the person that wrote the code, a person reviewing it, *all* verification tools, *all* compilers and target machines, and the poor person that has to maintain it in 10 years’ time. The “one meaning” property also enables a rational market for verification tools, since they should never contradict, and should only differ in their false-positive rate for verification on non-trivial properties. This provides a basis for rational comparison and selection of tools, and the application of diverse tools where appropriate. Compare this dream with the current state of verification tools for the (well-meaning but ambiguous) MISRA C (MISRA 2013) guidelines.

Longevity. A knock-on benefit of semantic consistency. Many of our customers build long-lived system that need to work and be maintained for decades. Some of these systems undergo “mid-life upgrade” (after only 10 years or so) where the target hardware might be refreshed or completely changed, which often infers a change of compiler technology as well. Ambiguous languages are dangerous in this situation: at the point of first construction, undefined and unspecified behaviour on software can go unnoticed since it “seems to work”. These issues only come to light ten years later, when a compiler change reveals differences in behaviour. Someone has to sort this all out, but the person that wrote the code has long since retired. With properly formal languages, this doesn’t occur - programs and specifications have the same meaning regardless of target-specific details.

5 Observations from Fumble Programming

If it wasn’t obvious by now, “Fumble” is short for “Formal and Humble”. I’d like to think that the two concepts are complementary. Formal verification makes you humble by pointing out how many mistakes you make. Through years of such “Fumble Programming”, I’d like to make a few more observations on its advantages and effects.

5.1 Soundness in Static Verification.

If a static verification tool is to be *sound* then it needs to know *exactly* what a program means, with no “guessing”, and analyses based on a reasonable and verifiable set of assumptions. Unfortunately, the designers of our popular imperative programming languages failed to heed Turing’s advice, bringing us languages riddled with “undefined” and “unspecified” behaviours in the interests of portability, expressiveness and execution speed. The incentive for the retrospective tool vendor is skewed to support this situation - to successfully sell a tool to lots of customers, you need a tool that can analyse any code at all, no matter how awful and undefined. The downside is that these tools, upon encountering an undefined behaviour, are doomed to be unsound. The founders of Coverity make a compelling case (Bessey et al 2010) in favour of unsound “best effort” analysis.

SPARK, on the other hand, was something of a rogue entity, aiming for soundness in verification as the primary, non-negotiable design goal. Soundness (once you get it) is attractive – it builds trust in tools, and allows developers to reduce or entirely eliminate verification activities (i.e. do much less testing) for defect classes that are entirely eliminated by analysis.

5.2 Programmer behaviour

At first, programmers tend to feel threatened by SPARK and are perhaps put off by the pedantry of the verification tools. It seems that every line of code is rejected by the tools for some reason or other, at least until programmers learn the rules and idioms of the language. This can be a humbling experience. Eventually, programmers learn to "beat the tool" by designing code that will be amenable to analysis, and will "prove OK" first time. This process also builds trust in the technology in the long run.

A final example of how formal analysis modifies programmers' behaviour. In the PSP training, one of the programming exercises requires the calculation of the standard linear regression coefficients for two sets of points. The problem contains a rather obvious (or so I thought) numerical division operation *in the specification itself.* Imagine each pair of points (x_i, y_i) plotted on a simple 2D graph. When all the points have the same X coordinate, the slope of the best-fit linear regression line is "infinity", and it never intercepts the Y axis at all. This issue exhibits itself as a potential for division-by-zero in the calculation of the slope. The actual formula specified is:

$$\beta_1 = \frac{\left(\sum_{i=1}^{n} x_i y_i\right) - \left(n x_{avg} y_{avg}\right)}{\left(\sum_{i=1}^{n} x_i^2\right) - \left(n x_{avg}^2\right)}$$

You can see that if all the x_i values are the same, then x_{avg} (the average value of all the x_i values) is equal to x_i, and the bottom line evaluates to zero.

In teaching PSP at Altran, we observed three distinct behavioural patterns with this issue:

- Group 1 – not using SPARK. Typically, engineers in this group wrote the code but did not produce a test case for the "division by zero" case, and didn't write defensive code to check it.
- Group 2 – using SPARK, but inexperienced. This group wrote the code and the SPARK toolset reliably pointed out the potential division-by-zero error. Programmers in this group logged this as a defect, added the necessary defensive check and a test case, and modified their personal code review checklist to look out for this kind of thing in future.

- Group 3 – Experienced using SPARK. This group saw the potential division by zero *in the specification* and designed a test case for it, and produced the correct defensive code first time with no defects.

Group 3 have internalized the tool's pedantry into their own discipline and have learnt to "beat the tool" by seeing the potential defect from the outset.

5.3 A Bit of Formal does you Good

Dijkstra dreamt that we would see programs that were "virtually free of bugs" by the end of the 1970s. In retrospect, it seems this was a little optimistic, and perhaps over-sold the potential of formal verification. It also leaves the impression that you *have to prove everything* about a program or not at all.

Our experience suggests that a more gradual adoption has huge benefits. In particular, SPARK was designed to support proof of "key properties" where full proof of functional behaviour is neither possible nor appropriate. The depth of the proof can be selected on a module-by-module basis, and systems can be designed with this idea in mind, so that key critical functions are isolated in a small number of modules.

One "key property" that can be verified is the absence of so-called "runtime errors." Programming language designers like to call this "type and memory safety", but it basically means that your program never crashes or raises a predefined exception following the failure of a runtime "type check" such as buffer overflow, division-by-zero, arithmetic overflow, memory exhaustion, and so on. It's also important to prove this property, since the failure of a runtime check in some non-critical piece of code can cause the entire program to terminate, thus denying the critical code the chance to run at all.

Proof of type safety is a non-trivial verification task, requiring use of techniques such as abstract interpretation or theorem proving, and so is bound to give rise to false positives – where the theorem prover says "don't know". Users of static analysis tools like to complain about false-alarm rate, but my colleagues at Altran decided to invert the problem, setting a target rate for developers to hit. This is easy to measure, since counting verification conditions (VCs) is easily automated: if the tool generates 100 VCs and 99 prove automatically, then your false alarm rate is 1%. We have found this measure to be an excellent proxy for overall code quality.

This led to an observation: programs which exhibit fully automated type safety proof (i.e. no false alarms at all) are highly likely to be functionally correct as well. How can this be so? While we do not have decisive empirical data to back this up, it seems two effects are at play. First, functional correctness bugs often exhibit themselves as type safety bugs, so fixing the latter also fixes the former en-passant without you noticing. Secondly, programs that exhibit a zero false positive rate for type safety have to be *simple* – employing programming idioms that do not defy the capabilities of the analysis tools. In striving the produce programs of such simplicity, our developers produce correct programs. In short: the theorem prover rewards humility and punishes[3] "clever tricks."

These ideas were put to the test in the NATS iFACTS project. An operational release of about 250ksloc generates about 150,000 verification conditions for type safety (Chapman and Schanda 2014), with the proof being fully automated. In making changes to the code developers are in fully "Constructive Mode" – making sure that the proof is maintained *before* committing their code to the configuration management system.

6 A Fumble Future

So, who will win? The hero programmers with their clever tricks, or the fumblers with their "tape measures", "stopwatches" and sound verification tools? Pragmatically, I think the only rational answer is "Hopefully both". For code that is unimportant or going to be thrown away in 6 months, a "quick and dirty" solution might be fine. For code that needs to do something seriously critical for a decade or more, I hope a more formal and measured approach would be chosen.

Acknowledgments Thanks to Mike Parsons, Martyn Thomas, and Jim Over for their comments on an early draft of this paper.

References

Bessey A., Block K., Chelf B et al (2010), A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. Communications of the ACM Vol 53 (2), pp. 66-75.https://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext, Accessed 29th September 2017.

Chapman R. and Hall A (2002), Correctness by Construction: Building a Commercial Secure System. IEEE Software. Vol 19(1), Jan/Feb 2002. pp. 18-15. doi: 10.1109/52.976937

Chapman R. and Schanda F (2014), Are we there yet? 20 years of industrial theorem proving with SPARK. *Invited Keynote Paper*, Proceedings of Interactive Theorem Proving (ITP) 2014. Springer-Verlag LNCS Vol. 8558, pp. 17-26. https://proteancode.com/wp-content/uploads/2015/05/keynote.pdf Accessed 29th September 2017.

Dijkstra E (1972) The Humble Programmer. Turing Award Lecture, 1972. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF Accessed 29th September 2017.

Humphrey W (2005) PSP: A Self-Improvement Process for Software Engineers. Addison Wesley, 2006. ISBN 0-321-30549-3.

King S., Hammond J., Chapman R et al (2000), Is proof more cost-effective than testing? IEEE Trans Software Eng, vol. 26, no. 8, Aug 2000, pp. 675–686, doi: 10.1109/32.879807

MISRA (2013), Guidelines for the Use of C Language in Critical Systems. ISBN 978-1-906400-10-1. March 2013.

SPARK (2017), SPARK 2014 Community Site. http://www.spark-2014.org/about Accessed 13th October 2017.

[3] Shades of "Big D" here perhaps?

Turing A (1947) Lecture to the London Mathematical Society on the Design of the Automatic Computing Engine, 20th February 1949. Turing Digital Archive AMT/B/1. http://www.turingarchive.org/browse.php/B/1 Accessed 29th September 2017.