The Fumble Programmer

Roderick Chapman,
Director, Protean Code Limited
Honorary Visiting Professor, Dept. of Computer Science, University of York
Contents

• The Humble Programmers
• The most basic discipline
• Why bother with Formal?
• A Fumble Future?
• In conclusion
• Homework...
The Humble Programmers...
The Humble Programmers...
Turing’s Lecture to the London Mathematical Society

- Delivered 20\textsuperscript{th} February 1947

- Mainly concerns the design and operation of the “Automatic Computing Engine (ACE)”

- Covers
  - Memory (Mercury delay lines)
  - Control (conditional loops)
  - Arithmetic (full adder design)
  - I/O (cards)
  - Programming
Turing on programming...

“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... *Invariant assertions* are a good idea!
Turing on programming…

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

One of our difficulties will be the maintenance of an appropriate discipline, so that we do not lose track of what we are doing.”
Turing on “Hero Programmers” and losing your job to automation…

• Turing predicts that any programming will be automated by the computer “as soon as any technique becomes at all stereotyped…” but notes that “The Masters” won’t like this at all…

“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.”
Turing on languages…

“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.”
The Humble Programmers...
25 years later...

The Humble Programmer.

by

Edsger W. Dijkstra
The Humble Programmer

- Worth re-reading every few years to see how much has or hasn’t changed…

- Contains a hilarious critique of both FORTRAN and PL/1…

- But…
The Humble Programmer

• Highlight 1 – Dijkstra beats Crosby by 5 years to realize that “Quality is Free”

“Those who want really reliable software will discover that they must find means of avoiding the majority of bugs start with, and as a result the programming process will become cheaper.”
The Humble Programmer

- Highlight 2 – Dijkstra wants a theorem prover

“A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. These rules are of two kinds…

Those of the first kind are easily imposed mechanically, viz, by a suitably chosen programming language.”
The Humble Programmer

• Highlight 2 – Dijkstra wants a theorem prover

“For those of the second kind...it seems to need some sort of automatic theorem prover for which I have no existence proof.”

• But it’s 2018 now... haven’t we got one of those?!?!
The Humble Programmer

• Highlight 3 – The famous quote about testing and the presence of bugs, not their absence...
The Humble Programmer

• Highlight 4a – Realization that retrospective verification is not a good idea…

“But one should not first make the program and then prove its correctness, because then the requirements of providing the proof would only increase the poor programmer’s burden.”
The Humble Programmer

- Highlight 4b – Constructive Verification…

“On the contrary: the programmer should let the correctness proof and the program grow hand in hand.”
The Humble Programmer

• Highlight 5 - the call for Humble

“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.”
The Humble Programmers...
Another 22 years later... 1994...
Quality and Cost in PSP...

“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.” (PSP Book, section 8.3)

Sounds familiar???
One word keeps cropping up...

Turing: “...an appropriate discipline…”

Dijkstra: “A Discipline of Programming”
(the book)

Humphrey: “A Discipline for Software Engineering”
Big or Little “D”?

• “Discipline” is unfortunately overloaded in English... 9 definitions for the noun and 3 for the verb on dictionary.com

• I prefer to differentiate:
  
  • “Big D” Discipline – being hit with a stick...
  
  • “Little d” discipline – repeatable method, organization etc. at both personal and team levels.
The most basic discipline?

Measurement

Watts Humphrey again: “Imagine you’re a sprinter training for the Olympics...

But you haven’t got a tape measure or a stopwatch...

How would you do?”
The most basic discipline?

• Just 3 basic measures are needed:
  
  • Time in *on-task effort*, but keep track of productive time versus waste.
  
  • Size (of the artefact – for estimating)
  
  • Defects (including time wasted to fix, process phase introduced etc. to drive root-cause analysis and improvement.)
Golden rule 1

No measurement implies no improvement.

Engineer: “We should spend £50k on a new static analysis tool.”
Project Manager: “How much money will we save?”
Engineer: “Err...dunno...”
<end of conversation>
What happens when you do measure?

- Key discovery from PSP Training...

- I make lots more mistakes than I thought...

- Most “programmers” are critically unaware of how talented they aren’t.

- The current fad for Test-First or Test-Driven development provides confirmation bias...
Confirmation bias?

I’m a really great programmer, honest…

My new code passes a bunch of test cases…

I was right! I must be a super-ninja black-belt rock-star programmer! Off to the pub!
Observation...

- I am *hopeless* at producing code without:
  
  - Defect-resistant programming languages...

  - *Sound* static verification tools... So you prevent *all* the bugs, not just *some of ‘em*...

- To get most bang-for-the-buck, you need *formal* and *unambiguous* languages.
The dreaded “F Word…”

- At Altran UK, my colleagues have used Formal languages and analyses for many years:
  - In specification: e.g. Z, CSP, SCADE etc.
  - In “code”: e.g. SPARK Ada subset + tools.
- What happens when you “Go Formal”? Is it worth it?
Why bother with Formal?

\[ \Delta \text{IDStation}; \Delta \text{RealWorld} \mid \]
\[ \text{TISOpThenUpdate} \]
\[ \Delta \text{line} = \text{locked} \land \text{latch'} = \text{unlocked} \]
\[ \neg \]
\[ (\exists \text{ValidToken} \land \text{goodT}(\theta \text{ValidToken}) \land \text{currentUserTok} \land \text{UserTooOKNoCurrentCheck} \land \text{FingerOK}) \]
\[ \lor \]
\[ (\exists \text{TokenWithValidAuth} \land \text{goodT}(\theta \text{TokenWithValidAuth}) \land \text{UserTokenOKNoCurrencyCheck}) \]
\[ \lor \]
\[ (\exists \text{ValidToken} \land \text{goodT}(\theta \text{ValidToken}) \land \text{UserTokenOKAuthCertNoCurrencyCheck}) \]

> See: TISOpThenUpdate (p. 5), UserTokenOKNoCurrencyCheck, UserTokenWithOKAuthCertNoCurrencyCheck (p. 5)
Why bother with Formal?
Thinking and Tooling exposes...

Ambiguity...
Thinking and Tooling exposes...

Contradiction...
Thinking and Tooling exposes...

Incompleteness...
Thinking and Tooling exposes...

#include <customer_conversation.h>
Formal notations...

• ...exhibit semantic consistency.

• In short, my “code” means exactly the same thing to:
  • All compilers
  • All “target” machines
  • All verification tools
  • The person that wrote it...
  • The person that reviews it...
  • The person that has to maintain it in N years time...
Semantic consistency...  
A side-benefit

- If my “Code” has exactly one meaning, then *all* verification tools should give the same results, right?

  - Well... nearly... False-positives will be different...

  - Some tools will be “better” than others at detecting certain defects.

  - Enables a rational market for *diverse* verification tools.

- Counter-example: try to get consistent results from 2 or more MISRA C tools... See what happens!
And finally...
Formal notations have *longevity*...

- If my “Code” has the same meaning in twenty years time as it does now.

- I can change compiler and target machine and *it should just work*...

- This happens with military and nuclear systems, for example... “mid life upgrade” etc.
And finally finally...

Formal notations enable **Soundness**

- If you know exactly what a “program” means, the static verification can reasonably be expected to exhibit a *false-negative* rate of Zero.

- Sound tool: “There are *no* bugs”

- Unsound tool: “I’ve tried my best and I can’t find *any more* bugs…”

- Which one would you prefer?
Does Soundness matter?

• “Soundness doesn’t matter”
  • Who says...
  • Err...Tools vendors with unsound tools...

• The market has spoken...

• The market is *broken*?
Using sound verification tools...

Theorem prover says “no”

SAT solver says “counter-example”
Using sound verification tools...

• The first few months can be pretty humbling...

• Everything I do is wrong! 😞

• But you get used to it...

• You learn to beat the tools...

• You learn *trust* the tools...
Do you trust a tool?

• Trust in verification tools is hard-won and easily-lost.

• Problem: almost all the “big name” static verification tools are blatantly (and some...honestly) unsound for verification of non-trivial properties...

• Why?

• See Coverity’s “Billion lines of code later...” article in CACM 2010. In short: the market decided...
Do you trust a tool?

- Time for “soundness cases”???

- Tool vendor presents:
  - “Soundness case...” (i.e. “Why you should trust us...”)
    - Defect history and corrective actions...
    - Reference to underlying maths
    - Lack of counter-evidence.
A social proof for Soundness?

• If “Formal Proof” of tool soundness is too hard, can we find a “Social Proof”???

• Possibly...

• Question: if tools says “No defects of class X”, do you still look for X in any later verification activity (like review and testing)? Do you ever find a class X defect?

• No? Congratulations... You are treating the tool “as-if sound.”

• Problem: convince your customer and regulator that the tool’s evidence is trustworthy...
A Fumble Future?

- Fumble = “Formal and Humble…”
- An Observation...
- A Call to Arms...
A Fumble Future?

- Observation:

- Google for “Formal Verification” – what do you notice?

- ”Formal” is now regarded as standard practice in the design of hardware, especially in the design of silicon SoCs, VLSI, FPGAs etc...

- How come?
A Fumble Future?

- What about software?

- We need formal notations, with *unambiguous* semantics
  
  - Turing’s advice was ignored... 😞

- Main offender: BCPL and all its offspring

- Signs of hope: SCADE, SPARK Ada, Eiffel, CakeML, Cryptol...
A Fumble Future?

- A Fumble vision

- As engineers, we need to be trained to recognize the serious limitations of our own talent.

- My ideal: PSP training becomes normal in all UK universities at undergraduate level.

- (Submit your PSP data for interview please... 😊)
In Conclusion...

• Formal + Humble can be done... And it works...

• Formality in verification makes you Humble...

  “It’s like Jazz – hard at first, but worth it in the long run...”
  Peter Amey, SPARK Team.

• Formality makes you better.
Homework (1)

• If you’re a “Programmer”

• Read Watts’ Humphrey’s PSP Book (10 marks)

• As above, but actually do the programming exercises as well, and measure your performance. (50 marks)

• I will grade your answers!
Homework (2)