Autom Softw Eng (2007) 14: 465–470
DOI 10.1007/s10515-007-0016-2

Desert island column

Keith Gallagher

Published online: 8 September 2007

I have recently moved across the ocean, so I have had, in a manner of speaking, a desert island experience. I had to clean out my personal professional library in the States and decide what books to ship and those to leave behind. So my shelf in Durham gives me a staring point for this exercise. The hard question becomes: Out of the 30 or so books that I shipped here, which are the 5 that I can't live without?

But first, what have I already left behind? Programming and data structures, "computing for the masses", compilers, operating systems. There is an exception noted below. I even left behind most of my software engineering text books.

It turned out that for each of my desert island selections I was forced to make a choice; so I will explain why I am taking what I take, and why I am leaving what I leave. My selections are filled with quotes, which is my attempt to give a taste of what I find interesting and challenging about my selections.

A partial explanation of my madness is in order. While at university, I toyed with undergraduate majors of physics, English, religion, psychology and mechanical engineering. My undergraduate degree is a Bachelor of *Arts* in Mathematics. I started the process of becoming a certified teacher, but didn't finish. I was first attracted to computing by studying mathematical and computational linguistics, that muddy ground between formalism, computers and the mind. Now I find myself doing "applied social psychology"; i.e., software engineering.

I will never write a software engineering text, but I do have a title: "Software Engineering: Learning to Play Together and Putting Your Toys Back". Software engineering, as I see it, is about human communication and cooperation. Playing together is working for a common goal. Putting your toys back is the bit of organization that

K. Gallagher (✉)
Centre of Software Maintenance and Evolution, Computer Science Department, Durham University, South Road, Durham DH1 3LE, UK
e-mail: k.b.gallagher@durham.ac.uk

every project needs. Software engineering is about people and communication; it's not about technology.

I broke the exercise into five sub-problems, based loosely on "a day in the life": arise, morning, early afternoon, late afternoon, evening.

Arise Jerry Harvey writes "Truthful communication enhances our spirits and . . . lies and miscommunication either bruise or destroy our souls." So I'm taking Harvey's "How Come Every Time I Get Stabbed in the Back My Fingerprints Are on the Knife?" (Harvey 1999). A professor of Management Science at George Washington University, Harvey defines "cheating" to his students thus: the failure to assist others if they request it. Then he goes on:

> During formal examinations, if a teacher catches students helping one another, what do we generally call the students' behavior? Most who have participated in any formal educational program will say 'cheating'. In fact, virtually all formal educational institutions, particularly those known for their honor codes, say that giving and receiving aid on examinations is an unequivocal example of nefarious dishonesty, called cheating. Do you find it peculiar that most of us in educational leadership roles believe that it is immoral for students to help one another during a time of need? Alternatively, do you find it puzzling that educational leaders, as a matter of institutional policy, require that their students be unhelpful, self-centered, narcissistic, and selfish, and then assert, with absolutely straight faces, that such behavior is a badge of honor?

I, for one, do find it puzzling. Such concepts regarding the giving and receiving of aid make teaching software engineering all the more difficult. One of the hardest parts of teaching software engineering is conveying the fact the project matters more than anything else; the group sinks or swims together. In all previous courses, students working together on a programming project is strictly verboten, unless authorized. Can you imagine a software engineering project where the group members *didn't* help each other during a time of need, and considered not helping the right thing to do? In software engineering project courses, *the project is the examination*.

(If the above quote doesn't whet your appetite and give you a good idea of why I'm taking this book, try this chapter title: "The Spin Doctors: An Invitation to Meditate on the Organizational Dynamics of the Last Supper and Why Judas was Not the Traitor". It's not anti-religious or even anti-Christian, but it will make you think. "Whether it happened or not I do not know; but you can see that it is true".)

I left behind Brooks' "Mythical Man-Month", because, at the end of the day, Professor Brooks' missive is about communication. In that regard, Harvey has the non-software-engineering version of the "Mythical Man-Month". Brooks tells me about communication in software engineering projects and lets me infer that the techniques apply to all human relations. Harvey tells me about human relations and lets me apply it to software engineering.

Morning Modern punditry claims that doing crossword puzzles and the like staves off Alzheimer's and other brain diseases, so to that end and to keep the spirit of the game I will take Adam Barr's "Find the Bug: A Book of Incorrect Programs" (Barr

2005). In this charming little tome there are 50 errant programs in C Java, Python, Perl and x86. To me, this would be the software engineering equivalent of taking my book of Sunday New York Times crossword puzzles. Even to re-read it and do the puzzles again would be stimulating, for who hasn't looked at an errant program and said "I've seen this before; if I could only remember what I did then???"

I was sorely tempted to take Alan Feuer's "C Puzzle Book", but I decided against it because Feuer's book is about one language. The goal of each puzzle is to demystify some C arcana, and each program, although a puzzle, is correct. I study software maintenance and evolution, so errors are an important part of my life! I also left behind James Adams' "Conceptual Blockbusting: A Guide to Better Ideas". It might come in handy on a desert island, but it's not really about software engineering.

Early Afternoon I'm taking Harlan Mills' collected anthology "Software Productivity" (Mills 1983). This little volume is amazing. I don't think I have *ever* read something in software engineering that wasn't alluded to in this book: clear mathematical thinking for programmers; group organization; proofs of correctness; statistical validation; buying software; education; enterprise computing. Pick a modern hot software engineering topic and you will find nascent ideas in Mills' writing. Mills' work is like Harvey's; he lays foundations and opens his thinking process, so that you can see how to adapt the ideas to your own problem.

My mathematical inclinations are verbalized by Mills:

> The idea of mathematics is to make life easier, to find simpler ways of doing things. A mathematics theorem is elegant. . .because it says more with less wasted motion. . . . We need this kind of power in computer programming to handle more detail with less effort. However, it is easy to mix up the simplicity that comes from a deep analysis with a simple-minded analysis, which leads to hopeless complexities. Finding the key simplicities . . . is a deep problem not often resolved by a simple-minded approach.

I haven't always followed this advice.

I left behind the text that he wrote with Vic Basili, John Gannon and Dick Hamlet. It has most of these ideas in an introductory programming text—the only programming text book I brought across the ocean.

Late Afternoon I'm taking C.P. Snow's "Two Cultures and a Second Look" (Snow 1969). The inspiring quote:

> A good many times I have been present at gatherings of people who, by the standards of the traditional culture, are thought highly educated and who have with considerable gusto been expressing their incredulity at the illiteracy of scientists. Once or twice I have been provoked and have asked the company how many of them could describe the Second Law of Thermodynamics, the law of entropy. The response was cold: it was also negative. Yet I was asking something which is about the scientific equivalent of: 'Have you read a work of Shakespeare's?'
> I now believe that if I had asked an even simpler question—such as, What you mean by mass, or acceleration, which is the scientific equivalent of saying, 'Can

> you read?'—not more than one in ten of the highly educated would have felt that I was speaking the same language. So the great edifice of modern physics goes up, and the majority of the cleverest people in the western world have about as much insight into it as their Neolithic ancestors would have had.

Some modern thinkers, most notably Stephen Jay Gould, have vigorously disagreed with Snow, going so far to argue that his viewpoint is actually destructive. For my part, I have experienced, more times than I care to remember, the coldness that Snow experienced. I was expected to know modern literature, but I was scoffed at when I asked in return who had read prize-winning science writing, and was treated as if reading it was the same as reading bathroom walls. I have been asked in all seriousness by a theologian: "Have you heard of Freud?" My answer was a recitation of the Freud's books that I have read.

Why am I taking this? Writing 50 years ago, many of Snow's predictions about the future state of education (in England) and the first-world response to the plight of the third-world were off the mark. I do like his analysis of intellectuals as natural Luddites, which gently pre-dates Chomsky's assertion that intellectuals are the keepers of the propaganda. It is a continual challenge for me to stay open to new, interesting and even weird ideas, and not immediately dismiss them. My deepest fear is that I will become famous for saying something stupid, like that nameless fellow of yore who remarked "*there is no reason for any individual to have a computer in his home*".

What did I leave? Hardy's "Mathematician's Apology". I first read it as a graduate student, and even started my own apology way back when. (I did write an *apologia* that I submitted with my promotion and tenure application. It was not as long as Hardy's.) After rereading Hardy in preparation for this essay, it now seems a bit narrow. I have also come to disagree with one major premise of the Apology: that good research can't be done after age 40. That might be true in mathematics, but it certainly isn't true in software engineering. (I am not speaking of myself!)

Evening I'm taking Bruce Blum's "Beyond Programming: To a New Era of Design" (Blum 1996).

> This book. . .constructs a normative model for software development- one that describes how software ought to be developed in the future. Given its objective, *Beyond Programming* cannot be a how to book; it must be one that fosters contemplation and enquiry. Therefore, it is appropriate that it begin by asking the most basic of questions. What are software engineering, computer science and—in a general sense—science itself?

Blum's predictions about the future will probably be wrong, much as Snow's predictions about the future were. I even disagree with some of his answers. That's ok, because his book does foster contemplation and enquiry.

I often paraphrase one particular nugget from this book. Software not only changes how I understand the world, it changes the world. If I had continued to study physics, my job would be to find *what is*, to observe phenomena. I would then have to adjust my understanding of reality to the observations. One of the reasons that computer science is not a science is that we do not find what is; we make what we want. Then after we make whatever-we-want, our world and our understanding of the world has

changed. For who, even after getting a piece of software that precisely meets its specifications, doesn't want more? The fact that we have what we want (in software) changes what we want (in software). We have changed our world, not observed it, by creating software. Our changed world changes our understanding of it. We software types adjust reality to fit our observations.

I left behind "Classics in Software Engineering", the collection of seminal papers edited by Ed Yourdon. This was a hard choice, the hardest of all. I finally decided on Blum's book because it looks forward and I have the Mills book to keep me grounded in the masters.

Bedtime I know I said I had broken the problem into five pieces—but here is the sixth! As I am permitted only five books, I need to find a way to assist myself in a time of need (is this cheating?) in order to get one more reading. The selection is an article, so my plan is to print it on one sheet (front and back) and use it as a dust jacket for one of the hardbacks! My last choice is the little 5 page charmer of Drew McDermott: "Artificial intelligence meets natural stupidity" (McDermott 1976). Its abstract:

> As a field, artificial intelligence has always been on the border of respectability, and therefore on the border of crack-pottery. Many critics have urged that we are over the border. We have been very defensive toward this charge, drawing ourselves up with dignity when it is made and folding the cloak of Science about us. On the other hand, in private, we have been justifiably proud of our willingness to explore weird ideas, because pursuing them is the only way to make progress.

Its three sections are "Wishful Mnemonics", "Unnatural Language" and "Only a Preliminary Version of the Program was Actually Implemented". The first two sections are about muddled naming and thinking. For example, calling a program "General Problem Solver" when it is actually a "Local-feature-guided Network Searcher" or saying that "Nixon is a Hitler" is like saying "Fido is a dog".

The third section should strike fear into the heart of every software engineering researcher. McDermott says "A common AI idiocy is to suppose that having identified the shortcomings of Version I is equivalent to having written Version II". While McDermott is writing about AI, how many times have we heard (or used?!?!) that phrase in software engineering research? I summarize McDermott. Version I is not finished. It has problems. Perhaps even interesting problems. Unsolved problems. The problems are enumerated. But the report on the work leaves the enticing but invalid conclusion that they have all been solved—when they haven't. This, of course, discourages new students from looking in the area as "the problem has been solved". This scares the bejesus out of me.

There are two reasons for taking this article: stay open to new ideas, even weird ideas, but above all, be honest. I couldn't find any other article that was short enough to be a dust jacket.

So I am off to my desert island. I hope the water is warm, the sharks are well-fed, and that I can take my guitar. Now that I have concluded this marvelous exercise (for which I gratefully acknowledge the editor's gracious invitation) I try to proof read it.

No luck. It does seem, though, that after re-reading this essay that I want to be helpful, able, precise, open, thoughtful and honest. That's not a bad set of requirements for any software engineer!

Goodnight.

References

Barr, A.: Find the Bug: A Book of Incorrect Programs. Addison-Wesley, Reading (2005). ISBN 0321-223918

Blum, B.: Beyond Programming: To a New Era of Design. Oxford University Press, London (1996). ISBN 0-19-509160-4

Harvey, J.: How Come Every Time I Get Stabbed in the Back My Fingerprints Are on the Knife? Josey-Bass, San Francisco (1999). ISBN 0-7879-4787-3

McDermott, D.: Artificial intelligence meets natural stupidity. ACM SIGART Bull. **57**, 4–9 (1976). ISSN 0163-5719

Mills, H.: Software Productivity. Little-Brown, Boston (1983). ISBN 0-316-57388-4

Snow, C.P.: Two Cultures and a Second Look. Cambridge University Press, Cambridge (1969). ISBN 0521-09576-X