Good Old Advice

Dr. Alistair Cockburn
Humans and Technology

From Winston Royce, Fred Brooks, Gerald Weinberg, and the speakers at the 1968 North Atlantic Treaty Organization (NATO) conference on software engineering to Barry Boehm, Victor Basili, Capers Jones, and Tom DeMarco, the best-known writers have been making the same recommendations, which we still ignore today.

"It's new, it's improved, it's old-fashioned," sings Tom Waits in "Step Right Up." How well he captures the evolution of software development!

With the benefit of hindsight, we can see that the best-known writers in the software field have been advocating the same four recommendations written in the agile manifesto for decades (see The Agile Manifesto section).

The older writers were ignored for decades while people searched for mechanical replacements for the key elements in developing software: thinking and communicating. But that's a separate story.

It Is About People

In the legendary "Psychology of Computer Programming," published in 1971, Gerald Weinberg starts his preface with these prescient words, "This book has only one major purpose – to trigger the beginning of a new field of study: computer programming as a human activity ... " [1].

His hope lay dormant for 15 years, until Tom DeMarco and Tom Lister wrote the equally legendary, "Peopleware" [2]; Raymonde Guindon and Bill Curtis wrote their landmark paper on cognitive strategies in design [3]; Turing Award winner Peter Naur wrote about programming as *theory building* [4]; and Fred Brooks wrote the oft-quoted but still not followed, "No Silver Bullet" [5].

People Trump Processes

Those authors established that software development is deeply intertwined with the mental and social processes of the people involved. As Weinberg put it, "I've learned a basic principle of the psychology of management: managers who pay attention to people get good results" [1].

Barry Boehm and Capers Jones captured details of projects over the years and gave us startling numbers. From Boehm's studies come the following [6]:

- Personnel/team capability were the most significant cost drivers, affecting productivity by 353 percent.
- Process maturity was 12th, with a 143 percent effect.

From Capers Jones' studies [7]:

- Upper management and staff experience contribute 120 percent to productivity.
- Effective methods/processes contribute only 35 percent.

Capers Jones also cites the following reverse effect:

- Staff inexperience: negative 177 percent.
- Ineffective methods/processes: negative 41 percent.

In other words, we have long known that success is governed more by who you hire, how they interact with each other, and how you treat them. Yet despite the data and the advice, we keep retreating to the weak hope that some new process, tool, or technique will relieve the need for talent and communication.

People Mean Communication

Once a certain amount of talent is in place, the speed of the project is directly related to the speed at which information moves between minds. Briefly, *anything that slows the movement of ideas between minds slows the project.*

That includes – most particularly – such issues as morale, trust, pride in work, personal safety, and the amount of face-to-face communication available.

Participants at the famous 1968 NATO conference on software engineering said as much. On page 53 of the conference report, J.N. Buxton says:

> We could use more and better and faster communication in a software group as a partial substitute for a science of software production. ... if I'm setting up a software group to carry out a project I'm extremely careful that all the people working on it are close personal friends, because then they will talk together frequently, and there will be strong lines of communication in all directions. [8]

T.J. Allen, of the Sloan School of Management, studied the question of communication in research and development labs [9]. He found that communication dropped steeply when people had to walk more than 10 meters to a colleague's desk – roughly the length of a school bus. Allen constructs what he considers an optimal research and development workspace. It is full of osmotic communication [10], dense conversation and chance overhearing, and strikingly resembles what we now refer to as a *war room.* Allen's results have been reproduced several times over the years by, among others, G. Olson and J. Olson [11].

Physical proximity not only increases the frequency but also the bandwidth of communication. J.C. McCarthy and A.E. Monk [12] detail how proximity provides people additional coordinated communication channels such as facial and hand gestures, vocal inflection, physical distance cues, cross-modality timing, and real-time question and answer.

Old Riddles Solved

This point is illustrated using the following two experiences from the 1960s. They were almost certainly puzzling when they occurred, but with our current understanding of software development, we can now see how these seemingly anomalous stories make good sense.

Experience 1: Gerald Weinberg writes the following:

> [At] a large university computing center ... a large common space was provided near the return window so that the students and other users could work on their programming problems. In the adjoining room, the center provided a consulting service for difficult problems, staffed by two graduate assistants. At one end of the common room was a collection of vending machines ... the noise from the revelers congregating at the machines often became more than some of the workers could bear ... [The computing center manager] went to investigate their complaint ... Without more than 15 seconds of observation, he went back to his office and inaugurated action to have the machines removed to some remote spot. The week after the machines had been removed – and signs urging quiet had been posted all around – the manager

received another delegation ... They had come to complain about the lack of consulting service; and, indeed, when he went to look for himself, he saw two long lines extending out of the consulting room into the common room. He spoke to the consultants to ask them why they were suddenly so slow in servicing their clients ... For some reason, they said, there were just a lot more people needing advice than there used to be ... After some time, he discovered the source of the problem. It was the vending machines! When the vending machines had been in the common room, a large crowd always hovered around them ... they were chatting about their programs. ... Since most of the student problems were similar, the chances were very high that they could find someone who knew what was wrong with their programs right there at the vending machines. Through this informal organization, the formal consulting mechanism was shunted, and its load was reduced to a level it could reasonably handle. [1]

It is to allow for just such serendipitous exchanges that many companies now put coffee machines, whiteboards, and chairs at various places in the buildings.

Experience 2: In the following example, Dee Hock presents us with a much more puzzling scenario with the development of the first Visa credit card clearance system in the 1960s:

We rented cheap space in a suburban building and dispensed with leasehold improvements in favor of medical curtains on rolling frames for the limited spatial separation required ... Swiftly, self-organization emerged. An entire wall became a pinboard with every remaining day calendared across the top. Someone grabbed an unwashed coffee cup and suspended it on a long piece of string pinned to the current date. Every element of work to be done was listed on a scrap of paper with the required completion date and name of the person who had accepted the work. Anyone could revise the elements, adding tasks or revising dates, provided that they coordinated with others affected. Everyone, at any time, could see the picture emerge and evolve. They could see how the whole depended on their work and how their work was connected to every other part of the effort. Groups constantly assembled in front of the board as need and inclination arose, discussing and deciding in continuous flow and then dissolving as needs were met. As each task was completed, its scrap of paper would be removed. Each day, the cup and string moved inexorably ahead. Every day, every scrap of paper that fell behind the grimy string would find an eager group of volunteers to undertake the work required to remove it. To be able to get one's own work done and help another became a sought-after privilege ... Leaders spontaneously emerged and re-emerged, none in control, but all in order. Ingenuity exploded. Individuality and diversity flourished. People astonished themselves at what they could accomplish and were amazed at the suppressed talents that emerged in others. Position became meaningless. Power over others became meaningless ... a community based on purpose, principle, and people arose. Individuality, self-worth, ingenuity, and creativity flourished and as they did, so did the sense of belonging to something larger than self, something beyond immediate gain and monetary gratification. No one ever forgot the joy of bringing to work the wholeness of mind, body, and spirit; discovering in the process that such wholeness is impossible without inseparable connection with the others in the larger purpose of community effort. No one ever replaced the dirty string and no one washed the cup ... The BASE-1 system came up on time, under budget, and exceeded all operating objectives. [13]

At the time, Hock's method must have seemed like madness. With the benefit of 40 more years of investigation, we might say that he had created an early version of Scrum [14].

All these experiences point to one thing: *Attend to the individuals and their interactions.*

It Is About Software

If building software is about the people doing the building, it is also about the software being built. Software has a merciless quality to it that is not present in any requirement or design document. Whether it runs or not depends on exactly what is written, not on how fervently the authors believe it will.

The best known writers in our field have been telling us for decades to *write some software* and then *learn from writing it.*

Winston Royce wrote the much (but falsely) maligned "Managing the Development of Large Software Systems" [15]. Rereading it in the light of modern agile development, we can see it as a model of clarity, not expressing at all what most people think it does.

Two commonly shown models of software development are ascribed to Royce: one with no feedback arrows at all between development stages, the other with feedback arrows from each stage to the previous. *These two pictures were never presented as even faintly possible actual development processes!* They appear only as illustrative stepping stones leading to a carefully drawn iterative model.

More interesting, though, is that after deriving the iterative model halfway through the article, Winston Royce says, "The remainder of this discussion presents five additional features that must be added to this basic approach to eliminate most of the development risks" [15].

Of these five features, fully two of them advise the manager to start programming *before* the analysis is done! The second piece of advice is:

Attempt to do the job twice – the first result provides an early simulation of the final product. Without this simulation the project manager is at the mercy of human judgment. With the simulation, he can at least perform experimental tests of some key hypotheses and scope down what remains for human judgment. [15]

The 1968 NATO conference participants said very much the same thing. A.G. Fraser reported the following:

Design and implementation proceeded in a number of stages ... Each stage produced a useable product and the period between the end of one stage and the start of the next provided the operational experience upon which the next design was based ... The first stage did not terminate with a useable object program but the process of implementation yielded the information that a major design change would result in a superior and less expensive final product. During the

> second stage the entire system was reconstructed; an act that was fully justified by subsequent experience ... The final major design change arose out of observing the slow but steady escalation of complexity in one area of the system. [8]

The participant J. Smith put it more sarcastically:

> I'm still bemused by the way they attempt to build software ... All documents associated with software are classified as engineering drawings. They begin with planning specification, go through functional specifications, implementation specifications, etc., etc. This activity is represented by a PERT chart with many nodes. If you look down the PERT chart you discover that all the nodes on it up until the last one produces nothing but paper. It is unfortunately true that, in my organization, people confuse the menu with the meal. [8]

Victor Basili, with A. Turner, formally proposed an evolutionary coding strategy in their 1975 paper conspicuously titled, "Iterative Enhancement: A Practical Technique for Software Development" (their skeletal subproject is referred to these days as a *walking skeleton* [16]):

> This paper recommends the iterative enhancement technique as a practical means of using a top-down, stepwise refinement approach to software development ... This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successful implementations in order to build the full implementation. [17]

In his famous paper, "No Silver Bullet," Fred Brooks described that technique as *growing* the software:

> I have seen the most dramatic results since I began urging this technique on the project builders in my software engineering laboratory class. Nothing in the past decade has so radically changed my own practice, or its effectiveness ... I find that teams can *grow* much more complex entities in four months than they can *build* ... The morale effects are startling. [5]

Craig Larman has documented the pedigree of such incremental-iterative development extensively [18]. Our industry experts have been recommending it for 40 years – but most project teams still find it a hard sell to their management.

The short form of the recommendation is to *put the mercilessness of running software to good use – write, and learn from the writing.*

It Is About the Users

Starting with Winston Royce again, one of his five steps (including his capital letters) advises: "INVOLVE THE CUSTOMER. For some reason, what a software design is going to do is subject to wide interpretation, even after previous agreement" [15].

Fred Brooks writes:

> I would go a step further and assert that it is really impossible for clients, even working with software engineers, to specify completely, precisely, and correctly the exact requirements of a modern software product before having built and tried some versions of the product they are specifying. [5]

In other words, it does not matter what they told you they want, when they finally see it, they will understand their world differently and the software must change to match that new understanding.

The researchers M. Keil and E. Carmel studied the effects of links to the users. They asked managers who had been on two projects, one successful and one less successful, to compare the forms of access to users. They wrote the following:

> Results of the analysis revealed that in 11 of the 14 paired cases, the more successful project involved a greater number of links than the less successful project ... This difference was found to be statistically significant in a paired t-test ($p<0.01$). [19]

Paying attention to the users goes beyond merely getting it right. Users constitute a powerful lobbying force that can shift a project from irrelevant to top priority or shift the developers from being seen as an enemy force to an ally – to the benefit of both the project and the users.

In other words, *get inside the heads of your users and customers, and get them on your side.*

It Takes (Re)Planning

It would be nice if we could properly plan these projects in advance, but the most recognized writers in our field warn us against this, as Royce says in the following:

> ... [there are] five steps that I feel necessary to transform a risky development process into one that will provide the desired product. I would emphasize that each item costs some additional sum of money. If the relatively simpler process without the five complexities described here would work successfully, then of course the additional money is not well spent. In my experience, however, the simpler method has never worked on large software development efforts and the costs to recover far exceeded those required to finance the five-step process listed. [15]

At the 1968 NATO conference, E. David said:

> Computing has one property, unique, I think, that seriously aggravates the uncertainties associated with software efforts. In computing, the research, development, and production phases are often telescoped into one process. In the competitive rush to make available the latest techniques ... we strive to take great forward leaps across gulfs of unknown width and depth ... Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but also uncalculable risks. [8]

D. Ross added:

> The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. [8]

Fred Brooks puts it more simply: "Plan to throw one away; you will, anyhow" [5].

Even in civil and aeronautical engineering, the sage advice is to plan to replan. Dr. A. Laufer, editor-in-chief emeritus of NASA's Academy Sharing Knowledge e-zine, summarized it: "Plan and control to accommodate change ... Start planning early and employ an evolving planning and control process" [20].

The short summary of the last 40 years of experience is this: *Plan coarse-grained long-term and fine-grained short term, and find a way* *to re-plan quickly – because you'll have to.*

One Process Will Not Fit All

People in our field search continually for the one, true process which, if we could only find it, would serve as the base for every project and also as master training material for newcomers to the field.

In [21], I investigated this question directly. I discovered that processes must change as technologies change, and also change to fit varied types of projects. How many types of projects are there? Capers Jones estimated there to be at least 37,400 different categories of projects [7]. No wonder no single methodology or process will fit them all. *Do not believe in any single process or methodology because each works only in a particular and limited context.*

The Agile Manifesto

In 2001, the authors of the Manifesto for Agile Software Development, made the same recommendations in their own way [22]. They wrote:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value in the items on the right, we value the items on the left more. [22]

The 12th and final principle in the manifesto says:

> At regular intervals, the team reflects on how to become more effective, then tunes and adjusts their behavior accordingly. [22]

Summary

I derived the four values of the Agile Manifesto from much older, recognized, and highly reliable articles. The point I wish to make is that the authors of the Agile Manifesto *did not* throw out all previous experience as they wrote it. On the contrary, what they did was cherry-pick four of the most important issues from among hundreds. The appropriateness of their selection is seconded by decades of prior experience and data, as evidenced by the numerous articles referenced in this article.

For decades, we have learned, documented – and ignored – the same lessons:

- Attend to the individuals and their interactions.
- Put the mercilessness of running software to good use – write, and learn from the writing.
- Get inside the heads of your users and customers, and get them on your side.
- Plan coarse-grained long term and fine-grained short term, and find a way to replan quickly – because you'll have to.

And, finally:

- Do not believe in any single process or methodology because each works only in a particular and limited context.

Those who do not learn from history are doomed to repeat it. Let's stop pretending we don't know everything and let's stop repeating painful history by following this *good, old* advice.◆

References

1. Weinberg, G. The Psychology of Computer Programming. Dorset House, 1998.
2. DeMarco, T., and T. Lister. Peopleware. Dorset House, 1999.
3. Guindon, R., and B. Curtis. Control of Cognitive Processes During Software Design: What Tools Are Needed? Proc. of Chi '88 Conference: Human Factors in Computing Systems. Washington, D.C., 1988.
4. Naur, P. "Programming as Theory Building." Computing: A Human Activity. ACM Press, 1992.
5. Brooks, F. "No Silver Bullet." Computer Apr. 1987: 10-19.
6. Boehm, B. Software Cost Estimation With COCOMO II. Prentice Hall PTR, 2000.
7. Jones, Capers. Software Assessments, Benchmarks and Best Practices. Addison-Wesley, 2000.
8. Naur, P., and B. Randell, eds. "Software Engineering: Report on a Conference Sponsored By the NATO Science Committee." <http://homepages.cs.ncl.ac.uk/brian.randell/NATO>.
9. Allen, T. Managing the Flow of Technology. MIT Press, 1984.
10. Cockburn, A. Agile Software Development: The Cooperative Game. Addison-Wesley, 2006.
11. Olson, G., and J. Olson. "Distance Matters." Human-Computer Interaction. (2001): 139-179.
12. McCarthy, J., and A. Monk. "Channels, Conversation, Cooperation, and Relevance: All You Wanted to Know About Communication But Were Afraid to Ask." Collaborative Computing. Vol. 1, Mar. (1994): 35-61.
13. Hock, D. Birth of the Chaordic Age. Berret-Koehler, 1999.
14. Schwaber, K., and M. Beedle. Agile Software Development With Scrum. Prentice-Hall, 2001.
15. Royce, W. "Managing the Development of Large Software Systems." IEEE Wescon Aug. 1970: 1-9 <www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>.
16. Cockburn, A. Surviving Object-Oriented Projects. Addison-Wesley, 1998.
17. Basili, Victor, and A. Turner. "Iterative Enhancement: A Practical Technique for Software Development." IEEE Transactions on Software Engineering. Dec. 1975: 390-396.
18. Larman, C. Agile and Iterative Development. Addison-Wesley, 2003.
19. Keil, M., and E. Carmel. "Customer-Developer Links." Communications of the ACM. May 1995: 33-44.
20. Laufer, A. "Managing Projects In a Dynamic Environment: Results-Focused Leadership" <http://appl.nasa.gov/ask/issues/19/19o_resources_letterfromtheeditor.php>.
21. Cockburn, A. "People and Methodologies in Software Development." Diss. University of Oslo, 2003 <http://Alistair.Cockburn.us>.
22. "Manifesto for Agile Software Development" <http://agilemanifesto.org>.

About the Author

Alistair Cockburn, Ph.D., is an expert on object-oriented (OO) design, software development methodologies, use cases, and project management. He is the author of "Agile Software Development," "Writing Effective Use Cases," and "Surviving OO Projects," and was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group, served as special advisor to the Central Bank of Norway, and has worked for companies in several countries. More can be found online at <http://alistair.cockburn.us>.

1814 East Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
E-mail: acockburn@aol.com