ThinkingISsues

Tony Clear
School of Computer and Information Sciences
Auckland University of Technology,
Private Bag 92006, Auckland 1020, New Zealand
Tony.Clear@aut.ac.nz

The Waterfall is Dead. Long Live the Waterfall!!

I find myself now in the throes of rewriting the guide I provide to our students undertaking their capstone software development projects. Like many such artefacts it has evolved over time and is an amalgam of my own and prior colleagues thoughts, informed by numerous sources from the literature. But we are now at some form of watershed, experiencing an increasing level of discomfort with the existing guide and the schedule we have applied for assessing students' work.

In our capstone software development project students undertake projects under three broad headings: 1) commercial software development for live client to deliver a working application, 2) R& D project for commercial client to undertake a technology evaluation and deliver a proof-of-concept application, 3) a more theoretical research project developing software within a research team or for a research sponsor.

The table below identifies four broad categories of work (among other criteria) under which students have previously been assessed.

Requirements analysis
Feasibility/Design
Construction
Implementation & testing

Table 1: Assessment Items

Students are required to submit a portfolio providing evidence of their work under each assessment item. So, for instance, to evidence their project management they may include copies of baseline project plans, project plan revisions, progress reports etc. So far so good!

Yet the development process as represented by the categories in table 1, becomes problematic. Inbuilt are the assumptions of linearity and segmentation. These may indeed be reinterpreted on a case by case basis, and tailored deliverables may be identified for each project, but there is an inevitable colouring portrayed by the words themselves. Students seem to revert to a waterfall development approach in order to produce the documentation they interpret as necessary for their assessments.

My colleague Anne Philpott has brought this home to me recently. Anne has been incorporating more agile approaches to software development in the earlier sequence of courses in our undergraduate degree. Initially applied within the software design and implementation course, this has now rippled forward to a review of the content of our software engineering course, and has caused me to rethink our capstone project. Anne's emphasis has been on Highsmiths's [1] interaction, cooperation and collaboration within the software process. Her students have applied various agile methodologies and techniques as discussed in [1] such as pair programming, SCRUM, and feature driven development.

So in rethinking this process, I find myself wrestling with the core distinctions between programming-in-the-small and programming-in-the-large. Key questions such as "what is programming?" come to mind. Is programming "the implementation of a design", as my colleague Bob Roggio has recently suggested? Or is it something else, the core activity of software development, around which a whole series of often confounding models and translation processes have evolved? Then too what is rigour in the software process? The most agile methods such as extreme programming [5] seem to concentrate on the code, the code and nothing but the code. But without supporting documentation to drive the thinking, and communicate the intentions to project stakeholders such as sponsors, users, development colleagues, future maintainers of the software, and operators of the systems, how does this differ from mere software hacking?

There seem to be a few conflicting issues here. The waterfall lifecycle is somehow deeply engrained in developers' psyches. Ambler and Constantine [3] note "that the iterative nature of the [RUP] lifecycle is foreign to many experienced developers, making acceptance of it more difficult".

Even the Object Oriented Software Process advocated in [3] is proposed as "serial in the large, iterative in the small, delivering incremental releases over time". Bruegge and Dutoit [4] likewise advocate very soundly the iterative and incremental nature of development, yet at the core, when their documents and artifacts are scrutinised the waterfall skeleton shows through.

Perhaps the use of the term "construct" is at the core of the issue. Do we really "construct" software, or is this a misplaced metaphor for a disaggregated "coding" stage? In the same way that "software engineering" is a problematic term, have we just borrowed the language of engineers to superimpose the carefully staged framework of bridge building - namely "design, build/construct, maintain" on the software process?

If as argued in [5] "when a development team creates a new system it is actually inventing a new way for people to work", is this as concrete and fixed an outcome as a bridge? And can a new technology supported work process be wholly envisaged from the inception and seamlessly delivered without ever being enacted? I doubt it. And this need to interact with the system, experience the proposed new practices, and comprehend the system's behaviour is the point at which misinterpretations can become apparent, new possibilities can be foreseen, and flaws in the original vision can be comprehended.

So it seem to me that we have a tension between four opposing forces:

- A force for change built upon an initial and evolving vision, which drives the software process
- a commercial force for certainty of cost and outcomes
- a project management force for certainty of delivery against targets
- a professional force for delivering quality software

It is the confluence of these forces and the borrowing of commercial and project management models from the engineering community that have brought us today's methodologies. Yet I do not think we have yet reconciled their inconsistency with the very nature of software, and the requirements of a quality software process. Until we do so, I can see the industry continuing with fictitious project progress reports to keep linear managers happy in the delusion of control of an inherently uncontrollable process.

1. Highsmith, J., *Agile Software Development Ecosystems*. The Agile Software Development Series, ed. A. Cockburn and J. Highsmith. 2002, Boston: Addison-Wesley. 404.
2. Beck, K., *Extreme Programming Explained: Embrace Change*. 2000, Boston: Addison-Wesley.

3. S. Ambler and L. Constantine, *The Unified Process Inception Phase*. Lawrence: CMP Books, 2000.
4. Bruegge, B. and A. Dutoit, *Object Oriented Software Engineering*. 2000, New Jersey: Prentice Hall.
5. H. Beyer and K. Holtzblatt, "Data Based Design," in *The Unified Process Inception Phase*, S. Ambler and L. Constantine, Eds. Lawrence: CMP Books, 2000, pp. 36 - 44.