ThinkingISsues

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

Documentation and Agile Methods: Striking a Balance

I have written previously about the need for students to develop discrimination as part of their preparation for professional practice during their undergraduate capstone courses. But nowhere is this need for discrimination more problematic than in the area of software documentation. Perhaps the only consolation is that professional developers are equally challenged. Yet in migrating students from the set of beliefs and practices that may have worked for them in programming-in-the-small, to those required for programming-in-the-large, sound documentation practices are critical to effective development and delivery of a professional product.

Belief systems related to documentation are intriguing. Within the software development community we see what Highsmith [1] has termed the "battle lines between proponents of agile software development ecosystems (ASDE's) and rigorous system development methodologies (RSM's), based upon fundamentally different assumptions about how the world and organizations work. On the one hand we see the more extreme proponents of agile methods arguing for the code itself as the main artifact and primary source of documentation for the project, and on the other hand the heavily process oriented and documentation driven methodologies of the software engineering camp as outlined in such formal representations as SWEBOK [1], and the SEI's Capability Maturity Model.

Part of the answer lies in the views of Naur who has advanced the notion of programming as "theory building", during which the programming team develops a jointly owned "theory of the world" to become frozen into software. He regards documentation as a secondary construct to the programmers' internalised theory of the program or system, and based upon this "Theory Building View, for the primary activity of the programming there can be no right method"[2], since the creative process of theory building is inherently not a method or rule driven activity. Thus the argument of the agile methodologists [1] for interaction, cooperation and collaboration, during development, rather than communication by formal documents, is given weight by a theory building perspective. For such dynamic interaction and communication is a necessary part of the process of developing the jointly owned 'theory of the world' to be represented in the software artifact arising from the project.

It follows then that documentation is <u>not</u> necessary if the programming team can jointly own and hold the theory of the world in their heads. This of course is the mindset of the novice programmer who sees "the code, the code and nothing but the code" as the key artifact from a software project. Having the likes of Kent Beck [3] advocating extreme programming then, is a great support for students who can code passably but either hate, or are barely able to write a coherent sentence.

Perhaps this also helps explain the programmer mentality about documentation as something external, something "other" than the primary work of coding, since it is only through the coding that the theory becomes encapsulated. For students this view is even more justifiable, given the limited scale of the problems which they encounter as they begin their programming exercises. The inability of a team to completely grasp more complex domains and the issues that arise with increasing the scale of projects, are not apparent to students, and it seems that they have to encounter them for themselves and learn by their own mistakes.

The question that must be answered then, is what is the role of documentation in software development? If students do not see the need for it, why do it? Why should we require it of them?

Ambler [5] suggests two primary reasons for documentation, namely that we should model (or document) to communicate, or model (or document) to understand. In our capstone projects I have advocated a document-driven methodology, in which each artifact builds upon and can be related back to prior artifacts. Like Ambler, I recommend that the documents be produced to support the thinking associated with each stage of the project. In other words producing the document develops and supports the understanding, so that in effect the writing 'writes' the thinking for the next steps. The resulting document then is available for communicating with its several audiences, and for mapping back to prior work to confirm completeness of subsequent stages (e.g. testing functionality against initial requirements specifications). The effect of course is cumulative and the full set of artifacts produced during the project also combine to create a project portfolio for assessment purposes.

But in observing the efficacy of this approach with students, I have noticed some undesirable side-effects and negative behaviours. The underlying principle behind the document driven approach is to have documentation produced in-line, or as a natural by-product of the project rather than as an after thought hurriedly pieced together at the end. It also provides a context in which configuration management has purpose, and the issues arising from inconsistent versions become apparent. In this way documentation is not an all-encompassing 'other' category, but a natural deliverable of each stage in the process. In supporting the thinking processes and providing checkpoints for review and reference, the value of the documentation is meant to become self-evident. However, from observing some student behaviour this is probably an over-optimistic view.

In our projects I try to have the students take some responsibility for their projects and exercise judgement over their planning and execution. So they are required to develop their own plans outlining the methodology they will adopt, and identifying their key milestones and deliverables. It appears that many of them although theoretically adopting an iterative lifecycle, appear to become bogged down with completing their requirements specification. This may be partly a domain comprehension problem and indicate weakness in their analytical skills, but it seems to be more than this.

In the project guidebook given out to students I had deliberately left process and deliverables relatively open for them to decide, and more agile approaches are not precluded. Technical assessment is based upon the four broad categories of: requirements; design; construction; implementation and testing. Yet waterfall, iterative and incremental lifecycle approaches are all permissible. Few however, seem to have adopted the active use of prototypes, screen mock-ups and joint application design sessions to complement the written parts of their requirements specifications. The term 'document' as opposed to useful artifact, appears to have been taken too literally, with a narrative specification supported by various models, typically UML use cases, activity diagrams and class diagrams being common. This in turn is probably a result of referring students to a sample table of contents from an object oriented software engineering text [6] as a pro-forma template for their document – often in response to their requests for more guidance. It may also be a response to their expectations of the assessment to which they will be subject, and a reluctance to provide work-in-progress materials as opposed to completed artifacts. Perhaps it is also a failure to appreciate the evolving nature of a software application, through the murky phases of conceptual, logical and physical design. So is offering a default template guidance or misdirection then? I am currently revising the project guidebook to make some of these issues hopefully more explicit, but again this requires that students do carefully read the guidebook, which is now becoming longer each semester. If as argued by Highsmith [1] we document to communicate, then this itself is a classic documentation catch-22!

1. Highsmith, J., *Agile Software Development Ecosystems*. The Agile Software Development Series, ed. A. Cockburn and J. Highsmith. 2002, Boston: Addison-Wesley. 404.
2. Abran, A., et al., eds. *Guide to the Software Engineering Body of Knowledge*. Vol. 1. 2001, IEEE Computer Society Press: New Jersey. 228.
3. Naur, P., *Programming as Theory Building*. Microprocessing and Microprogramming, 1985. **15**: p. 253-261.
4. Ambler, S., *Essay - Agile Documentation*. 2001-2002, The Official Agile Modelling Site, Ronin International. Retrieved from http://www.agilemodeling.com/essays/agileDocumentation.htm, 14/01/2003
5. Beck, K., *Extreme Programming Explained: Embrace Change*. 2000, Boston: Addison-Wesley.
6. Bruegge, B. and A. Dutoit, *Object Oriented Software Engineering*. 2000, New Jersey: Prentice Hall.