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

Comprehending Large Code Bases - The Skills Required for Working in a "Brown Fields" Environment

In the search for answers to the effective teaching of programming at the beginner level, we are now seeing broader programs of research investigate the distinctions between reading, comprehending and writing small programs [1], [2]. In New Zealand we have joined this work with the "Bracelet" project, in which multiple institutions will investigate how students comprehend small computer programs. We hope this may help answer critical teaching and assessment questions.

A contrasting stream of research [8], [9], [10] has been investigating how professional programmers comprehend the often large and complex software artefacts which they must maintain. The importance of this work is demonstrated in the figures provided in [8] who assert that "program comprehension is a major part of software development…up to 70% of lifecycle costs are consumed within the maintenance phase and that up to 50% of maintenance costs relate to comprehension alone".

As I prepare to teach our undergraduate software engineering course this semester, I find myself grappling with the question of how to effectively convey to students the twin notions, critical to Software Engineering, of scale and complexity.

Our SE course comes as a mid-degree course in the three year AUT Bachelor of Computer and Information Science, in the semester prior to the final year capstone. The course attempts to simulate reality by adopting an "authentic learning" approach [12], and providing a project context to which the concepts taught in the accompanying lecture program may be related.

The single semester duration and size of the course naturally constrain the scope and complexity of the tasks that may be assigned. The challenge is to select a project of a suitable scale and complexity to enable SE processes and practices to be sensibly exercised, while having an assignment that can be successfully completed within the allotted time. Invariably we find ourselves in the situation where perceived complexity is insufficient for students to actively adopt the relevant practice, e.g. configuration management by use of a source control tool; careful use of work break down structures in the assignment of roles, tasks and responsibilities; selection of a suitable O.O. architecture and relevant design patterns; planning for quality assurance and risk management strategies and techniques; and regular monitoring and recalculating of estimates.

One alternative to this approach that we have been considering is to have the teams work on an existing, large code base to produce a manageable extension module.

This, at first glance, encapsulates all the aspects that we are seeking to include in the course: scope, complexity, use of others' code; integration into an existing architecture; possible refactoring of designs, the motivation of a possible contribution to a code base for an existing open source application; exposure to an existing set of development practices and standards; and a clear demonstration of the

need for documentation deliverables at key stages in the process.

We have indeed adopted this approach in the course of our capstone projects [which are larger and of longer duration] where students have operated in a "brown-fields" (pre-existing) environment contributing to or extending an existing application. These projects have been very effective learning exercises for the teams involved, but the work involved in comprehension of the software has proven very time consuming. Fully grasping what Naur has called "the theory of the program" [4] is a challenging conceptual task, involving not only comprehension of the problem domain, but also how it has been addressed by the software artefact under study.

Therefore in our far more compressed SE projects we do not adopt a brown-fields approach. One superficially attractive project could involve extending Eclipse. Billed as "a kind of universal tool platform - an open extensible IDE for anything and nothing in particular", [4] Eclipse offers a large open source code base and considerable scope for the writing of extensions and add-ons. Yet our past capstone experience suggests that SE students would struggle: firstly in comprehending this complex environment; secondly with determining how the existing code operates and thirdly with understanding where any necessary changes might be applied. With some preparatory work on our part we might be able to reduce the scope of these problems, but at this stage we have not conceived a suitable design for this development.

This set of reflections raises some critical issues for software engineering education. The SWEBOK [5] has defined a body of knowledge for the discipline, but is remarkably silent on the skills, knowledge and abilities required to master development in a "brown-fields" environment. Yet all too often this is the environment in which developers operate, integrating new components into existing software contexts, and interfacing with large and complex application frameworks, suites or software packages. Modification, interconnection and extension are increasingly the norm in software development.

So how does one go about comprehending an existing software environment? In [8] it is suggested that a combination of top down and bottom-up strategies are used by professional programmers. A more specific model of comprehension has been proposed in [10] involving a combination of three models:

- a top down model representing knowledge schemas about the application domain;
- a program model built from the bottom-up as a control flow abstraction of the program;
- a situation model also built from the bottom-up and using the program model to create a data-flow/ functional abstraction.

It was observed in 1995 [9] that research in the area of program comprehension is still in its infancy. Now ten years on, we seem little further ahead.

So if the "brown fields" environment is indeed a typical development scenario today, how are we going about preparing our students for such work? From my observations of SE texts such as [6] & [7] they tend to address the creation of new systems, rather than the comprehension of an existing context and base of software, and only loosely touch on the topic of maintenance. While object oriented design techniques may help in designing modular applications, have we really moved far from the scenario painted by Fred Brooks [11] thirty years ago "All repairs tend to destroy structure … even the most skilful program maintenance only delays the program's subsidence into unfixable chaos, from which there has to be a ground-up redesign".

Given the paucity of extant theory in this area, I suspect we have a reality of everyday practice that greatly impacts the work of professional developers. The corollary is that we have a limited theory base relating to the comprehension of code, from which to devise suitable curricula and approaches to developing and assessing the required student capabilities. Our state of the art

seems akin to studying reading without the key notions of comprehension level or reading age.

In a recent discussion with a commercial software development colleague, she even made the point that the complexity and incomprehensibility of most modern development environments actively discourages code re-use, and standard practice is to develop code components “from scratch”, the purpose and impact of which the developer can at least comprehend.

So I return to my conundrum with the SE course. How much of a large existing code base can students realistically be expected to comprehend, when developing a non-trivial extension? How can a project best be designed as an “authentic learning” [12] experience to exercise the software engineering principles and practices in a meaningful way?

1. Lister, R., Adams, E., Fitzgerald, S., Fome, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, J., Sanders, K., Seppala, O., Simon, B. and Thomas, L. A Study of the Programming Knowledge of First-Year CS Students - Draft Working Group Report, Innovation and Technology in Computer Science Education Conference. Impagliazzo, J. ed. *Innovation and Technology In Computer Science Education Conference*, ACM, Leeds, UK, (Forthcoming).
2. M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First Year CS Students," in *SIGCSE Bulletin*, vol. 33, 2001.
3. Naur, P. Programming as Theory Building. *Microprocessing and Microprogramming*, *15* (1985), 253-261.
4. Eclipse. The Eclipse.org Website [http://www.eclipse.org], Eclipse Foundation, 2005.
5. Abran, A., Moore, J., Bourque, R., Dupuis, P. and Tripp, L. (eds.). *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society Press, New Jersey, 2001.
6. R. Pressman, *Software Engineering - A Practitioner's Approach*, 5th International ed. Singapore: McGraw-Hill, 2001.
7. Bruegge, B. and Dutoit, A. *Object Oriented Software Engineering using UML, Patterns and Java*. Prentice Hall, Upper Saddle River, New Jersey, 2004.
8. M. O'Brien, J. Buckley, and T. Shaft, "Expectation-based, inference based, and bottom-up software comprehension," *Journal of Software Maintenance and Evolution: Research and Practice*, pp. 427-447, 2004.
9. A. von Mayrhauser and M. A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, pp. 44-55, 1995.
10. A. von Mayrhauser and M. A. Vans, "Program Understanding Behaviour During Debugging of Large Scale Software," presented at 7th Workshop on Empirical Studies of Programmers, Alexandria, VA, 1997.
11. F. Brooks, *The Mythical Man-Month*, Anniversary ed. Boston: Addison Wesley Longman, 1995.
12. J. Herrington, R. Oliver, and T. Reeves, "Patterns of engagement in authentic online learning," *Australian Journal of Educational Technology*, vol. 19, pp. 59-71, 2003.