ThinkingISsues

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

"Programming in the Large" and the Need for Professional Discrimination

A common goal in teaching capstone project and software engineering type courses is to develop in students the capabilities required of a professional software developer. Unfortunately their prior educational experiences in the computing discipline may have done anything but prepare them for professional practice. Although the situation may vary considerably from country to country and institution to institution, there are several common deficiencies to be addressed.

Often the introductory "programming in-the-small" courses may have consisted of several small exercises and assignments completed by the students acting as individuals. This does little to prepare them to function effectively as a member of a team, as commonly demanded by "programming in the large" in a professional context.

Likewise, earlier courses are typically oversimplified to enable students to engage in manageable, while challenging, design and coding level activities. Early assignments typically come neatly packaged, with a well-specified set of requirements to be implemented. But this emphasis only echoes the CS '91 definition of *programming* as "activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems."[1] The emphasis upon "well-specified" problems becomes problematic when the focus shifts from "programming in the small" to "programming in the large", or from programs to systems. The CS 2000 curriculum recommendations [2] again emphasize algorithms, and maintain that "The real-world performance of any software system depends on only two things: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation." While this viewpoint may be valid, it is at best only a half-truth, and omits several other considerations, when larger systems are to be developed.

Parnas cited in [3] includes among his nine tasks performed by software engineers, "analyze the intended application to determine the requirements that must be satisfied, and record these requirements in a precise, well ordered and easy to use document." This deceptively simple task description belies a huge area of complexity and challenge for practicing software developers, let alone for students. Read simplistically it implies that students are actually able to write such a document – often a major challenge for those who have flocked to computing courses as a haven for the functionally illiterate. Unfortunately, some early programming courses may have sheltered students from the requirement to express themselves extensively in the written mode.

But the written expression is far from the main challenge in this area. Parnas' comments suggest a waterfall methodology, whereby the wise software engineer can come into a complex application domain, rapidly acquire the knowledge necessary, and document a flawless requirement statement for subsequent implementation. This drastically understates the process of requirements elicitation, and omits the essentially interactive and iterative nature of the software development process.

The process of developing software for a client involves a large range of variables and several, often conflicting, demands. In capstone courses where live clients are involved, students will often fail to address these conflicting demands and instead of confronting issues will often simply acquiesce to the demands of their client. The resultant risks are that students may produce a poorly engineered product based upon the client's

unduly narrow view of their own domain area or pet software product or design preferences. Students may be led to conduct their analysis or requirements engineering insufficiently rigorously to meet the "implied" as opposed to the stated or "specified" user requirements, or may allow an unacceptable degree of scope creep.

Even in a "taught" software engineering course it will be common for the specification to have areas of ambiguity, deficiency or inconsistency, and students will frequently be expected to ask questions of the instructor as surrogate client, to clarify the requirements before they continue.

Where students fail to see the need to consult their client to gather the necessary information, or fail to confront the client over differences and merrily forge ahead in their own way to produce a result to their own specification, a potential disaster looms.

Moving students away from this unconscious arrogance of the technocratic designer, is an important preparation for informed and sensitive professional practice. The software development process is better modeled as one of joint learning, wherein the user and designer's expertise are equally acknowledged, resulting in a more active and balanced development process. The role of the software developer is not simply to focus on solving problems, but on envisioning possibilities and enabling opportunities. This envisioning process is very similar to that used in corporate strategic planning and visioning processes. [4] In working with a client it is important to work actively to create a shared vision informed by sponsor need and technology capabilities, within the overall project constraints. Thus, active management of this visioning process and maintaining good communication and shared expectations is demanded of students to ensure a process that is respectful of the needs of the parties to the development.

Use of a sound project management process, including a project plan, a methodology with agreed deliverables, predefined review points and regular progress reporting, is a useful mechanism to control the evolution of a project. Yet success demands more than formal management mechanisms and techniques. Developing professional judgement and discrimination is a complex area, and one with which students previously schooled in highly structured courses will struggle.

Let us take the case where a standard software process is mandated, whether that be a software engineering standard, a methodology imposed by a consulting firm such as Arthur Andersen, a Department of Defense standard, a development framework such as the Microsoft Solutions Framework, the client's own in-house methodology or one imposed by a CASE tool set such as the Rational Unified Process. In each of these situations there will be a number of standard processes and deliverables.

Is it then simply a matter of the student developer reading through the guides and ticking off the tasks in order? That would be a recipe for project overrun and failure. Each project has its own unique characteristics, context and requirements. While the common aspects of the methodology or toolset may be applied in a fairly standard manner, the selection of which steps to omit or adapt to the project or client circumstances requires considered judgement. How are students to acquire this judgement and be prepared to avoid mistakes, such as applying the costly, large scale corporate or military approaches in a micro business context, such as a fixed price development contract quoted by a small software house? And furthermore, how do they judge what is a professional piece of work in such a context?

This year in our capstone project course I have given students the responsibility for planning their own projects by selecting an appropriate methodology and set of deliverables. Some students have reacted well and consciously chose a suitable development approach. The incremental, the spiral model and some approaches based upon prototyping or a modified waterfall lifecycle have all been adopted. Some weaker students and teams have only rather loosely clarified their approach and the requisite deliverables. I am presently reviewing progress with each team or student and have supplemented their course guidebooks with a more detailed deliverables guide to assist students in both re-planning their projects and ensuring they are able to complete all the items demanded for their final project portfolio. But I have refused to adopt a checklist approach, both to ensure that students critically assess the needs of their own projects, and because the "one-size- fits-all" mentality is not helpful in reinforcing this need for critical evaluation and adaptation based upon reflection as the project progresses.

In conclusion, the teacher's task in developing professional discrimination also requires discrimination, in this case about the degree of supervision and support required and when to intervene.

1 ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. ACM Press, 1991.

2 Cross J., Chang C., Denning P., et al. (2001), (Eds.), Computing Curricula 2001 Volume II Computer Science, Iron man Draft —(February 1, 2001), Appendix A, CS Body of Knowledge, The Joint Task Force on Computing Curricula, IEEE-CS, ACM

3 Bourque P., Dupuis R., et al. (1998), (Eds.), SWEBOK Guide to the Software Engineering Body of Knowledge, Straw Man Version, IEEE-CS

4 Lipton M., (1996), Demystifying the Development of an Organizational Vision, *Sloan Management Review*, Summer pp. 83 - 92