# *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

## Disciplined Design Practices – a Role for Refactoring in Software Engineering?

Reflecting upon the recent experience of teaching our undergraduate software engineering course has caused me to revisit several questions at the core of the discipline. What is the essence of software design, how should it be taught and how does it relate to software engineering?

Turning to the Guide to the Software Engineering Body of Knowledge (Swebok) [1] issues related to design can be found in two of the ten knowledge areas, which are identified as "software design" and "construction". Software engineering itself is then further defined as: "the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software" [1].

In distinguishing the "construction" activity, the Swebok [1] notes that linear models of the software development process treat construction as "an activity that occurs only after significant prerequisite work has been completed – including detailed requirements work, extensive design work and detailed planning". So then, does this 'system', 'discipline' and 'quantification' of the SE process inherently depend upon a structured linear process, with progressive delivery of robust formalized artifacts? Problematically SE courses adopting these approaches, while enabling the instructors to design a tidy sequential course structure, often result in a student perception of software engineering as "document engineering".

Yet, in elaborating upon construction models, the Swebok [1] further notes that more iterative models including "evolutionary prototyping, Extreme Programming and Scrum…tend to treat construction as an activity that occurs concurrently with other software development activities, including requirements, design or planning". In such models design, coding and testing activities are intermingled, and in combination tend to be treated as "construction". Which begs the question for SE - where does the system, discipline and quantification lie in such iterative "construction" models??

My own view on the question of software development and design as expressed in our capstone project guide offers only a partial response.

"In some sense we may think of development as involving a mapping process, which perhaps more generally reflects the whole process of design. This mapping process takes some real world practice or issue, transforms it into a conceptual model or series of models, and then further transforms that into a computer implemented solution. In a good software development process, these transformations reconceive the real world practice in some way that will improve upon the present. So the developer works in partnership with a client to add value in producing the new work practice or process and/or supporting technologies and software" [2].

In addition to this model of design as a mapping process, the conclusion we have come to regarding

the “system” and “discipline” of SE, is that it lies inherently in a series of practices and processes which support the activities involved in ‘software development’ – a term I much prefer to the building metaphor of ‘construction’.

So in returning to our question of ‘design’ and the teaching of design versus construction, we can now consider the question of effective design practices, and how can they be taught in the context of more iterative or agile methodologies.

Here we can introduce the notion of refactoring [3] as one such useful practice. “Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written”.

“With refactoring you find the balance of work changes. You find that design, rather than occurring all up front, occurs continuously during development. You learn from building the system how to improve the design. The resulting interaction leads to a program with a design that stays good as development continues.” [3].

By way of contrast with this active, continual and iterative model of design and construction suggested by Fowler, the Swebok appears to consider refactoring merely as a maintenance technique, being a reverse engineering method, for program transformation “which seeks to improve program structure” [1].

Given the Object Oriented nature of the Java development projects that our SE teams undertake, and the variable nature of the lifecycle models they may select, the idea of formal instruction in refactoring seemed a useful contribution to improving the design and construction practices of our SE teams. Therefore we incorporated a session on refactoring into the software engineering course this semester. We surmised that the iterative O.O. development undertaken by students would involve a fair amount of tweaking of their code, and from past experiences this redesign would often result in software far removed from earlier versions of any class diagrams that may have been developed in an initial design activity. Which begs the question - does a retrospectively developed tidy class diagram handed in with the final portfolio submission constitute good design practice? Or does this simply represent the student state of the art, a “hack first - document last” methodology?
If this then could be predicted as the classic outcome from the team’s design process in a “requirements, design, construction” model of SE, perhaps more conscious practices supporting continual and iterative design, and closer to the code itself may be helpful and may actually be applied by students.

At the completion of the course, students being students it was not apparent that all our teams had consciously applied refactoring and reflected upon the practice. However, one team definitely used it to good effect, and two of the specifically mentioned refactoring procedures were the ‘move method” and the “extract class”.

The “Move method” procedure supports reduced coupling by moving a method from one class to another class in which it more naturally belongs. The feedback was that it had indeed proven useful, had simplified and cleaned up a lot of the design, both reducing the amount and increasing the quality of the code.

The “extract class” procedure creates a new class with selected attributes and methods from other classes to improve cohesion. Use of this procedure was reported by the team to have avoided unnecessary repetition of attributes and methods in their java bean classes and helped to create a cleaner hierarchy and higher quality design.

Thus it appeared that the team were able to comprehend some of the reasonably simple yet powerful techniques of refactoring to improve their design and the quality of their code as the project developed. For me this demonstrated powerfully the value of refactoring as an active design and construction technique, and one which has a definite place in any SE course attempting to teach sound O.O. design practices.


1. Abran, A., Moore, J., Bourque, P., DuPuis, R. and Tripp, L. Guide to the Software Engineering Body of Knowledge - 2004 Version - SWEBOK®, IEEE-CS - Professional Practices Committee, Los Alamitos, California, 2004, 202.
2. Clear, T. Bachelor of Computer & Information Sciences - Research & Development Project Guidebook, v. 1.9, Auckland University of Technology, Auckland, 2005, 1 -50.
3. Fowler, M. *Refactoring - Improving the Design of Existing Code*. Addison Wesley Longman, Boston, 1999.