Is teaching software design a ‘wicked’ problem too?

David Budgen
Department of Computer Science
Keele University
Staffordshire U.K.
ST5 5BG
db@cs.keele.ac.uk

Abstract

The creative act of design has been described as being a wicked or ill-structured problem, in which finding a solution to one aspect may only serve to reveal a more complex problem. The task of designing software offers additional dimensions to this, in that the media is invisible, is capable of almost infinite variations of form, and has both dynamic and static properties. These characteristics provide difficulties when teaching our students about software design, and our ideas about how to do this also tend to be coloured by our experiences in teaching them how to program. In this paper we argue that an approach based on a ‘programming’ metaphor is inadequate for teaching students about designing software, and that the activity of teaching design is not readily amenable to the use of highly structured teaching practices. Indeed, we can therefore identify this too as being an example of a wicked problem.

1 Introduction

It is possible to argue that it is those aspects of Software Engineering practice that are both creative and abstract that provide the greatest difficulty for the teacher. Some particular examples that come to mind are the activities that are involved in performing such tasks as those of requirements elicitation, software design and system testing. All of these involve an element of creative thinking, that needs to be combined with experiences gained from practice; and such skills are particularly difficult to convey via the classroom or the teaching laboratory.

This paper focuses upon the problems of teaching software design skills to students, and draws upon experience with:

- Teaching undergraduates and postgraduates.
- Teaching practising software developers.
- Developing a methodological study of design practices [1].
While these experiences have revealed the inadequacies of the teaching strategies used, they have unfortunately been less helpful in providing pointers to any approaches that might be more effective. (As an aside, a survey of the papers presented at the last three CSEEs shows only a total of six papers addressing the issues of teaching design, mostly describing experiences with either fairly low-level design issues or with tools. The papers also share a general recognition of the difficulties of teaching design, a point particularly addressed in [2].) To counter this emphasis on the problematical aspects of teaching design, the later parts of this paper explore some ideas about how we might seek to improve the ways that design is taught.

Our ideas about teaching software design knowledge and skills are commonly centred around the following two strategies:

- Teaching *about* designing software, by describing the nature of the processes involved and the strategies that are used.

- Teaching *how* to design software, by providing some framework that can be used to transfer knowledge that has been gained from experience (which is of course one of the major roles of a design method).

Ideally, a combination of these should serve to give the student both insight into the field by means of a survey in breadth, and also a deeper understanding by providing experience. In practice though, finding a satisfactory blend seems to be an elusive goal.

In [3], Fred Brooks identified some of the particular attributes of software that serve to complicate the teacher’s task. Among others, he identified complexity, changeability and invisibility as being properties that were of particular importance. While changeability may be less significant in a teaching context, the other two properties ensure that any abstractions used for describing software designs are unlikely to be either simple or two-dimensional.

Our problems in teaching design do not entirely arise from the nature of software itself. The design task is itself a complex one, regardless of the domain, and the description of design as a *wicked* problem [4] (which was coined in the completely non-software context of social planning) suggests that design tasks have undesirable properties in their own right. Figure 1 summarises the set of rules that Rittel and Webber originally proposed as those defining the form of a wicked problem, and almost all of these can be related quite directly to the characteristics that we see in software design activities. A more concise description is that:

*Solving one aspect of the problem will merely reveal another—with the new one potentially being more complex than the original.*

The concept of a wicked problem can be quite a difficult one to grasp for those of us who have traditional scientific trainings. We (both teachers and students) are apt to be accustomed to working with forms such as equations where complexity can normally be reduced by separating the problem into more simple components and then solving these. We are also accustomed to the concept of there being a ‘right’ answer to most of our problems (such as measuring the value of g, the gravitational constant, at a given point on the earth’s surface).