Bad theory versus bad teachers: Toward a pragmatic synthesis of constructivism and objectivism

Raymond Lister
University of Technology, Sydney, Australia
raymond@it.uts.edu.au

John Leaney
University of Technology, Sydney, Australia
johnl@it.uts.edu.a

Abstract: *Booth (2001) identified three views of computer programming, which we refer to as the "coding", "problem solving", and "social" views. The traditional approach for teaching programming to novices does not explicitly distinguish between these three views. Furthermore, the teaching of novice programming is traditionally objectivist. In our approach to teaching programming, the coding view is taught using an objectivist approach while the social view is taught using a constructivist approach, while the problem solving view is taught using a blend of both approaches.*

Keywords*: constructivism; objectivism; Hegelian dialectic transcendence*

Introduction

The language of research is frequently the language of conflict and conquest where, in the form of a "paradigm shift", as expounded by the philosopher Thomas Kuhn (see Andersen, 2001), a superior theory replaces the orthodox theory. In education research, proponents of constructivism argue that their learner-centred theory is superior to the teacher-centred orthodoxy of objectivism, and that a paradigm shift is underway. Much of the recent education literature asserts that the orthodox objectivist approach has failed completely. Objectivism, so it is asserted in the literature, is a system where the teacher (the 'sage on the stage') drones out small, predigested dollops of information, where assessment exercises have no real connection to how the student will apply their skills on graduation, where students are implicitly encouraged to adopt a shallow approach to learning (Biggs, 1999; Ramsden, 1992). That same literature abounds with positive reports of subjects reorganised according to constructivist principles, where the teacher (the 'guide on the side') provides a rich, authentic 'real world' learning environment in which students construct their own knowledge.

In the practice of teaching, however, we need to distinguish between bad theory and bad teachers. Biggs (1999, p. 12) wrote, "most teachers … are not interested in theories of learning as much as in improving their teaching". Biggs did not mean that teachers should be completely uninterested in theories of learning. Instead, he was merely observing that, while the cut and thrust of 'theory wars' may be the stuff of life for the academic-as-researcher, such wars are of far less interest to the academic-as-teacher. Accordingly, for the practicing teacher, the philosophy of Hegel (see Trejo, 1993) rather than Kuhn may provide a better model for growth and change. Hegel argued that any theory (a thesis) is incomplete. This incompleteness leads to the development of a community who believe in an alternative theory (the anti-thesis) that addresses the shortcomings of the thesis. However, this anti-thesis is also incomplete. Genuine progress comes from synthesis, when the conflict between the thesis and anti-thesis are reconciled at a higher level. In this paper, objectivism is the thesis and constructivism is the anti-thesis. This paper describes a subject that introduces first year students to computer programming. The subject transcends the constructivism versus objectivism debate, synthesising the two philosophies.

Three views of computer programming

In a phenomenographical study, Booth (2001) identified three qualitatively distinct ways ("views") in which students experience programming. These are discussed below.

Programming as coding

The first student view of programming is oriented toward the computer itself. In this view, the student focuses upon the mechanics of producing program code that compiles (i.e., the computer recognises the program as a legal set of instructions).

Programming as problem solving

In this view, the student focuses upon producing a program that, not only compiles, but also exhibits the desired behaviour. The student's focus is on the problem to be solved, not the computer. Most academics teaching a first-year programming subject focus upon problem solving (Fincher, 1999). Many programming textbooks contain the term ‘problem solving’ in their title, indicating that emphasis is placed upon teaching students how to take an ambiguous problem description, refine it, and decompose the program into sub-problems, before proceeding with the coding.

Programming as a social activity

The third view is when the student focus is upon the computer program as a product, to be used by other people. In this view, it is not sufficient to produce a correct program, it must also be a quality solution. This view has two people-oriented facets. First, the student must be conscious of the people who will use the program, making the program easy for those people to use, always being conscious that not everyone thinks the same way; that what is obvious to one person is obscure to another. Second, the student must be conscious of other programmers. If a program is widely used and long-lived, then it is likely to be updated during its lifetime to perform some extra functions (consider the many versions of Microsoft's operating systems). A good programmer designs a program so that its code can be easily read and modified by another programmer.

Synthesising objectivism and constructivism

In this section, we explore the advantages and disadvantages of objectivism and constructivism, in the context of the above three views of programming. The difference between constructivism and objectivism is illustrated as follows. Suppose our aim is to improve the written expression of primary school children. A constructivist approach would be to get the children to produce a class newspaper. Early editions of the newspaper may contain many instances of poor expression, but with guidance and time, we might hope to see an improvement. The newspaper is an ‘authentic’ learning task, in that the students can see how it relates to something of significance in their world.

This authenticity should help to maintain enthusiasm. On the other hand, an objectivist approach might be to teach the rules of grammar (e.g., ‘never split your infinitives’).

Advantages of constructivism in programming

The constructivist approach is very well suited to developing a student's view of programming as a social activity. We will elaborate on how we do that later in this paper, when we describe the learning task we set for the ‘high distinction’ grade.

Disadvantages of constructivism in programming

A constructivist approach is not well suited to developing the coding view in students. Buck and Stucki (2001, p. 17) observed that many students who are attempting to write code with little direction from teachers are "overwhelmed, uncertain of how to begin, and grasping at the air … [leading] … to the self-destructive tendency to do experimental programming, where they just randomly throw things in to see if it helps." In their landmark multi-institution study of first year programming students, McCracken et. al. (2001) noted that weak students become so focussed upon the coding aspect of programming that they neglect the problem solving view and "are then surprised by what the program really does when presented with data"(p. 135)

For the middle ‘problem-solving’ view of programming, constructivism may have some advantages. It has long been a common practice in most universities to give first-year programming students an assignment with some constructivist elements. The students are given a vaguely specified problem statement, and they are expected to both refine the problem and develop a solution. Students who complete such assignments without cheating undoubtedly do learn a great deal. However, it has long been suspected among IT academics across Australia that cheating is common in such assignments. Recent survey results suggest that around half of IT students have cheated on such assignments (Sheard, Carbone, & Dick, 2003), which suggests that half the students are not ready, in their very first semester, for a constructivist approach to learning problem solving. While all students should at some point in their IT degree demonstrate the ability to solve such problems, the very first semester appears to be too early for around half of programming students.

Advantages of objectivism in programming

It is crucial that we distinguish between *rote learning* and *meaningful reception learning* (Ausubel, 1963). Rote learning occurs when bad teachers present facts without any organising or explanatory principles, without integrating new learning tasks with previously presented materials, and when students are tested by measuring their ability to regurgitate these isolated facts in the same way or context in which those facts were originally presented. We are not defending rote learning.

Ausubel (1963) summarised the role of the objectivist teacher as the presentation "of ideas and information meaningfully and effectively - so that clear, stable, and unambiguous meanings emerge and are retained over a long period of time as an organized body of knowledge" (p. 19). The objectivist teacher takes on "the job of selecting, organizing, presenting, and translating subject-matter content in a developmentally appropriate manner" (p. 19). In constructivism, the students build their own knowledge structure. In objectivism the structure is given.

While the structure of knowledge is given, meaningful reception learning is not passive. As any good lecturer knows, it is one thing for a student to watch the lecturer apply a technique to a problem, and another thing for the student to demonstrate a command of that technique on a different problem.

Ausubel (1963) gave the obvious advantage of objectivism: its efficiency. It is faster to receive a wisdom (assuming it is possible to receive it) than to construct the same wisdom by the trial and error of constructivism. Time and money are real considerations in today's universities.

Computer programmers need a common vocabulary, and a common set of concepts, so that they can communicate with their fellow professionals. To some extent there is a single world that all programmers share. There is less scope for each programmer to explore in the constructivist tradition and build his or her own meaning. The objectivist approach to teaching is a student's initiation into the shared single reality of the programming profession.

Disadvantages of objectivism in programming

The danger with objectivism is that a bad teacher will become overly absorbed in teaching the detail, and the student loses the vital connection between the material being taught and the 'big picture', the 'real world'. When that happens, students will rush to adopt a shallow approach to learning. Students are then reluctant to think for themselves, and they approach learning as if its is simply the acquisition of facts, preferably the facts in the exam. There is little transference of skill to previously unseen problems. These problems are amply documented in the contemporary education literature (Biggs, 1999; Ramsden, 1992). When teaching in the objectivist approach, a good teacher must emphasise the 'big picture', must stress the relationship with the 'real world'.

Given objectivism's emphasis on 'objective' knowledge, and its emphasis on learning as an individual activity, objectivism is not a promising approach to teaching the social view of programming.

The 'P' grade and the programming view

At our university, there are four passing grades, with a 'Pass' (or just 'P') being the lowest. In our criterion-referenced grading scheme, students qualify for a 'P' if they demonstrate mastery of the 'coding' view of programming. The teaching aimed at this grade is very much in the tradition of objectivism, with lectures, and prescribed exercises for tutorials and laboratory sessions. Students must perform satisfactorily in three assessment components to demonstrate their mastery: a set of graduated lab exercises, an invigilated lab exam, and a final multiple-choice exam. For each component, students are given at least two attempts. The pass mark for the multiple-choice exam is set to 70 percent, a typical pass threshold for 'mastery' exams. Approximately one-third of the class fail the multiple-choice exam at their first attempt, but the failure rate improves to approximately 10 percent after their second attempt at a similar exam.

We have already acknowledged that a weakness of the objectivist approach is that it can lead to shallow learning, or even rote learning, as students may focus upon the assessment exercises, not the deeper issues of the subject. Once again, we draw a distinction between bad theory and bad teaching. As long as a good teacher is aware of this weakness of the objectivist approach, steps can be taken to minimise the impact of this weakness. For example, in our lectures, we emphasise the study of various concepts of programming within the context of complete programs. We begin by demonstrating the program, showing the students that the program has a behaviour that is of interest to them, and has some bearing on the real world. It is possible that this program has parts that are incomprehensible to those novice programmers who aspire for the 'P' grade. We find, however, that students are prepared to ignore that part of the program, after being reassured that they are not expected to understand that part, and they will focus on the part that we want them to study.

An examination of student performance on our multiple-choice questions indicates that students are not passing by rote learning. Our exam contains a mix of questions. Some questions had been seen by students before the exam, and some questions are ‘unseen’ (i.e., were about code that students had not seen before the exam). Analysis of the answers from the ‘bottom passing’ students (i.e., students who scored 70-77%) indicate that these students have generalised beyond the previously seen material. In our most recent exam, 63-85 percent of bottom-passing students correctly answered each of the unseen questions, with a median of 78%. On previously seen questions, 63-89% of bottom-passing students correctly answered each of those questions, with a median of 81%. These comparable percentages suggest that bottom-passing students have not engaged in rote learning, but instead have engaged in meaningful reception learning (Ausubel, 1963).

One manifestation of bad objectivist teaching is a multiple-choice exam where all questions are based upon previously seen material. In multiple choice exams where there is a mix of seen and unseen questions, another manifestation of bad objectivist teaching is the absence of any quality assurance process that generates statistics like the ones we gave above.)

The 'C' and 'D' grades and the problem-solving view

Above the “P” grade, there are three higher grades, ‘Credit’, ‘Distinction’, and ‘High Distinction’ (or simply ‘C’, ‘D’, ‘HD’ respectively). In developing our grading criteria, we found it relatively easy to assign criteria to ‘P’ and ‘HD’. However, we struggled to devise qualitatively different criteria to distinguish between the middle two grades, ‘C’ and ‘D’. We decided to assign the same broad criteria to the middle two grades. To be awarded a ‘C’ or ‘D’ grade, students must first meet all the requirements for the 'P' grade, and also develop problem-solving skills.

We do not teach problem-solving explicitly in our lectures and tutorials. Instead, we set a programming assignment, and allow each student to acquire problem-solving skills as they attempt the assignment. The assignment requires the students to make a series of changes to a given computer program. The first few changes require the student to write very little code, but instead require the student to demonstrate an understanding of the code provided. The sophistication of the code to be written by the student rises with each task. The final exercise is quite demanding, and little direction is given to students on how to achieve it.

Our approach to the acquisition of problem solving skills is a blend of objectivist and constructivist principles (unlike some universities, where entire subjects, taught along objectivist lines, are devoted to teaching problem solving). The first few changes to the program tend more toward the objectivist philosophy, while the latter tasks are closer to constructivist. The early tasks are objectivist in that, although we do not lecture on problem-solving, we provide written advice on how the changes should be made. In particular, these early tasks are designed to reinforce specific programming principles, and we nominate those principles to the students. The latter changes that students are required to make are strongly aligned with constructivism, in that little direction is given on how they should be done, and the student must acquire a wider grasp of programming skills. The assignment as a whole leans toward constructivism, for two reasons. First, the program to be modified is designed to be a credible example of a ‘real world’ problem. In the most recent semester, the program was a computer game. Second, modifying a program is an authentic task. As we explained earlier, if a program is widely used and long-lived, then it is likely to be changed during its lifetime.

Since the "P" grade is entirely determined by the lab exercises, the lab exam, and the multiple-choice exam, participation in the assignment does not directly affect a student's likelihood of failure.

Typically, about one-third of a class elect to settle for a ‘P’ and not attempt any part of the assignment. Among those who do attempt the assignment, we find the atmosphere less fraught than in traditional classes, where the assignment does influence pass/fail. This calmer atmosphere is more conducive to learning and less conducive to plagiarism.

The 'HD' grade and the social activity view

To be awarded an 'HD', students must meet all the requirements of the lower grades, and participate in a two-phase activity designed along constructivist principles. The aim of these activities is to reinforce the view of programming as a social activity. The first phase is an individual project. The second phase is a peer evaluation of other projects from the first phase. About 10 percent of the class elect to participate.

In the fifty years since the first computer programs were written, computer scientists have painfully developed many rules-of-thumb for producing programs that are easy to use and easy to change. However, the reasons for these rules are not obvious to a novice. To the novice programmer, these rules-of-thumb often seem pointless. While these rules make programming easier for the acculturated expert programmer, these rules make reading and writing programs harder for a novice. In most first-year programming subjects across the world, these rules of thumb are taught in the objectivist tradition. Lecturers do indeed drone out these rules in small, predigested dollops, and the utility of these rules is not obvious in the small programming exercises that students are required write. In this aspect of teaching programming, objectivism has been a failure. It is a common lament among IT academics that students write ‘bad’ code. The authors have come to the view that the importance of good coding style, the social side of programming, is best learnt constructively.

Individual project

For a grade of ‘HD’, students are first required to write a program of their own choice. There are no restrictions on what the program does. We encourage them to write a program that is of some interest to them. The only restrictions we apply are designed to ensure that this project is not a token effort. We specify a certain minimum size for the project, and prescribe that certain coding features must be present in the program. Beyond that, the students are free to play and learn as they please.

Student peer review

The project described above is merely a student's ‘entry ticket’ into the ‘HD community’. After completing their own projects, students must then examine the projects of two or three other students, and write a report. We do not provide detailed criteria for the evaluations. Instead, we provide two broad criteria for the evaluation. The first criterion relates to the useability of the program. Essentially, we ask each student, "Did you find the other student's program easy to use?” The second criterion relates to the potential mutability of the programs. Essentially, we ask each student, "Do you understand the code written by the other student?”

The layperson's stereotype of programming is that of the ‘computer nerd’, alone with their computer. The reality is quite different. The need to produce useful software makes communication skills paramount among professional software engineers. However, we acknowledge that the traditional objectivist approach to teaching programming does encourage nerdish behaviour. In devising our constructivist approach for the ‘HD’ grade, our aim was to breakdown the nerd mentality among strong students. We wanted our strong students to think of a good program as something that is worth reading. When a strong student is writing code, we want them to maintain strong mental images of a peer reading ‘over their shoulder’ and of someone other than themselves using the software.

If these social aspects of computing are so important, why not get the non-HD students to go through this constructivist learning exercise as well? We certainly think they should, as soon as they are ready for it. However, we believe that many of our students are not ready in their very first semester. Earlier in the paper, we cited literature reporting that many students struggle when given little direction on how to complete programming tasks (Buck & Stucki, 2001, McCracken et. al., 2001). Even more compelling than the literature is the choices our own students make. That our students decide for themselves whether they wish to participate in the HD activity, and 90 percent choose not to participate, is a powerful indication that these students are not ready for this constructivist approach.

Student feedback

In the most recent completed semester, the class was surveyed as a routine part of the faculty's quality assurance process. The students were asked the faculty's standard set of questions, to which they answered, on the common 5-point Likert scale (strongly disagree, disagree, neutral, agree, and strongly agree). One of the survey items was "my learning experience in this subject were interesting and thought provoking". In response 11 percent of the class strongly agreed, 58 percent agreed (total 69%), 18 percent were neutral, and 13 percent disagreed. Another survey item of great interest was "I found the assessment fair and reasonable". In response 7 percent of the class strongly agreed, 57 percent agreed (total 64%), 30 percent were neutral, and only 7 percent disagreed. These are good figures for a first semester IT subject. These survey figures confirm our anecdotal experience that the majority of students are happy with our approach.

Conclusion

It is an error to confuse bad theory with bad teaching. Both objectivism and constructivism have strengths and weakness. A bad teacher is more likely to apply either constructivism or objectivism in such a way as to illustrate their weaknesses. A good teacher is neither a complete constructivist nor complete objectivist. A good teacher is responsive to the needs of their students, sometimes working on the detail, sometimes the big picture, sometimes leaving the student to struggle with a problem, sometimes stepping in to provide support. Good teachers live the synthesis of objectivism and constructivism.

References

Andersen, H. (2001) *On Kuhn*. Wadsworth publishing

Ausubel, D.P. (1963). *The psychology of meaningful verbal learning*. London: Grune & Stratton.

Biggs, J. (1999). *Teaching for quality learning at university*. Buckingham: Open University Press.

Booth, S. (2001). *Learning to program as entering the datalogical culture: A phenomenographic exploration*. 9th European Conference for Research on Learning and Instruction, Fribourg, Germany. Retrieved from http://www.pedu.chalmers.se/shirley/EARLI2001/Booth.pdf

Buck, D. & Stucki, D. (2001). JKarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum. *Proc. SIGCSE Technical Symposium on Computer Science Education,* Charlotte NC, USA, (pp. 16-20), ACM Press.

Fincher, S. (1999). What are we doing when we teach programming? *Proceeding of Frontiers in Education '99,* 12a4-1 to12a4-5, IEEE Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B-D., Laxer, C., Thomas, L. & Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, *SIGCSE Bulletin, 33*(4),1-16. ACM Press.

Ramsden, P. (1992) *Learning to teach in higher education*. London: Routledge.

Sheard, J., Carbone, A. & Dick, M. (2003). *Determination of factors which impact on IT students' propensity to cheat*. Fifth Australasian Computer Education Conference, Adelaide.

Trejo, P. (1993). *Summary of Hegel's Philosphy of Mind*. Retrieved from http://eserver.org/philosophy/hegel-summary.html.