A Tool to Support the Use of Part-Complete Solutions in the Learning of Programming

Stuart Garner
Edith Cowan University, Australia
s.garner@ecu.edu.au

Abstract

The purpose of this paper is to present a new software tool that has been developed, the purpose of which is to help novices learn programming. The tool supports what is know as the "completion" method of learning to program.

It begins by discussing the difficulties that students face when learning to program and the use of part-complete solutions as a teaching and learning method. CORT has been developed to support this use of part-complete solutions and its features are outlined. When used by a student, a part-complete solution to a given programming problem is displayed in one window and possible lines of code that can be used to complete the solution are displayed within another window. The lines can easily be moved between the windows in order to complete the solution, the solution then being transferred to the target programming environment for testing purposes.

Preliminary feedback from students indicates that CORT is easy to use and perceived to be helping them in their learning of programming. Three different methods of using CORT have been identified and these will be the subject of future research.

Keywords: Scaffolding, Programming, Flexible Learning

Introduction

Learning to write computer programs is not easy (du Boulay, 1986; Scholtz & Wiedenbeck, 1992) and this is reflected in the low levels of achievement experienced by many students in first programming courses. For example Lisack (Lisack, 1998) states that:

> *Students have difficulty learning programming as they are trying to develop skills in three areas at the same time, these being: using the program development environment; learning the programming language syntax; and developing logic design.*

Also, the learning of programming is becoming more difficult because the event driven nature of such languages

adds to the already high cognitive load of the subject. As Lisack points out:

> *With the new event-driven environments, the complexity of the design process is magnified for some students.*

Much has been written about the problems that students have in learning programming and many ideas and initiatives have been put forward for improvements in the teaching and learning process with varying degrees of success. In practice, the ways in which teaching and learning takes place in the domain of programming have changed little and many students still find the learning of programming a very difficult process. As Fowler and Fowler (Fowler & Fowler, 1993) point out, the challenge of learning programming in introductory courses lies in simultaneously learning: general problem solving skills; algorithm design; program design; a programming language in which to implement algorithms as programs; and an environment to support the program design and implementation. In addition, students need to learn testing and debugging techniques to validate programs and to identify and fix problems that they may have within their programs.

Additionally, we are moving ever more rapidly to use more student centred and flexible learning methods within the teaching and learning process. This means that our instructional design for programming courses needs to take notice of these moves and utilise these methods. Fortunately technological improvements have also been significant over the last few years enabling us to more easily produce engaging courseware that can help students studying in a flexible learning mode. As courseware designers, we can produce electronic scaffolds to help students in their learning processes when they are studying on their own with limited access to a human tutor.

Use of Worked Examples in the Teaching and Learning of Problem Solving and Programming

One of the methods used in the teaching and learning of programming makes extensive use of worked examples. Several researchers have experimented with the use of worked examples in place of conventional instruction and found strong advantages. In the domain of algebra, Sweller and Cooper suggested that students would learn better by studying worked examples until they had "mastered" them rather than attempting to solve problems as soon as they had been presented with, or familiarised themselves, with new material (Sweller & Cooper, 1985). In their research, students studied worked examples and teachers answered any questions that the students had. Students then had to explain the goal of each problem together with the steps involved in the solution and then complete similar problems until they could be solved without errors. Sweller

and Cooper found that this method was less time-consuming than the conventional practice-based model and that students made fewer errors in solving similar problems than students who were exposed to the conventional practice-based model of instruction. There was no significant difference between the "worked example" group and the "conventional" problem solving group when they attempted to solve novel problems and it was therefore concluded that learning was more efficient and yet no less effective when this worked example method was used.

The "reading" method of learning programming makes use of worked examples. According to Van Merrienboer (van Merrienboer & Krammer, 1987; Van Merrienboer, Krammer, & Maaswinkel, 1994), the reading approach emphasises the reading, comprehension, modification and amplification of non-trivial, well-designed working programs. However, they also suggest that presenting worked examples to students is not sufficient as the students may not "abstract" the programming plans from them, a plan being a stereotyped sequence of computer instructions as shown in figure 1.

"Mindful" abstraction of plans is required by the voluntary investment of effort and the question then arises as to how we can get students to study the worked examples properly. In practice, students tend to rush through the examples, even if they have been asked to trace them in a debugger, as they often believe that they are only making progress in their learning when they are attempting to solve problems.

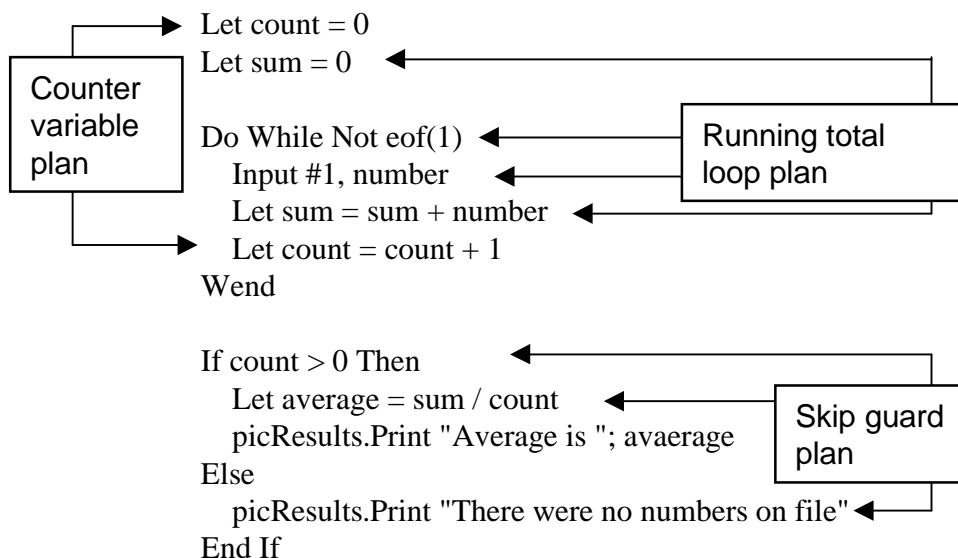One suggestion (Lieberman, 1986) is that students should

Figure 1

223

annotate worked examples with information about what they do or what they illustrate. Another suggestion is to use part-complete, well-structured and understandable program examples that require students to generate the missing code or "complete" the examples. This latter approach forces students to study the part-complete examples as it would not be possible for their completion without a thorough understanding of the examples' workings. An important aspect is that the part-complete examples are carefully designed as they have to contain enough "clues" in the code to guide the students in their completion. It is suggested that this method facilitates both automation, students having blueprints available for mapping to new problem situations, and schemata acquisition as they are forced to mindfully abstract these from the incomplete programs (Van Merrienboer & Paas, 1990).

The stimulation of the "mindful of abstraction" of schemata in students can possibly be improved further requiring them to also annotate the solutions with details of the scope and goals of the solutions and to answer questions on the inner workings of the solutions. The "degree" of completion of the solutions is an important aspect within the completion strategy and in some later work (van Merrienboer et al., 1994) examples are given of completion assignments that might be used early and later in a programming course. In an early part of a course, an example may indeed be complete and include explanations and a question on its inner workings. In the latter part of a course, the example may be largely incomplete and include a question on its workings and an instructional task. Between these two extremes, examples will have varying degree of completeness and in all cases, the incomplete examples are acting as scaffolds for the students.

The Cloze Procedure

CORT, the scaffolding tool that has been produced, allows students to fill in lines of missing code from programs and this method is based upon the cloze procedure. The term is derived from "closure", a Gestalt psychology term referring to the human tendency to complete a familiar but not quite finished pattern (Cook, Bregar, & Foote, 1984). The use of cloze was first used to measure comprehension in English readability (Klare, 1974-75) however it has also been used in the teaching and learning of programming as a way of measuring student understanding of programs (Hall & Zweben, 1986; Thomas & Zweben, 1986). Such program comprehension tests are constructed by replacing some of the "words" or tokens by blanks and requiring students to fill in the blanks during a test. The use of the cloze procedure in testing was found to correlate well with conventional comprehension, question – answer, type

quizzes and is also much easier to create and administer, see for example (Cook et al., 1984).

Other researchers have experimented with the testing of program comprehension by omitting complete lines of code from programs and requiring students to fill in those lines (Norcio, 1980a; Norcio, 1980b; Norcio, 1981; Norcio, 1982; Ehrlich & Soloway, 1984). Norcio found that students were more likely to supply correct statements if they had been omitted within a logic segment rather than from the beginning of a segment. This is consistent with the chunking hypothesis (Miller, 1956) that specifies that the first element of a chunk provides the key to the contents of the entire unit. Ehrlich looked at the differences between experts and novices in filling in missing lines within various programming plans and, as expected, found that the experts filled in the lines correctly taking into account the surrounding plan whereas novices had more difficulty.

In the various experiments in program comprehension using the cloze procedure, the students had to fill in the lines of code without being given a selection of lines to choose from. In some work done in an area unrelated to programming, students were expected to create an essay using a file of statements, only some of which were relevant to the topic (Edward, 1997). The students were expected to copy and paste only the statements which they believed to be relevant and then to link them with their own text and it was suggested that learners would consolidate their understanding of the topics by having to actively evaluate all possible statements. The file of statements was acting as a scaffold to student learning.

Although the literature suggests that the cloze procedure has only been used in measuring program comprehension, it appears that it could prove useful as a way of scaffolding student learning of programming. An incomplete solution to a programming problem could be given to a student together with a choice of statements that might be used in the solution. The student would then have to study the incomplete solution and the choice of statements and decide which statements to use and where to put them. CORT uses this idea making the mechanics of placing the statements into the incomplete solution very straightforward for the student and eliminating typing errors and therefore also syntax errors.

The Code Restructuring Tool (CORT)

CORT has been designed to support the "completion" method of learning to program and it was decided that the following features would be required in the first prototype:

- Support for part-complete solutions to programming problems. Such solutions help in schemata creation and also reduce cognitive load.
- A mechanism so that missing statements can easily be inserted into a part-complete solution and also moved within that solution. This provides scaffolding for students.
- A facility so that students can add and amend lines of code. This would allow scaffolding to be reduced and for students to add more of their own code.
- For visual programming, a facility for students to easily view the target interface. The interface should be annotated with the various object names thereby reducing any split-attention effect and helping reduce cognitive load (Chandler & Sweller, 1991).
- A facility to access tutor created questions concerning the programming problems being attempted and for

students to enter answers to those questions. This will promote reflection and higher order thinking.
- A facility to easily transfer a completed solution from CORT to the target programming environment.
- A facility to easily transfer programming code from the target programming environment back into CORT for further amendment.

The CORT Design

The user interface of CORT has been designed taking into consideration the three issues that have been suggested by Marcus (Marcus, 1992) as being fundamental to interface design, namely development, useability, and acceptance. The interface for CORT is shown in figure 2.

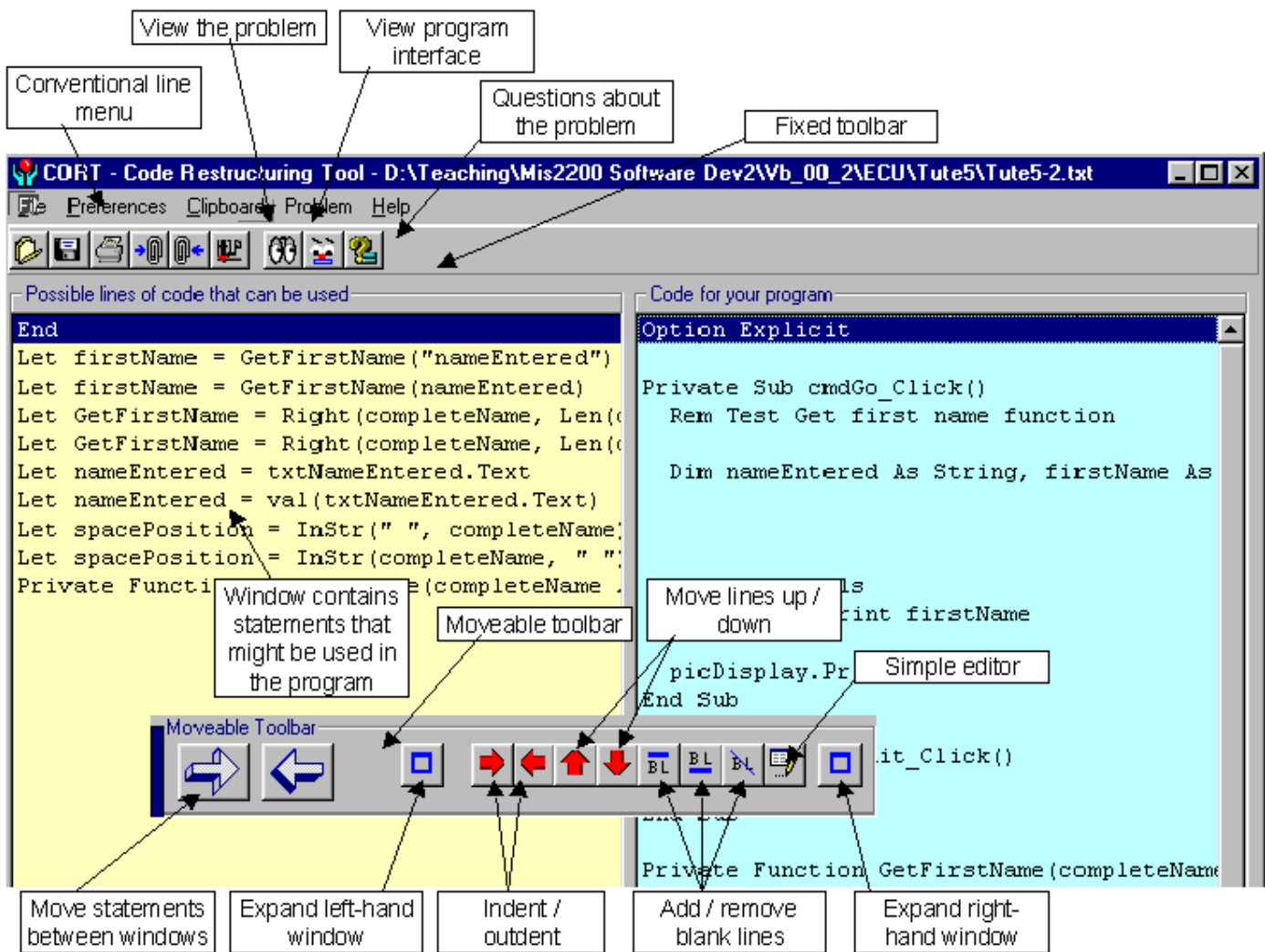The ways in which the CORT design supports the list of

Figure 2

225

A Tool to Support the Use of Part-Complete Solutions in the Learning of Programming

required features are described in table 1.

Use of CORT by Students

A student would typically use CORT as follows:

1. A student loads in a CORT file and the two windows display a part-complete solution to a problem together with possible lines to be used. There is a facility available for the contents of the two windows to be printed out.
2. The student can view the problem statement and the Visual BASIC solution interface by clicking on the appropriate tool bars on the moveable toolbar. The problem statement may have already been provided to the student in the form of a handout, however there will also be a facility to print it from within CORT.
3. The student moves certain lines from the left hand window to the right hand window in an attempt to complete the solution. Lines can be moved up or down, and indented or outdented in the right hand window. Some problems have too many lines in the left hand window, some of those lines being incorrect.
4. If necessary, the student can invoke a simple editor to amend, add or delete lines of code.
5. The student clicks on the appropriate button to copy the contents of the right hand window to the Windows clipboard.
6. The student invokes Visual BASIC and loads the file that contains the interface for the solution. This is in effect the Visual BASIC solution to the problem without the lines of code and was created by the tutor.
7. The student pastes the contents of the Windows clipboard into the Visual BASIC editor and tests the program to determine if it works correctly. Use is made of the trace and debugging facilities of Visual BASIC.

Feature	Support in CORT Design
Support for part-complete solutions to programming problems	The part-complete solutions will be automatically loaded into the right hand window and possible statements into the left hand window. Students will load these from a file.
A mechanism so that missing statements can easily be inserted into a part-complete solution and also moved within that solution	Two buttons in the middle of the screen will move lines between the windows. One line, or several lines will be able to be selected and moved across.
A facility so that students can add and amend lines of code	A simple editor will be provided so that students can add their own lines or amend existing lines.
For visual programming, a facility for students to easily view the target interface	Access to this feature will be via a button on the moveable toolbar.
A facility to access tutor created questions on the workings of the programming examples and to enter student answers	Access to this feature will be via a line menu item.
A facility to easily transfer a completed solution from CORT to the target programming environment	This is provided by a button on the main toolbar. A single click will copy the contents of the right hand window to the Windows clipboard ready for pasting into the Visual BASIC programming environment.
A facility to easily transfer programming code from the target programming environment back into CORT for further amendment	This is provided by a button on the main toolbar. A single click will paste the contents of the Windows clipboard into the right hand window, overwriting what is there.

Table 1: Features of CORT

These facilities provide an insight to the workings of the notional machine.

8. If the student finds a problem with the working of the program, they can return to CORT and make the changes to the code there.
9. The student repeats steps 3 to 8 until they have a working program.
10. The student answers the tutor's questions concerning the programming problem that they have just attempted.

Initial Student Feedback

CORT has been used for two semesters with both undergraduate and postgraduate students in the Faculty of Business and Public Management who are learning programming. The particular units are in software development and the language that the students learn is Visual BASIC.

Each week the students had to undertake completion programming exercises using CORT and after each problem they were asked to comment on the use of CORT for the particular problem that they had just finished. The data was collected on-line through the Web. Below are some of the comments that were received:

1. It's very helpful. I can see the interface of the program before actually running it.
2. Comments on CORT: I think CORT is a very useful tool to play around the codes. It saves me time copying and pasting.
3. Considering the increased workload as the semester progresses it is a bit of a relief that the exercises are much easier with the "fill in the gap" type format in CORT.
4. Without CORT, it's sure that I'll have a lot trouble with this particular problem, which focuses on arrays (a difficult topic). Thanks CORT...
5. CORT was useful in that the part solution helped to understand the logic of VB code
6. CORT is useful . However, I have used the unit text to try to understand the indentation format when writing the code. The directional keys are great for editing the code to meet the required format.
7. This was a challenge! I think that CORT is useful so long as I am not tempted to simply manipulate code until the program runs. If I were having to write programs from scratch I would use CORT so as to format and manipulate code and modules or sub procedures etc.

Conclusions

As can be seen from the above, the initial feedback on the use of CORT appears to have been favourable. We have found that students can undertake two or three small programming problems within a one hour tutorial whereas without CORT they could only undertake one such problem. Also, without using CORT students often never manage to successfully complete their assigned problems and this certainly affected their motivation.

By using CORT, students do not have to be concerned with the design of programming interfaces that considerably reduces the cognitive load in the initial stages of learning programming. Also, the reduction of "split attention affect" by labelling all the objects with their names has been very popular with the students.

The above has described a preliminary study of the use of CORT and it has been undertaken to determine its suitability and to fine tune some of its features. CORT can be used in several ways and three distinct methods have now been identified. These will be the subject of further research. The three methods are as follows:

1. **All** of the lines that are required to complete a program are made available in the left hand window of CORT. There are no extra lines displayed in the left hand window.
2. **All** of the lines that are required to complete a program are made available in the left hand window of CORT. There are also additional lines displayed in the left hand window that are not required within the program. The extra lines are similar to the required lines, however they are incorrect and act as "red herrings".
3. **Some** of the lines that are required to complete a program are made available in the left hand window of CORT. Other lines that are required for the program completion need to be keyed in by the student.

References

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8, 293-332.

Cook, C., Bregar, W., & Foote, D. (1984). A Preliminary Investigation of the use of the Cloze Procedure as a Measure of program Understanding. *Information Processing & Management*, 20(1-2), 199-208.

du Boulay, B. (1986). Some Difficulties in Learning to Program. *Journal of Educational Computing Research*, 2(1), 57-73.

Edward, N. (1997). *Development of a cost effective computer assisted learning (CAL) package to facilitate conceptual understanding*. Paper presented at the CAL97, University of Exeter, UK.

A Tool to Support the Use of Part-Complete Solutions in the Learning of Programming

Ehrlich, K., & Soloway, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. In J. Thomas & M. L. Schneider (Eds.), *Human Factors in Compter Systems* (pp. 113-133). Norwood, New Jersey: Ablex.

Fowler, W. A. L., & Fowler, R. H. (1993). A Hypertext Approach to Computer Science Education Unifying programming Principles. *Journal of Multimedia and Hypermedia*, 2(4), 433-441.

Hall, W. E., III, & Zweben, S. H. (1986). The Cloze Procedure and Software Comprehensibility Measurement. *IEEE Transactions on Software Engineering*, May 1986, 608-623.

Klare, G. R. (1974-75). Assessing Readability. *Reading research quarterly* (10), 63-102.

Lieberman, H. (1986). An Example Based Environment for beginning Programmers. *Instructional Science*, 14(3), 277-292.

Lisack, S. K. (1998, October 1-3, 1998). *Helping Students Succeed in a First programming Course: A Way to Correct Background Deficiencies*. Paper presented at the International Association for Computer Information Systems Conference, Cancun, Mexico.

Marcus, A. (1992). *Graphic Design for Electronic Documents and User Interfaces*. New York: ACM Press.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity to Process Information. *Psychological Review* (63), 81-97.

Norcio, A. F. (1980a). *Comprehension Aids for Computer Programs*. Paper presented at the American Psychological Association Annual Meeting, Montreal.

Norcio, A. F. (1980b). *Human Memory Processes for Comprehending Computer Programs*. Paper presented at the Cybernetics and Society, Cambridge, Massachusetts.

Norcio, A. F. (1981). *Chunking and Understanding Computer Programs*. Paper presented at the Human-Machine Systems Symposium, Boston, USA.

Norcio, A. F. (1982). *Indentation, Documentation and Programmer Comprehension*. Paper presented at the Human Factors in Computer Systems, Gaithersburg, Maryland.

Scholtz, J., & Wiedenbeck, S. (1992). The role of planning in learning a new programming language. *International journal of man-machine studies*, 37, 191-214.

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1), 59-89.

Thomas, M., & Zweben, S. (1986). The Effects of Program-Dependent and Program-Independent Deletions on Software Cloze Tests. *Empirical Studies of Programmers*, 138-152.

van Merrienboer, J. J. G., & Krammer, H. (1987). Instructional Strategies and tactics for the design of Introductory Computer programming Courses in High School. *Instructional Science*, 16(3), 251-285.

van Merrienboer, J. J. G., Krammer, H. P. M., & Maaswinkel, R. M. (1994). Automating the planning and construction of programming assignments for teaching introductory computer programming. In R. D. Tennyson (Ed.), *Automating Instructional Design, Development, and Delivery* (NATO ASI Series F, Vol. 119) (pp. 61-77): Springer Verlag, Berlin.

van Merrienboer, J. J. G., & Paas, F. (1990). Automation and Schema Acquisition in learning elementary computer programming. *Computers in Human Behavior* (6), 273-289.

Biography

Stuart Garner is a member of the school of Management Information Systems within the faculty of Business and Public Management at Edith Cowan University in Perth, Western Australia. He teaches in the areas of systems and software development and Web commerce development.