Learning Programming via Worked-examples

Siti Soraya Abdul Rahman

School of Informatics
University of Sussex
s.abdul-rahman@sussex.ac.uk

Benedict du Boulay

School of Informatics
University of Sussex
b.du-boulay@sussex.ac.uk

Keywords: Structure-emphasising strategy, completion strategy, cognitive load, learning styles

Abstract

Worked-examples play an important role in learning and problem solving and are crucial to cognitive skill acquisition. However students' learning styles are a factor in how much they are willing to expend serious effort on understanding worked-examples, with active students tending to be more impatient of them than reflective students. In employing worked-examples to teach programming effectively to both active and reflective students, we propose a worked-example format that combines a completion strategy with a structure-emphasising strategy. In this paper, we present two types of web-based interface supporting both the strategies and briefly discuss their specific interface designs.

1. Introduction

Worked-examples play an important role in learning and problem solving (Pirolli & Anderson, 1985; Chi, Bassok, Lewis, Reimann, & Glaser 1989) and are relevant to the acquisition of initial cognitive skills (Atkinson, Derry, Renkl, & Wortham, 2000), and hence play an important role in learning programming. Schema acquisition is one of the underlying processes in acquiring and managing such skills (Van Merrienboer & Paas, 1990). Several researchers have proposed different means to promote schema acquisition. For instance, Van Merrienboer et al. (1990) claimed that successful schema acquisition requires an investment of effort from students and this can be achieved by providing students with partial worked-example solutions that have to be completed (i.e. a completion strategy). Similarly, Quilici and Mayer (2002) reported that requiring students to abstract the underlying structural features of example problems (i.e. a structure-emphasising strategy) and so organize them into a generalised problem model facilitates the students' development of problem schemas.

In terms of learning style, students can be divided broadly into the active and the reflective (Felder & Spurlin, 2005). Both the completion strategy and the structure-emphasising strategy provide reflective students with more opportunity for *thinking things through*, to use Felder et al.'s (2005) phrase. By contrast, neither format provides an opportunity for active students to work with the examples dynamically by *trying things out*, again to use Felder et al.'s (2005) phrase. As a consequence, active students may become overwhelmed and may experience cognitive overload because they are forced into an uncongenial, more reflective style of learning. Note that, Van Merrienboer's (1990a) studies found little support for the idea that the negative effects of impulsivity (a feature of active students) could be compensated by an instructional strategy which emphasised program completion. Note also that students with low working memory capacity tend to prefer an active style of learning (Graf, Lin, & Kinshuk, 2008). Finally, instructional design based on cognitive load theory (CLT) argues for the careful utilisation of working memory capacity to encourage more effective schema construction (Sweller, Van Merrienboer, & Paas, 1998).

2. Worked-example format

In employing worked-examples to teach programming effectively to both active and reflective students, we propose a worked-example format (see Table 1) that combines the completion strategy (Van Merrienboer, 1990b; Van Merrienboer et al., 1990) with the structure-emphasising strategy

(Quilici et al., 2002). It is hypothesised that the combined format leads to better learning for active students and that reflective students would do no worse than with either completion strategy or structure-emphasising strategy alone. Using CLT, it is argued that structure-emphasising strategy decreases extraneous load and subsequently increases germane load because students are guided by plan-focused self-explanation prompts. A similar benefit should be achieved with the completion strategy. This is because, following the study of a structure-emphasising worked-example, students should be able to solve the completion task with less effort by mapping this task onto existing plan schemas.

Instructional principles from worked-example research (Atkinson et al.,2000)	**Worked-example format**	
	Structure-emphasising strategy (Quilici et al., 2002)	*Completion strategy* (Van Merrienboer, 1990b; Van Merrienboer et al., 1990)
Intra-example features	Place surface features deliberately so as to promote search for meaningful conceptual structures (Atkinson et al., 2000).	Worked-example solution steps left to complete are place strategically (Sweller et al., 1998).
Inter-example features	Example/problem pairs (Trafton & Reiser, 1993). That is, presenting structure-emphasising worked-example (Quilici et al., 2002) followed by a matched problem that has to be completed (Van Merrienboer, 1990b; Van Merrienboer et al., 1990).	
Self-explain worked-example	Structure-based (i.e. programming plans: Soloway, 1986) self-explanation guidance through plan-focused prompts.	▨

Table 1: Combined worked-example format

3. Web-based worked-examples interface

Table 1 describes some of the principles underlying the design of the format of the worked examples. A central notion is self-explanation. Self-explanation refers to the process of "generating explanations to oneself to clarify an example's worked-out solution" (Conati & VanLehn, 2000). In the view of Soloway (1986), "learning to program amounts to learning how to construct mechanisms and how to construct explanations".

3.1 Related work underlying the interface design

We adopt a "dissection" method (Kelley & Pohl, 1996) of explained worked-examples used in programming textbook, similar to that implemented in WebEx (Brusilovsky, 2001) - a web-based tool for learning from tutor-explained examples in a programming course. The WebEx interface comprises of a program example with bullets (white and green) appended to the left side of each line of program. A white bullet indicates that no explanation is available for that line; on the other hand when a green bullet is clicked, the interface displays a textual explanation for the chosen line. Instead of providing explanations for each line of program, as in the WebEx, we propose eliciting self-explanation on various instances of plan structures in the example solution.

SE-COACH (Conati et al., 2000) is a computer-based tutor that coaches self-explanation within the domain of Newtonian physics. Like SE-COACH, we intend to use a masking mechanism for the initial presentation of example solutions and to employ self-questioning (Webb, 1989) prompts. In the SE-COACH, different example parts are covered with grey boxes. As students uncover part of the example, the interface reveals some text or graphics. The interface appends a self-explain button only if the uncovered part contains information worth explaining. When students click on the button, the SE-COACH generates a prompt, which is designed to initiate self-explanation by means of motivating self-questioning (e.g. *“this choice is correct because…”*). In place of grey boxes, we use a collapsible button (as in the WebEx) to hide and unhide the plan structures. Our mechanism has four different rationales. First, it draws students’ attention to the underlying plan structure so that self-explanation can be promoted. Second, it helps students to abstract away the details of the plan structures linked to the programming problem. Third, plan names may well serve as cues to retrieve plan schemas for future problem solving. Finally, it encourages “structural awareness” (Quilici et al., 2002).

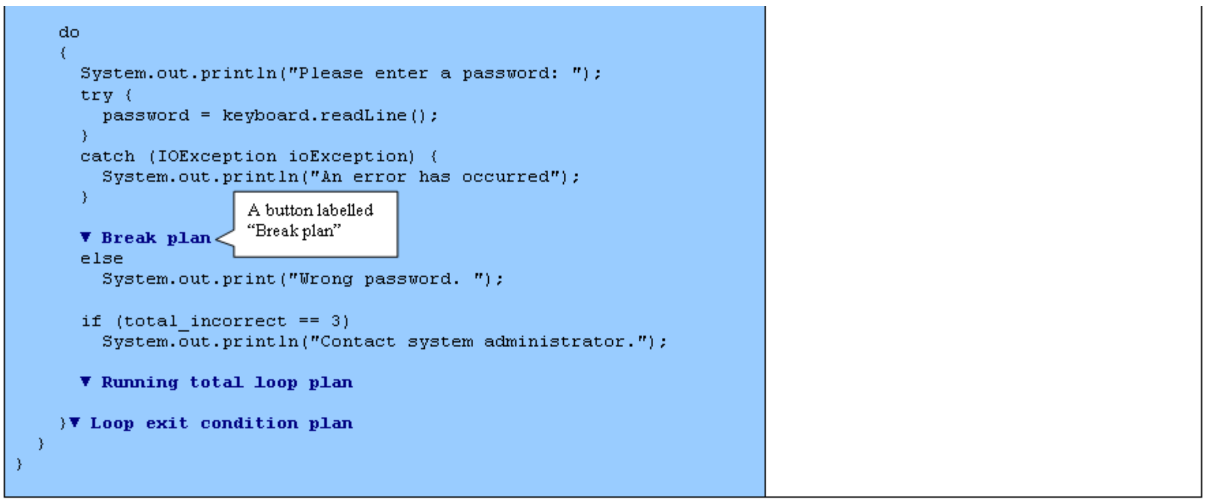
CORT (Code Restructuring Tool) developed by Garner (2007) supports the part-complete solution method (PCSM) of learning programming. The CORT interface consists of two windows, namely the left and the right windows. The right window contains the part-complete program whereas the left window contains lines of code to be used in the completion task along with extra lines that act as “distracters” (Garner, 2007). Some of the missing lines from the program however must be provided by students. To complete the program, students move the lines between the windows and rearrange the lines in the right window. When students have done with it, they can copy the program and paste into a program development editor and run it. In this respect, CORT gives freedom to students to solve the completion task. Nevertheless, this may cause students to come up with several different solutions to a programming problem, hence it is difficult to provide specific feedback to students on their final program solution. To alleviate this problem, we propose that the missing parts of the program ought to be left intact in the incomplete program fragment. Figure 3 further illustrates this idea.

Next sections further describe two types of interface, supporting structure-emphasising strategy and completion strategy.

3.2 Structure-emphasising strategy

The first type of interface, which we describe as supporting a structure-emphasising strategy, is designed to encourage students to abstract away the details of an example problem and to self-explain various instances of plan structures in an example solution. In particular, the interface provides a means by which students are able to construct self-explanations, guided by plan-focused prompts.

The interface presents students with a worked-example consisting of a programming problem and an example solution together with a list of programming plan names. Plans are generic program fragments that represent “stereotypic actions in a program” (Ehrlich & Soloway, 1984). Programming plans serve to highlight the structure of, and relationships between types of programming problems and types of programs (Soloway, 1985).



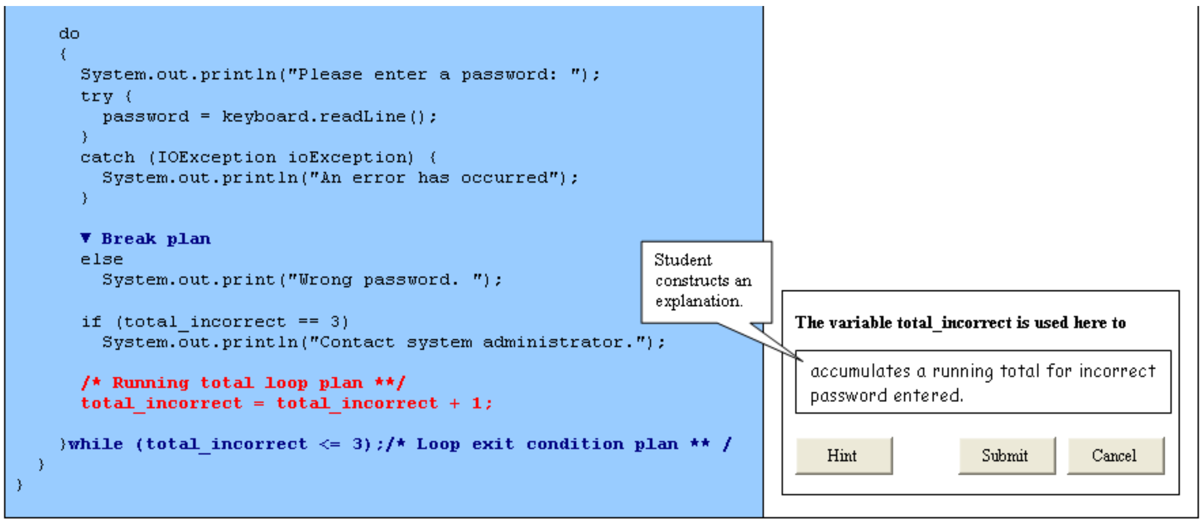
```
    do
    {
      System.out.println("Please enter a password: ");
      try {
        password = keyboard.readLine();
      }
      catch (IOException ioException) {
        System.out.println("An error has occurred");
      }

      ▼ Break plan
      else
        System.out.print("Wrong password. ");

      if (total_incorrect == 3)
        System.out.println("Contact system administrator.");

      ▼ Running total loop plan

    }▼ Loop exit condition plan
  }
}
```

Figure 1: Initial presentation of an example solution

The underlying plan structures are initially invisible (see Figure 1). Clicking on a down-pointing triangular button next to a plan name reveals its structure. A prompt dialog box immediately appears next to the plan structure presently explored to start a self-explanation exercise (see Figure 2).

Exploration of programming plans is fully under the student's control. That is, students can open several plans at the same time and in no particular order. The interface uses a different font colour for program code. Dark blue denotes various instances of plan structures in an example solution and red denotes the plan structure currently being explored (see Figure 2). In this way, students stay focused on one plan at a time.

The interface provides a text area for self-explanation input and a hint button to help students with the self-explanation exercise. No feedback is given to students' explanations. However, after a specified period of time, the interface presents complete descriptions of each of the plan structures to allow students to reflect on their previously generated explanations. The use of hints to aid self-explanation is similar to that used in the English Grammar Tutor (Wylie, Koedinger, & Mitamura, 2009). The tutor (within the domain of English articles) provides students with access to a series of on-demand hints for selecting an article and for explaining that article selection.



```
    do
    {
      System.out.println("Please enter a password: ");
      try {
        password = keyboard.readLine();
      }
      catch (IOException ioException) {
        System.out.println("An error has occurred");
      }

      ▼ Break plan
      else
        System.out.print("Wrong password. ");

      if (total_incorrect == 3)
        System.out.println("Contact system administrator.");

      /* Running total loop plan **/
      total_incorrect = total_incorrect + 1;

    }while (total_incorrect <= 3);/* Loop exit condition plan ** /
  }
}
```

The variable total_incorrect is used here to

accumulates a running total for incorrect password entered.

Figure 2: Visible plan structures and prompt

3.3 Completion strategy

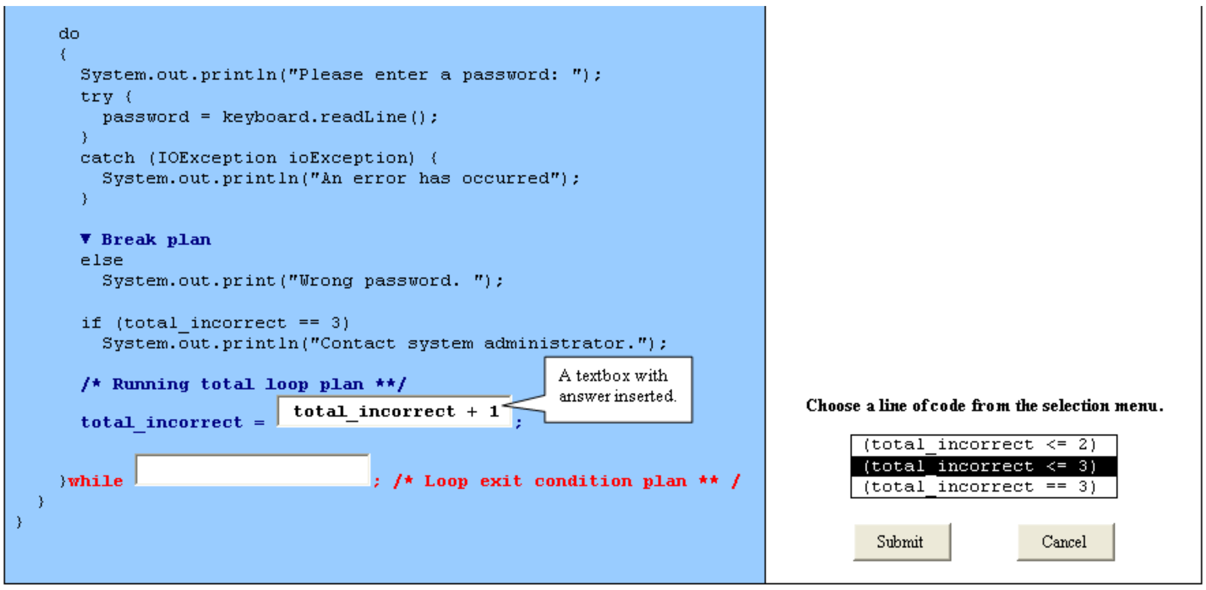
The second type of interface, which we describe as supporting a completion strategy, is designed to encourage students to complete partial code pertaining to a number of instances of plan structures in an example solution. The strategy requires students to study the partial code provided in the completion assignment otherwise they cannot correctly solve the task (Van Merrienboer, 1990b; Van Merrienboer et al., 1990). The strategy promotes "mindful abstraction" (Van Merrienboer et al., 1990).

The initial presentation of an example solution and code exploration for a completion strategy is very similar to that of a structure-emphasising strategy, other than presence of partial code in the example solution that has to be completed.

Clicking on a down-pointing triangular button next to a plan name reveals the partial code of the plan structure. The interface provides students with both (i) a menu to choose a line of code (see Figure 3) and (ii) a text box in which they have to insert suitable code to complete the program. Students receive feedback on the correctness of answers for:

- the line of code chosen from a menu. Using such a menu drastically simplifies the construction of feedback.

- the line of code inserted into a textbox. In this case, students' answer must match the correct answer.



```
    do
    {
      System.out.println("Please enter a password: ");
      try {
        password = keyboard.readLine();
      }
      catch (IOException ioException) {
        System.out.println("An error has occurred");
      }

      ▼ Break plan
      else
        System.out.print("Wrong password. ");

      if (total_incorrect == 3)
        System.out.println("Contact system administrator.");

      /* Running total loop plan **/

      total_incorrect = total_incorrect + 1 ;

    }while                    ; /* Loop exit condition plan ** /
  }
}
```

Figure 3: Completion task and a menu

4. Conclusion

We have presented a design for an interface aimed at helping active students gain more benefit from being exposed to worked-examples, while at the same time not disadvantaging reflective students. The proposed format is intended to moderate the effectiveness of the worked-example by combining a completion strategy with a structure-emphasising strategy. The paper has presented two types of web-based interface supporting the strategies, argued for the approach with reference to the research literature, and outlined their specific interface designs.

Apart from implementing the system, the next stage is to experiment with matched groups of active and reflective students so as to test whether the hypothesis underlying the approach can be supported. The basic idea will be to compare students who learn either via the completion strategy on its own, or via the structure emphasising strategy on its own, or via the proposed "paired-method" strategy of using both completion and structure emphasis. Groups will have the same time on task and both process data as well as pre/post data will be collected. The expectation is that active students will engage well with the paired method of interacting with the worked-examples and will show learning gains approaching those of reflective students using the same paired method. Active students will be expected to do worse than reflective students in the two groups exposed only to a single method of using worked examples. Finally reflective students using the paired method will show learning gains slightly better than reflective students using either of the single methods on their own.

5. References

Atkinson, R.K., Derry S.J., Renkl, A., & Wortham, D. (2000). Learning from examples: instructional principles from the worked examples research. *Review of Educational Research, 70,* 181-214.

Brusilovsky, P. (2001). WebEx: Learning from examples in a programming course. In Lawrence-Fowler, W. A. & Hasebrook, J (Eds.), *Proceedings of WebNet 2001- World Conference of the WWW and Internet* (pp.124-129), Orlando, Florida: AACE.

Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science: A Multidisciplinary Journal, 13,* 145-182.

Conati, S. & VanLehn, K. (2000). Toward Computer-based Support of Meta-Cognitive Skills: A Computational Framework to Coach Self-Explanation. *International Journal of Artificial Intelligence in Education, 11*, 389-415.

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 Computer Systems* (pp. 113-133). Norwood, NJ: Ablex Publishing Corp.

Felder, R.M. & Spurlin, J. (2005). Applications, Reliability and Validity of the Index of Learning Styles. *International Journal of Engineering Education, 21*, 103-112.

Garner, S. (2007). An Exploration of How a Technology-facilitated Part-complete Solution Method Supports the Learning of Computer Programming. *Issues in Informing Science and Information Technology, 4*, 491-501.

Graf, S., Lin, T., & Kinshuk, (2008). The relationship between learning styles and cognitive traits – Getting additional information for improving student modelling. *Computers in Human Behaviour, 24*, 122-137.

Kelly, A. & Pohl, I. (1996). C by Dissection: The Essentials of C Programming (3rd ed.), CA: Addison-Wesley.

Pirolli, P.L. & Anderson, J.R. (1985). The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology, 39*, 240-272.

Quilici, J.L. & Mayer, R.E. (2002). Teaching students to recognize structural similarities between statistics word problems. *Applied Cognitive Psychology, 16*, 325-342.

Soloway, E. (1985). From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research, 1*, 157-172.

Soloway, E. (1986). Learning to program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM, 29*, 850-858.

Sweller, J., Van Merrienboer, J.J.G., & Paas, F.G.W.C.(1998). Cognitive architecture and instructional design. *Educational Psychology Review, 10*, 251-296.

Trafton, J.G. & Reiser, B.J. (1993). The contributions of studying examples and solving problems to skill acquisition. *Proceedings of the 15th Annual Conference of the Cognitive Science Society* (pp. 1017-1022), Hillsdale, NJ: Erlbaum.

Van Merrienboer, J.J.G. (1990a). Instructional strategies for teaching computer programming: Interactions with the cognitive style reflection-impulsivity. *Journal of Research on Computing in Education, 23*, 45-53.

Van Merrienboer, J.J.G. (1990b). Strategies for Programming Instruction in High School: Program Completion Vs. Program Generation. *Journal of Educational Computing Research, 6*, 265-285.

Van Merrienboer, J.J.G. & Paas, F.G.W.C. (1990). Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice. *Computers in Human Behaviour, 6*, 273-289.

Webb, N.M. (1989). Peer Interaction and Learning in Small Groups. *International Journal of Educational Research, 13*, 21-40.

Wylie, R., Koedinger, K.R., & Mitamura, T. (2009). Is Self-Explanation Always Better? The Effects of Adding Self-Explanation Prompts to an English Grammar Tutor. In N.A. Taatgen & H. van Rijn (Eds.). *Proceedings of the 31st Annual Conference of the Cognitive Science Society* (pp. 1300-1305), Austin, TX: Cognitive Science Society.