Teaching and Assessing Programming Strategies Explicitly

Michael de Raadt, Richard Watson
Department of Mathematics and Computing
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
{deraadt, rwatson}@usq.edu.au

Mark Toleman
School of Information Systems
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
markt@usq.edu.au

Abstract
This paper describes how programming *strategies* were explicitly instructed and assessed in an introductory programming course and describes the impact of this curricular change. A description is given of how *strategies* were explicitly integrated into teaching materials and assessed in assignments and examinations. Comparisons are made between the outcomes of novices under the new curriculum and results of novices' learning under the previous implicit-only *strategy* curriculum, measured in an earlier study. This comparison shows improvement in novices' *strategy* application under the new curriculum.

Keywords: Strategies, introductory programming, curriculum.

1. Introduction
It is possible to distinguish programming *knowledge* from programming *strategies. Knowledge* involves the declarative nature (syntax and semantics) of a programming language, while *strategies* describe how programming *knowledge* is applied (Davies, 1993). Programming *strategies* involve the application of programming *knowledge* to solve a problem. A literary survey that defines these terms and highlights this distinction is given by Robins, Rountree, & Rountree (2003).

Programming *strategies* can be *plans* as described by Soloway (1985), or *patterns* (Wallingford, 1996), *algorithms*, etc., together with the associated means of incorporating these into a single solution. Soloway suggests programming *knowledge* is not a "stumbling block" (1986, p. 850) for novices and suggests teaching should reach beyond a focus on syntax and target programming *strategies*. Robins *et al* (2003) also suggest that the key to novices becoming effective lies in them learning programming *strategies* rather than acquiring programming *knowledge*.

Another distinction relevant to this study is found between programming *comprehension* (the ability to read and understand the outcomes of an existing piece of code) and *generation* (the ability to create a piece of code that achieves certain outcomes). Whalley et al. contend that "a vital step toward being able to write programs is the capacity to read a piece of code and describe it" (2006, p. 249) meaning that a novice must be able to comprehend a solution (and the *knowledge* and *strategies* within it) before they can generate a solution at the same level of difficulty. According to Brooks (1983), expert and novice programmers can be distinguished by how they undertake *comprehension*. During program *generation* an expert can rely on a tacit body of programming *plans* developed through solving past problems (Soloway, 1986), while novices are traditionally expected to conceive and apply *plans*, with varying degrees of success (Rist, 1991).

The Leeds group (Lister et al., 2004) attempted to isolate the cause of poor novice results measured by the McCracken group (McCracken et al., 2001). The Leeds group reported that many instructors attribute poor results to poor problem-solving ability in novices. The group attempted to create programming questions that required no problem-solving ability to answer. If novices succeeded in the test it would confirm that novices can successfully acquire programming *knowledge* and instructors could put this issue aside and focus their attention on improving *strategy* instruction. If novices failed this test, it would indicate a failure in programming *knowledge*. Results of the Leeds group study, and the BRACElet project (Whalley et al., 2006) that followed, showed that many novices exhibit a fragile programming *knowledge* and very few can demonstrate programming *strategy* understanding in a *comprehension* exercise. It is therefore important to consider both programming *knowledge* and *strategy* together in curricula.

When considering the problems novices are expected to solve in an introductory programming course, de Raadt, Toleman and Watson (2006) use a scale of problems with three levels being "system", "algorithmic" and "sub-algorithmic". The simplest of these is sub-algorithmic level problems, with solutions that do not involve algorithms or system design. Examples of problems of this scale include avoiding division-by-zero, achieving repetition until a sentinel is found, and so on. *Strategies* used to solve problems at this level are particularly relevant to novices in their initial exposure to programming, yet these *strategies* are also a fundamental part of solving problems at any level.

1.1 Previous Work
1.1.1 Initial Study
A previous study (de Raadt, Toleman, & Watson, 2004) found weaknesses in a traditional curriculum used in teaching an introductory programming course to novices where *strategies* were not taught explicitly. Instead, students were expected to learn *strategies* implicitly by seeing examples and solving problems. Students who participated in the study were asked to create a solution to a simple averaging problem. A number of common flaws were detected when students' solutions were scrutinised under Goal/Plan Analysis (Soloway, 1986).

Participating students were not consistently able to:

- initialise sum and/or count variables,
- use a correct looping *strategy* for the given problem,
- guard against events such as division by zero, or
- merge *plans* that should be achieved together.

Students, on average, were only able to demonstrate application of 57% of the *strategies* required for a complete solution. These flaws implied weaknesses in the curriculum being delivered to the students at the time.

1.1.2 Pilot Study

Educational research experiments (Biederman & Shiffrar, 1987; Reber, 1993) have shown that explicit instruction can be more powerful than implicit-only instruction, so it was proposed that programming *strategies* be taught explicitly. A number of attempts have been made to represent sub-algorithmic *strategies* in a form that can be presented to novices; with most recent studies focussing on *patterns* (Muller, Haberman, & Ginat, 2007; Porter & Calder, 2003; Wallingford, 2007). For this study *plans* were chosen as they can be used with multiple paradigms, including the object paradigm. *Plans* can be expressed simply, particularly at a sub-algorithmic level. de Raadt, Toleman and Watson (2006) showed that *plans* suitable for novice instruction at a sub-algorithmic level can be identified in solutions produced by expert programmers. Although *plans* were chosen as a *strategy* representation, the focus of this study is on instruction of *strategies*, and this could be tested with any form of *strategy*.

Before introducing programming *strategies* in a full introductory programming course, a pilot study was undertaken (de Raadt, Toleman, & Watson, 2007). A controlled experiment was conducted that compared two curricula: one including programming *strategies* explicitly and a traditional curriculum that required students to learn *strategies* implicitly. Each curriculum was delivered over a weekend with students who had no programming experience. The experiment showed that it is possible to incorporate *strategies* explicitly into a curriculum. At the end of the weekend, participants were asked to generate solutions to three problems including the averaging problem used in the initial study and two similar problems. Experimental participants, who had been exposed to explicit *strategy* instruction, used *strategies* in their solutions, although no significance was proven as the number of participants was small. After the weekend courses, control and experimental participants were interviewed to probe their understanding of the *strategies* they were exposed to, either implicitly or explicitly. Participants were asked to describe their understanding of the problem statements. They were asked to lead the interviewer through their solution, describing each part. Participants were also asked say if they felt their solution would solve the problem. Participants exposed to explicit *strategy* instruction used terms from a *strategy* vocabulary to describe their solutions and showed greater confidence than those exposed to a traditional curriculum.

After the pilot study *strategies* were introduced into an actual introductory programming course held over a semester. A larger set of programming *strategies* was expressed and incorporated into teaching materials, lectures, formative and summative assessments and the examination.

The main testing approach used to gauge *strategy* application in previous studies was Goal/Plan Analysis (Soloway, 1986). With novices, this approach is limited to analysing solutions generated at or near the end of an introductory programming course. After the pilot study it was proposed that analysis of *strategy* skill should be conducted in more flexible ways throughout the course by taking the ideas inherent in Goal/Plan Analysis and using them to assess student work in assignments and examinations. The following are ways *strategies* were incorporated in assignments and examinations.

- Encouraging students to use particular *strategies* when generating solutions for assignments
- Awarding credit for application of *strategies* in assignment marking criteria
- Using problems that focus on programming *strategies* as part of the final examination
- Analysing examination solutions in a Goal/Plan-Analysis-like manner

Awarding credit for applying *strategies* in assessments was also done to encourage students to value this component of programming and devote more effort to learning it.

1.2 Participants and Setting

Participants in this study were novices studying in a first-year introductory programming course. The course is delivered to students on-campus (approximately 40% of the student cohort) and students studying externally (via distance education, potentially anywhere in the world). On-campus students are expected to attend two one-hour lectures followed later in the week by a one-hour tutorial (in a normal classroom) and a two-hour practical class with computers. External students study independently by reading the same written materials, accessing lectures online, and undertaking tutorial and practical exercises. The course runs twice a year, each year, but this study will focus on the results of three particular cohorts.

Table 1. Cohorts involved in the study

Semester	N	Student Location	Strategies
2003	42	on-campus	implicit-only
2005	36	on-campus, external	explicit
2007	45	on-campus, external	explicit

Table 1 shows which cohorts were the focus of comparisons in this study. The initial study, reported in (de Raadt, Toleman, & Watson, 2004), was conducted 2003 in class with on-campus students only. The later cohorts also included students studying externally as testing was conducted as part of the examination; this also kept participant numbers consistent between comparisons during a period of decline in student numbers. In each cohort, participants included school leavers and mature-aged students. Students were from a range of discipline areas but were primarily IT and Engineering students. The entry standard was consistent throughout the period of study. The mix of students has varied with more non-computing students undertaking the course in later years.

Apart from the inclusion of explicit *strategy* instruction (described in detail in section 0) the curriculum was unchanged between the offerings listed above. The course follows a procedural paradigm using the C programming language teaching topics including functions, data storage, selection, iteration, arrays, I/O and recursion. The instructor was the same in all instances.

1.3 Research Questions

This section is divided into two parts related to two perspectives (integration and impact) taken when conducting this study. This two-perspective structure is mirrored in the Methodology, Results and Discussion sections of this paper.

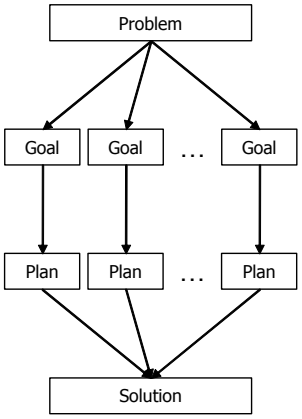
1.3.1 Integration Questions

The first two questions consider the possibility of instructing and assessing programming *strategies* explicitly. Although this was established on a smaller scale in the pilot, it needs to be tested with a complete curriculum in a full-scale introductory programming course.

> ***RQ1.*** *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*
>
> ***RQ2.*** *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

1.6.1 Design

An expert programmer will take time to properly design a solution. It is tempting to jump to implementation, but often, without a reasonable design, a programmer can waste time correcting a poor implementation and take far longer than if they had spent a small amount of time on design first.



From a *problem statement* a programmer will identify the *goals* that need to be achieved. These goals can usually be found through a careful reading of the problem statement.

STRATEGY

When the goals of the problem have been identified, a programmer can choose appropriate *plans* that satisfy goals. A plan is a small, independent strategy that the programmer has applied in a past solution. During this course we will be covering programming knowledge and also the strategies that you can use to apply this knowledge. Look for the STRATEGY sidebar to differentiate parts of this book that cover strategies.

Once plans have been identified they need to be combined together to form a solution. Plans can be combined together in three possible ways.

- **Abutment**
  Placing the plans one after another in the correct sequence that will solve the problem.
- **Merging**
  Integrating plans so that common parts are performed together
- **Nesting**
  Placing one plan inside another plan

Figure 1. Introduction to *strategies* from the Study Book

1.3.2 Impact Question

The third question relates to the effect of introducing explicit programming *strategies* to novice programmers. This question will be answered by analysing novice performance on assessments in the course and comparing this to the baseline performance described by the initial study (de Raadt, Toleman, & Watson, 2004).

> ***RQ3.*** *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

2. Integrating Strategies

Over the two-and-a-half-year period between the second half of 2005 and the end of 2007, programming *strategies* have been incorporated into the curriculum of an introductory programming course.

Programming *knowledge* was presented in a similar manner to the traditional curriculum used. *Strategies* are interwoven through the course in an explicit manner. In the beginning of the course the distinction between *knowledge* and *strategies* is presented. Figure 1 shows an initial description of *plans* as *strategies* within a description of the programming process. *Strategies* are a part of the curriculum and testing students' *strategy* skills forms part of the assessment. Students are informed of this at the outset.

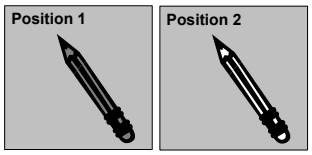
Written materials provided to students include notes for each module of the course and exercises for each week. Students are encouraged to read the written materials before attending or listening to lectures provided online (with audio for external students). The lectures complement the written materials and allow opportunities for questions and further explanations. Each week students are expected to undertake written and computer-based exercises, in tutorials and practicals, to reinforce the material for the week.

The following sub-sections describe how programming *strategies* were explicitly incorporated into written materials, lectures, weekly exercises, assignments and in the course examination.

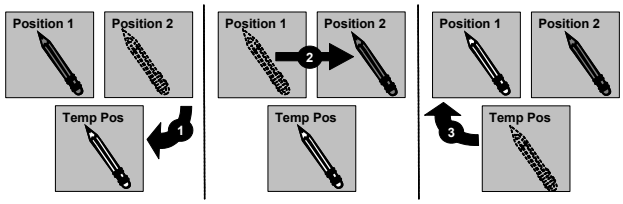
Plan 6. Triangular Swap Plan

This plan requires an understanding of variables and the assignment operator.

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now, imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int firstPosition  = 5; // First position containing value to swap
      int secondPosition = 6; // Second position containing value to swap
      int tempPosition;       // Temporary position for swap

      // Output the numbers after the swap
      printf("Before Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);

      // Swap the two numbers in a triangular swap
      // 1. Copy the value from the second position to temp
      tempPosition = secondPosition;

      // 2. Copy the value from the first position to the second
      secondPosition = firstPosition;

      // 3. Copy the value from the temp position to the first
      firstPosition = tempPosition;

      // Output the numbers after the swap
      printf("After Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```

Here is the output of the above program.

```
Before Swap...
First: 5, Second: 6
After Swap...
First: 6, Second: 5
```

The above results show the values are swapped and not duplicated.

Figure 2. An example of a *plan* from the Strategy Guide

2.1 The 'Strategy Guide'

The major component of written material provided to novices in the course is referred to as a 'Study Book'. More detail about the Study Book modules is given in section 2.2 below. At the end of the Study Book two appendices are given: one is a syntax guide and the other collects together all the *strategies* that are covered in the course. This 'Strategy Guide' is available online (de Raadt, 2008).

The Strategy Guide begins by defining how *strategies* can be integrated. *Abutment*, *nesting* and *merging* are discussed in this introduction. Each *strategy* is then described as either a *plan* or, in the case of some later *strategies*, as a basic algorithm. An example is given in Figure 2. The programming *knowledge* required to apply each *plan* is stated at the beginning of each *plan* description. Examples and diagrams are provided for most *strategies*. The Strategy Guide forms a resource for novices studying in the course, and possibly after they have completed the course. All *strategies* assessed in assignments and the examination can be found in this guide; students are told this at the beginning of the course and again before the examination. *Strategies* are addressed individually in context within the modules of the Study Book and lectures.

The Strategy Guide contains 18 *strategies* ranging in scale from very simple *plans* such as finding an average, through several sub-algorithmic *plans* such as a triangular swap (see Figure 2 for this example), and on to some algorithmic *strategies* such as sorting. The *strategies* currently in the Strategy Guide are listed below.

1. Average plan
2. Divisibility plan
3. Cycle Position plan
4. Number Decomposition plan
5. Initialisation plan
6. Triangular Swap plan
7. Guarded Exception plans (including Guarded Division plan)
8. Counter-Controlled Loop plan
9. Primed Sentinel-Controlled Loop plan
10. Sum and Count plans
11. Validation plan
12. Min/Max plans
13. Tallying plan
14. Search algorithm
15. Bubble Sort algorithm
16. Command Line Arguments plan
17. File Use plan
18. Recursion plans (single- and multi-branching)

2.2 Explicit Incorporation in Written Notes

Within the 12 modules of the Study Book, programming *strategies* are introduced after presenting the programming *knowledge* applied in each *strategy*. In this context the *strategies* show immediately how the *knowledge* can be applied, which, in its purest sense, is the nature of a *strategy*. This is followed by a code example showing the *plan* applied. For instance, the Triangular Swap plan is shown after students cover *variables* and *assignment* as programming *knowledge* components. This takes place in the third module, covered during the third week of the course. This *plan* is discussed in lectures, reinforced in tutorial and practical exercises and assessed in assignments and in the examination. The Triangular Swap plan appears again when the Bubble Sort Algorithm is presented in a later module of the course. This demonstrates how identifying *strategies* and creating a vocabulary for *strategies* allows instructors to use this vocabulary, and in doing so, reinforce *strategies* when they appear later in the course.

In the Study Book a sidebar down the left is used to visually distinguish parts covering programming *strategy* from other parts of the Study Book.

2.3 Explicit Incorporation in Lectures

During lectures, *strategies* are presented and discussed after relevant programming *knowledge* content had been covered. Lectures are presented in person to a class of on-campus students. The lecture is also recorded and the slides and audio are presented together and placed on the course website.

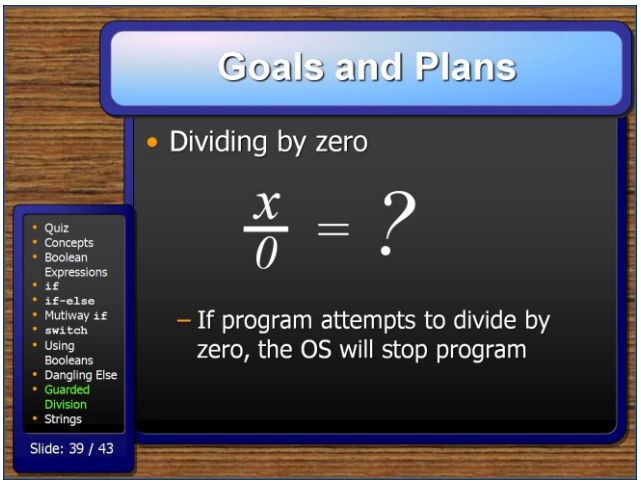


Figure 3. Example of a lecture slide showing the Guarded Division plan

The example shown in Figure 3 is one of a number of related slides that discuss the Guarded Division plan. On the left of the slide the outline of the lecture is shown and the current topic, 'Guarded Division', is highlighted. Observe that much of the previous content of the lecture has covered programming *knowledge*. Before a guarded division can be applied, novices must be aware of the `if` statement and the division operator (covered in a previous module). Students are shown how to apply this plan. This *strategy* is reinforced in the tutorial class held later that same week and is assessed in assignments and has been assessed in the examination.

2.4 Strategies in Tutorial and Practical Exercises

Programming is practiced in tutorial and practical classes. Exercises for these classes are listed in the Study Book following the content of each module. Prior to adding *strategy* content explicitly, the following exercise was given as an example.

> *Write a program that will allow the user to enter words. Use the* `%s` *format sequence in a* `scanf()` *call to capture each word one at a time. Find the length of each word using* `strlen()`*. To end the user input, the user will enter the string* `"end"`*. At the end of the program, output the count of words and the average length of the words.*

This example demonstrates how novices were expected to learn programming *strategies* implicitly in order to solve problems. The problem statement describes what needs to be achieved, but does not suggest how a solution should be constructed, and no strategy to solve the problem had been given in previous instruction.

Computer Exercises

8 Write a program that will allow the user to enter words. Use the **%s** format sequence in a **scanf()** call to capture each word one at a time (this will skip whitespace between words). You don't have to keep the user inputs in memory; you only need to deal with each word one at a time. Create an array with 256 characters for the input word. Set the maximum word size as a constant.

Find the length of each word using **strlen()**. To end the user input, the user will enter the string "end" (you will have to use **strcmp()** to test for this). You will need to include **string.h** to use these functions. Set the sentinel word as a constant.

At the end of the program, output the count of words, the total number of letters and the average length of the words. Be sure to use a sentinel controlled loop and guard the calculation of the average word length. Keep all numeric values as integers.

Your program should work if several words are entered before the sentinel, or if the sentinel is entered as the first input. Test your program by entering "end" as the first word. Try entering more than one word per line of input.

Figure 4. Example exercise requiring the Sentinel-Controlled Loop and Guarded Division plans. Highlighting (added for this figure only) shows *strategy* content

As a contrast, a new version is shown in Figure 4 above. In the new version students are given the same initial requirement with a few programming *knowledge* embellishments (such as the size of an array). Following this, in the third and fourth paragraphs of the problem statement, *strategy* instructions are given. Students are expected to use a Primed Sentinel-Controlled Loop to achieve repetition; this *plan* is named and its use is directed. The students are also reminded to guard the division when calculating the average. At this stage students are expected to know what a sentinel-controlled loop is and how to achieve a guarded division. This problem relies on students possessing a vocabulary that includes the term 'sentinel', which is used to define the value that, when encountered, will stop the repetition.

13. Fill in the blanks in the following code which swaps the values of two character variables and then outputs the variables new values.

```
#include <stdio.h>

int main() {
      char letter1 = 'a'; // First letter
      char letter2 = 'b'; // Second letter
      char temp = '-';    // Temporary position

      // Swap the two letters in a triangular swap
      
      
      

      // Output the letters
      
}
```

Figure 5. Example exercise from Module 3 requiring Triangular Swap plan

The example shown in Figure 5 requires students to apply a Triangular Swap plan to swap two character values. The *plan* name is mentioned explicitly in the code (in a comment) and three blanks imply the use of the triangular swap. Later in the course this *strategy* is used again in an exercise where students write a function that takes two pointers and orders the values to which they point.

Computer Exercise s

6. Copy the *Guarding Division* function example from page 15 that will calculate an average. Add a **main()** function that will call the **average()** function. It should still work when the value passed to **count** is zero.
   6.1 Remove the guarding **if-else** statement so all that remains in the function is the **return** statement. Now test the function sending zero as the value of **count**. When the program is compiled and run, the operating system should shut the program down and display an error.
   6.2 Restore the guard to the function and test that it works correctly again.

Figure 6. Example exercise from Module 5 testing the Division by Zero plan

Figure 6 contains an example of an exercise that asks students to experiment with the Guarded Division plan. This exercise encourages novices to experience the consequences (a program crash) resulting from dividing by zero. Through this, novices will hopefully come to appreciate the necessity of protecting the division with a guard.

Students are deliberately led to practise application of particular *strategies* for these problems in the same way that an instructor might encourage students to use a particular language construct, such as a **for** loop. In the examination, students are expected to apply required *strategies* without being led in this manner.

2.5 Assignment Instructions

As well as being introduced explicitly into instructional materials, programming *strategies* also became assessable in the course. Sections 2.5 to 2.8 describe how programming *strategies* have been included in assignment instructions and marking criteria as well as how examinations have been designed and marked to include testing of strategy-related abilities.

When teaching *strategies* explicitly, the challenge for instructors is to create problems that focus on particular programming *strategies*. Achieving this allows novices to demonstrate specific *strategies* in assignments and the examination.

- In your program, create the following functions.

…

```
void decryptEncryptLine(int shift);
```

  - This function will shift alphabetic characters by the amount of the shift. The function performs in the same manner for encryption and decryption. If the shift is a positive amount, this will shift characters forward (encrypt characters) and if negative it will shift them back (decrypt characters).
  - The function will input and process each character one at a time until a newline character is detected. Use a **primed sentinel controlled loop**. Do not try to store or process entire lines.

Figure 7. Extract from assignment instructions highlighting the requirement for a specific programming *strategy*

In assignment instructions students are given tasks that require them to apply specific programming *strategies*. Figure 7 above is an extract from an assignment's instructions where students are asked to use a Primed Sentinel-Controlled Loop to input characters entered by a user until the end-of-line is encountered.

2.6 Assignment Marking Criteria

As well as requiring specific *strategies* to be applied in the creation of solutions, the marking schema used to evaluate solutions also explicitly includes references to specific *strategies*.

In the course described here students participate in electronic peer-review as part of each assignment. Marking schema are constructed well in advance and released as part of the assignment instructions. Students are therefore aware of how their submission will be judged before they submit. They can see that they will receive marks for applying specific programming *strategies*. Being involved in peer-review, students are also expected to be able to judge if a peer-student has correctly applied a specific *strategy* where required by a criterion.

…

Check that no variables are declared outside functions. This does not include global constants.

☐ **A Primed Sentinel Controlled Loop is used to process menu options in the `main()` function**

The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters the quit option in the first instance, the loop body should not be entered.

☐ **A Primed Sentinel Controlled Loop is used to gather characters for input until the end of a line in the `decryptEncryptLine()` function**

The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters a blank line, the loop body should not be entered.

☐ **Code is indented consistently and no line is longer than 80 characters**

…

Figure 8. Extract from the marking scheme showing *strategies* are required in the solution for a programming assignment

Criteria relating to programming *strategies* are mixed with other criteria in each marking scheme. Figure 8 is an extract from the marking scheme for the same assignment that was used in the previous section.

2.7 Examination Questions

Questions in the examination are designed to separate ability in *knowledge* from *strategy* and ability in *comprehension* from *generation*. By combining these aspects, four types of question can be defined as shown in Figure 9.

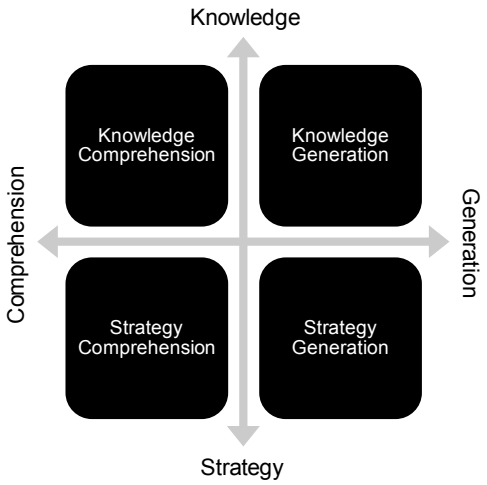


Figure 9. Four types of examination questions based on novice instruction aspects

Targeting questions to one of these four areas is not always simple. Some questions may stray over the boundaries between areas. The focus of the question can be reinforced by criteria used to award marks (see section 2.8).

2.7.1 Knowledge-Comprehension Questions

To test *knowledge* and *comprehension,* an examination question must focus primarily on language syntax skills but not ask the novice to generate any code. The question should test that the student understands an example shown to them, possibly by simulating how the code would be executed. A knowledge-comprehension examination question is shown in Figure 10.

QUESTION 1 **(10 marks, 12min)**

What will the following output?

```
#include <stdio.h>

int testFunc(int *ptr, int num);

int main() {
      int x=7, y=3, z=5;
      printf("%i %i\n", x, y);
      z = testFunc(&y, x);
      printf("%i %i %i\n", x, y, z);
}

int testFunc(int *ptr, int num) {
      int temp;
      printf("%i %i\n", *ptr, num);
      temp = num;
      num = *ptr;
      *ptr = temp;
      printf("%i %i\n", *ptr, num);
      return num + (*ptr);
}
```

Figure 10. A Knowledge-Comprehension examination question

2.7.2 Knowledge-Generation Questions

Knowledge-generation questions should require novices to generate code but not solve a problem requiring any programming *strategies*. The question should instead prompt the novice to create code that demonstrates their understanding of specific language constructs. An example of such a question is given as **Error! Reference source not found.**.

QUESTION 4 **(10 marks, 17min)**

Write a `main()` function that input an integer from a user and then use a `switch` statement to respond to the user's input with one of the following outputs:

Where `0` is entered, output `hello`
Where `1` is entered, output `bye`
Where any other value is entered, output `invalid`

Figure 11. A Knowledge-Generation examination question

2.7.3 Strategy-Comprehension Questions

Strategy-comprehension questions are perhaps the most difficult to define. These questions must test the *strategy* potential of a novice without asking them to generate any code. Possible ways to achieve this include the following.

- Asking novices to identify or describe *strategies* used in a given solution
- Asking novices to relate common *strategies* applied across multiple solutions
- Asking novices to identify how a *strategy* has been incorrectly applied in, or is absent from, a solution

In Figure 12 we see an example of a strategy-comprehension question that asks the novice to identify the strategy-related error in the code and state how the error could be corrected. The error can occur when the argument count has a value of zero, which would cause a division by zero. There is no guard to protect against this. To remedy this problem the student should apply a guard against division by zero. The exact 'Guarded Division' terminology is not critical if the novice can express this solution using other words.

QUESTION 5 **(5 marks, 18min)**

The following function contains a logic error. In a few words, describe what the error is **and** how you would remedy the error. Do not re-write the whole function.

```
int getAverage(int sum, int count) {
      return sum/count;
}
```

Figure 12. A Strategy-Comprehension examination question

2.7.4 Strategy-Generation Questions

Strategy-generation questions are probably what most instructors think of when they write a *generation* question for an examination. Such problems were designed to allow

QUESTION 7 **(20 marks, 24min)**

Write a function, using the following prototype, which will prompt the user and read in a valid positive integer. If the user enters invalid input, or a negative integer, the function will tell them their input was invalid and prompt them to enter another value. The function will repeat this until the user enters a valid input.

```
int getValidPositiveInteger();
```

For your reference, the following lines of code will clear the standard input stream.

```
scanf("%*[^\n]");
scanf("%*c");
```

QUESTION 8 **(20 marks, 24min)**

Write a `main()` function that will read in integers and output their average. Input will be gathered using the `getValidPositiveInteger()` function as described above (do not re-write that function). Stop reading when the value 99999 is entered (this is not to be used as an input).

Figure 13. A Strategy-Generation examination questions

novices to apply specific *strategies* they have learned in the course.

Figure 13 gives an example of two questions that formed a series from the S2, 2007 examination. The first question asks the novice to demonstrate a Validation plan. The Validation plan involves a Sentinel-Controlled Loop plan where a valid input is the sentinel. The second question in Figure 13 is essentially the same classic averaging problem, defined by Soloway (1986), and used in the initial study (de Raadt, Toleman, & Watson, 2004). This question requires novices to apply the following *plans*, each of which is covered explicitly in the course.

- Primed Sentinel-Controlled Loop plan
- Sum plan
- Count plan
- Guarded Division plan
- Average plan
- Output plan

2.8 Marking the use of Strategies in the Examination

When assessing the use of *strategies* in an examination it is critical that the marking scheme does not fall back on syntactical measures. The marking criteria for *strategy* related questions should seek the application of specific *strategies* or *comprehension* of those *strategies*. Strategy-generation questions should target specific *strategies* and the marking scheme for these questions should award marks where the required *strategies* have been applied, rather than for syntactical correctness.

Distinguishing how knowledge-related and strategy-related questions are marked forces a greater focus on particular areas from Figure 9 at the beginning of section 2.7.

3. Methodology

The comparison described in this paper can be considered from two perspectives, which can be related back to the research questions stated earlier:

- to test the possibility of explicitly incorporating and assessing programming strategies in an actual introductory programming course (RQ1 and RQ2); and
- to measure the impact of explicit programming strategy instruction and assessment on novices by comparing results produced under the new curriculum with benchmark measurements from the initial study (RQ3).

The method for achieving these aims is described in the following sub-sections.

3.1 Integration

The first and second research questions (RQ1 and RQ2) raised in section 1.2 consider the possibility of integrating *strategy* content into an actual introductory programming course. The success of this integration, drawing on examples presented earlier, is discussed in section 4.1. Observations are made on student response to the newly incorporated materials and assessment.

3.2 Impact

The third research question (RQ3) seeks to measure impact of the new curriculum relative to curriculum measured in the initial study (de Raadt, Toleman, & Watson, 2004). Students who participated in the initial study had studied using a curriculum that required them to learn *strategies* implicitly. In the initial study students were asked to create a solution to a classic averaging problem. Several *strategy* gaps were detected in student solutions indicating flawed understandings of the required *strategies*. Of particular interest was the lack of application of the Guarded Division plan.

Comparison of performance under the new curriculum with the benchmark performance was achieved through two examination questions. One question was included in the examination that followed the first integration of explicit programming *strategy* instruction in the second half of 2005 and another from an examination at the end of 2007. Results of these two examination question comparisons are shown in section 4.2.

3.2.1 Guarded Division Problem (2005 Examination)

One of the major flaws in novice *strategy* skill, detected in the initial study, was poor use of guarded division. A 2005 examination question shown as Figure 12 (section 2.7.3) is a strategy-comprehension question that targets the Guarded Division plan. This question yields either a correct or incorrect response. Student responses to this question were analysed and compared to application of Guarded Division in the initial study.

3.2.2 Averaging Problem (2007 Examination)

A 2007 examination question shown as Question 8 in Figure 13 (section 2.7.4) was a strategy-generation question that repeated the averaging problem given to novices in the initial study. Solutions to this question were analysed using the same approach as used in the initial study. Eight features were analysed in student solutions: seven *plans*, and the correct merging of *plans*. The presence or absence of each of these features was checked in all attempts. The features measured were as follows.

- Initialisation of a sum variable
- Initialisation of a count variable
- A Sum plan in a Primed Sentinel-Controlled Loop
- A Count plan in a Primed Sentinel-Controlled Loop
- A guard against division by zero
- An Average plan
- An Output plan
- Merging of the Sum and Count plans inside the Primed Sentinel-Controlled Loop

Strategies were judged as being either present or absent in solutions. For more detail on how these features can be

identified in a solution, see (de Raadt, Toleman, & Watson, 2006).

The circumstances surrounding the initial testing were slightly different to a final examination. The initial study was conducted under examination-like conditions (students were not permitted to talk to each other or use resource materials), but in tutorial classes during the course. Final examinations are held at the end of the course, giving students more time between exposure and testing of the necessary *plans*. These differences need to be kept in mind when comparing performance between these tests.

3.2.3 Avoiding Bias

Neither of these two specific questions had been used in the course prior to the examinations. The closest problem resembling the averaging problem was the average word length exercise given in practicals and shown in Figure 4 (section 2.4). The course materials covered each of the required *strategies*. Students had opportunities to practice each of the required *strategies*. These *strategies* were not emphasised more than any other *strategies* taught in the course.

In the two examination questions, students are not led to use any specific *strategies*; they are expected to have learned which *strategies* to apply at this stage (during the exam).

4. Results

Results are presented below, again divided by the two perspectives used earlier. First the success of integrating programming *strategies* in an actual introductory programming course is discussed. Specific strategy-related responses elicited under the traditional and new curriculum are then compared.

4.1 Integration

Integrating explicit *strategy* instruction and assessment into an actual introductory programming course was achieved. The examples of curricular materials and assessment items shown in section 0 demonstrate how this was achieved.

Although it is not scientific, some observations can be made. Perhaps the most arduous part of integrating *strategies* explicitly was in conceiving well focused assessment items. It is challenging to create problems that required students to apply specific *plans*, while maintaining interesting problems. Even so, a set of problems was developed to assess *strategy* skill in assignments and examinations.

Students accepted the new instruction as part of the course; no student protested against the inclusion of *strategies* as legitimate content. As each new cohort undertook the new curriculum, they were not aware that it was different to the traditional curriculum that preceded it. Students did not protest against having their *strategy* skills assessed. As mentioned earlier (see section 2.6), assignments involved peer review, so students were being asked to evaluate the work of their peers. Students were asked to complete reviews that required them to judge the presence or absence of *strategies* in the work of their peers.

4.2 Impact

Two specific questions were used to compare *strategy* skill under the previous and new curricula. The questions were drawn from two examinations, one which took place at the end of 2005 after the first instance of the course to include explicit *strategy* instruction, and one in the most recent instance at the end of 2007.

4.2.1 Guarded Division Problem (2005 Examination)

During the initial study a particularly poorly applied *plan* was the Guarded Division plan, with only four students out of 42 applying this *plan*. In the S2 2005 examination, under the new curriculum, the strategy-comprehension question given as Figure 12 (section 2.7.2) was used to specifically target comprehension of the Guarded Division plan after explicit instruction. This question showed a function used to calculate an average; however, there was no guard around the division so it was susceptible to failure if the count of values was zero. Students were asked to identify the flaw and suggest a remedy.

Table 2. Change in Guarded Division application

	Correct	Proportion
Application in generation study before explicit *strategy* instruction	4 of 42	10%
Comprehension in 2005 exam under new curriculum	25 of 36	69%

Results from Table 2 show the poor application of the Guarded Division plan under implicit-only *strategy* instruction and the potential of students to comprehend this *plan* after explicit instruction. After explicit *strategy* instruction, correct answers to the Guarded Division were provided by 25 of 36 students. This indicates that most students had learned and could comprehend the Guarded Division plan, knowing where it should be applied.

Testing *comprehension* of a *strategy* (as in this problem) is not directly comparable to *generation* of that *strategy* (as with the initial study). However, knowing that 69% of students comprehend the Guarded Division plan should be kept in mind when considering the results of a comparison using a generation task in the next subsection.

4.2.2 Averaging Problem (2007 Examination)

During the examination from S2 2007 the questions shown in Figure 13 (section 2.7.4) were used. From this figure Question 8 repeats the averaging problem used in the initial study (de Raadt, Toleman, & Watson, 2004).

Solutions to this problem were analysed under Goal/Plan Analysis, with the same list of *plans* sought. Figure 14 distinguishes results between the initial test, where novices learned programming *strategies* in an implicit-only manner and attempted the problem in class in the second last week of semester, and the examination question under the new curriculum that included programming *strategies* explicitly. Results show consistent improvement in all *plans* except one. The Guarded Division plan is still the most poorly applied *plan*, with only 38% of participants using this *plan* even after explicit instruction; however, this is a significant increase ($\chi^2 \approx 9.47$, $p \approx 0.002$), almost fourfold from the initial study, and this level is higher than the level demonstrated by experts (de Raadt, Toleman, & Watson, 2006). There was also a significant increase in use of the Sentinel-Controlled Count Loop plan ($\chi^2 \approx 4.98$, $p \approx 0.03$).

Figure 15 compares the completeness (use of all expected *plans*) from the initial study and results from the averaging question in an examination under a curriculum with explicit programming *strategies*. Under the new curriculum, the proportion of correct solutions increased from 2% (1 of 42) to 31% (14 of 45) which is a significant increase ($\chi^2 \approx 12.56$, $p \approx 0.0004$). If the most poorly applied *plan*, Guarded Division, is ignored the proportion of complete (and near-complete) answers has increased from 20% (10/42) to 49% (22/45) which is also a significant increase ($\chi^2 \approx 5.88$, $p \approx 0.02$).

Table 3. Improvement between cohorts

Exam	Average Plan Application
Implicit-only (2003)	4.0 of 7 plans (57%)
Explicit (2007)	4.8 of 7 plans (69%)

There was an improvement in the average proportion of application of the seven expected *plans* between the student cohorts. As shown in Table 3, prior to explicit instruction of programming *strategies*, students applied 57% of the expected *plans* on average. With explicit instruction of programming *strategies*, this increased to 69% of the expected *plans* on average. Using a two-sample t-test (one-tailed) there is evidence of a statistically significant improvement between the two cohorts ($df=85$, $t \approx 1.66$, $p \approx 0.02$).

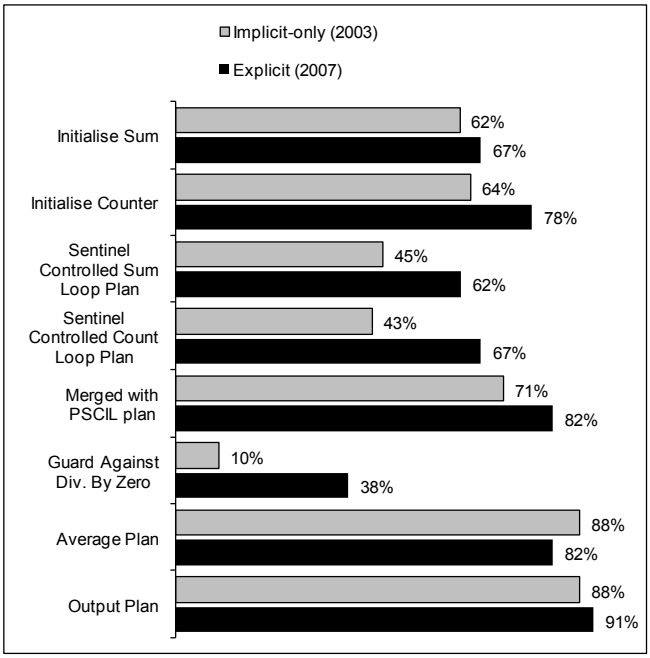


Figure 14. Comparison of *plan* use in the averaging problem between implicit-only and explicit *strategy* instruction

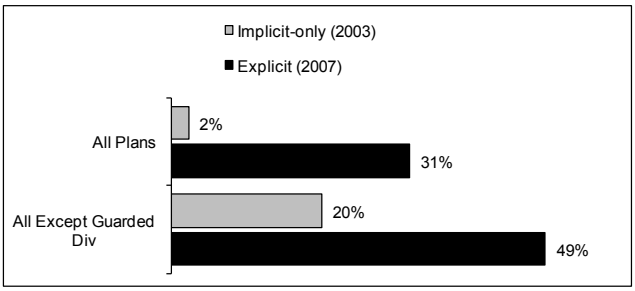


Figure 15. Comparison of complete and near-complete correctness in averaging problem before and after explicit *strategy* instruction

5. Discussion

In this section we use the results from section 4 to answer the research questions posed in section 1.3.

5.1 Integration

RQ1. *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*

While it did take some time and effort to transform a traditional curriculum, adding explicit *strategy* content, this was shown to be possible. The amount of *strategy* content is not necessarily fixed and needs to be further refined. Sharing these *strategies* with other instructors will allow this development. It is useful to reiterate that *strategies* can be used with most imperative and object-oriented languages so they would suit the majority of introductory programming courses, requiring little change for different languages.

RQ2. *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

It is possible to measure programming *strategy* ability in novices with tests that address both *comprehension* and *generation*. A number of different forms of assessment have been demonstrated for programming assignments and examinations, providing additional instruments, beyond Goal/Plan Analysis for gauging *strategy* skill. Most assessment methods used in the new curriculum resemble traditional curriculum assessment items, but with careful problem design and objective criteria for evaluation, assessment items can be used to focus testing of *knowledge* and *strategies* independently.

5.2 Impact

RQ3. *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

The results show students' use of *strategies* under a curriculum where *strategies* are covered explicitly is better compared to those results achieved under an implicit instruction curriculum. There is a strong improvement in overall completeness of solutions to the averaging problem tested between the initial study (de Raadt, Toleman, & Watson, 2004) and an examination under the new curriculum. There is a specific improvement in the use of the most poorly applied *strategy*, the Guarded Division plan, although its application is still relatively low.

However, the results shown here are clearly retrospective and do not definitively prove the benefits of explicit *strategy* instruction. The results are consistent and the sample sizes

provide confidence in the result. However, with two disparate cohorts separated by four years, student capability and individual differences make it difficult to definitively claim that improvement in this very specific task is attributable to the change of teaching method. There is still a need for a more direct comparison to isolate the impact of such instruction.

6. Conclusions and Future Work

This study has shown that it is possible to instruct and assess programming *strategies*. Teaching programming *strategies* in this way creates a vocabulary that can be used in teaching and assessment, and reused and reinforced after they are presented.

This study has also shown that *strategies* can be a valid part of assessment and can therefore be a valuable part of an introductory programming curriculum that aims to train novice programmers to apply programming *strategies*. The methods of *strategy* skill assessment used can be applied to both *comprehension* and *generation* exercises and conducted throughout a course. Strategy-related questions in examinations can elicit results consistent with questions that assess programming *knowledge* skill. *Strategy* skill testing can also be achieved in regular assignments. With a more precise vocabulary for defining a complete solution to a problem, instructors can avoid vague terms such as 'elegance' and 'connoisseurship' when assessing the work of a novice; instead, instructors can point out what *strategies* are absent or misapplied in novices' solutions.

Students seem to learn and apply programming *strategies* more consistently when they are presented in an explicit manner than when they are learned implicitly. However, further experimentation is required to isolate the effects of this approach on the development of novices.

With a well defined distinction between programming knowledge and strategies in an introductory course, there is potential to investigate programming strategies as possible threshold concepts (Boustedt et al., 2007; Entwistle, 2007).

7. References

Biederman, I., & Shiffrar, M. M. (1987): Sexing Day-Old Chicks: A Case Study and Expert Systems Analysis of a Difficult Perceptual-Learning Task. *Journal of Experimental Psychology: Learning, Memory and Cognition,* **13**(4):640 - 645.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007): Threshold concepts in computer science: do they exist and are they useful? *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, Covington, Kentucky, USA 504 - 508, ACM Press.

Brooks, R. E. (1983): Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies,* **18**:543 – 554.

Davies, S. P. (1993): Models and theories of programming strategy. *International Journal of Man-Machine Studies,* **39**(2):237 - 267.

de Raadt, M. (2008) Strategies Reference, http://www.sci.usq.edu.au/staff/deraadt/research/dissertation/Strategies%20Reference.pdf. Accessed November 24 2008.

de Raadt, M., Toleman, M., & Watson, R. (2004): Training strategic problem solvers. *ACM SIGCSE Bulletin,* **36**(2):48 - 51.

de Raadt, M., Toleman, M., & Watson, R. (2006): Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications,* **28**(5):55 - 62.

de Raadt, M., Toleman, M., & Watson, R. (2007): Incorporating Programming Strategies Explicitly into Curricula. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Koli, Finland 53 - 64.

Entwistle, N. (2007): Conceptions of Learning and the Experience of Understanding: Thresholds, Contextual Influences, and Knowledge Objects. In S. Vosniadou, A. Baltas & X. Vamvakoussi (Eds.), *Re-Framing the Conceptual Change Approach in Learning and instruction* (pp. 123 - 143): Elsevier, in association with the European Association for Learning and Instruction.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin,* **36**(4):119 - 150.

McCracken, M., Wilusz, T., 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. *ACM SIGCSE Bulletin,* **33**(4):125 - 180.

Muller, O., Haberman, B., & Ginat, D. (2007): Pattern-Oriented Instruction and its Influence on Problem Decomposition and Solution Construction. *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007)*, Dundee, Scotland.

Porter, R., & Calder, P. (2003): A Pattern-Based Problem-Solving Process for Novice Programmers. *Proceedings of the Fifth Australasian Computing Education Conference (ACE2003)*, Adelaide, Australia 20:231 - 238, Conferences in Research and Practice in Information Technology.

Reber, A. S. (1993): *Implicit Learning and Tacit Knowledge*. New York, USA: Oxford University Press.

Rist, R. S. (1991): Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction,* **6**:1 - 46.

Robins, A., Rountree, J., & Rountree, N. (2003): Learning and Teaching Programming: A Review and Discussion. *Computer Science Education,* **13**(2):137 - 173.

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**(2):157-172.

Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM,* **29**(9):850 - 858.

Wallingford, E. (1996): Toward a first course based on object-oriented patterns. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, Philadelphia, PA USA 27 - 31, ACM Press, New York, NY, USA.

Wallingford, E. (2007) The Elementary Patterns Home Page, http://cns2.uni.edu/~wallingf/patterns/elementary/. Accessed 19th November 2007.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A., & Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia 52:243 - 252.