Incorporating Programming Strategies Explicitly into Curricula

Michael de Raadt, Mark Toleman
School of Information Systems
Faculty of Business
University of Southern Queensland
Toowoomba, Queensland, 4350, Australia
{deraadt,markt}@usq.edu.au

Richard Watson
Department of Mathematics and Computing
Faculty of Sciences
University of Southern Queensland
Toowoomba, Queensland, 4350, Australia
rwatson@usq.edu.au

Abstract

An experiment was conducted to test a curriculum that explicitly incorporated programming strategies in lectures, written course materials, exercises and assessment. A control curriculum was also established to allow for comparison and isolation of effects. The two curricula were delivered to two groups of volunteer students who had no previous programming experience. The experimental group showed understanding and application of programming strategies, used the vocabulary plans in interviews and showed greater confidence in their solutions to problems. This suggested that explicit incorporation of programming strategies into an introductory programming curriculum has the potential to improve outcomes for novice programmers.

Keywords: Introductory programming, curriculum, programming strategies.

1 Introduction

An important dimension identified in literature by Robins, Rountree, & Rountree (2003) is the knowledge-strategy dimension. *Knowledge* involves the declarative nature of a programming language while *strategies* describe how programming knowledge is applied (Davies, 1993). Programming strategies are made up of plans (Soloway, 1985) (or schema or patterns) and the associated means of incorporating these into a single solution. Soloway (1986) suggests:

> …language constructs do not pose major stumbling blocks for novices... rather, the real problems novices have lie in "putting the pieces together," composing and coordinating components of a program. (p. 850)

Soloway then suggests teaching should reach beyond a focus on syntax (as programming knowledge) and focus on programming strategies. Recent studies (Lister et al., 2004; Whalley et al., 2006) have suggested novice programming knowledge can be fragile, so it is important to focus on both programming knowledge and strategy in curricula. See de Raadt (2007b) for an overview of recent experiments in this area.

de Raadt, Tolman and Watson (2006) place problems faced by programmers on a scale as follows.

- **System Level Problems**
  Problems at this level are large in scale and usually unique. An example of a problem at this level might be designing an accounting system for a large corporation. Students generally study problem solving at this level in a systems analysis course.
- **Algorithmic Level Problems**
  Problems at this level are identifiable parts of a greater problem. (In an academic setting they may be addressed independently.) For such problems a solution is usually achieved by adopting well refined algorithms, widely used in the programming community. A novice may be able to start using such strategies at the end of an initial course in programming and may use them in greater depth in a second or third course in programming.
- **Sub-algorithmic Level Problems**
  Problems at this level are at their most basic. Attempting to decompose and describe a problem below this scale will lead to syntactical definitions. Examples of problems at this scale are avoiding division-by-zero, achieving repetition until a sentinel is found, and so on. This level of problem solving is particularly relevant to novices in their initial exposure to programming. This level is perhaps the least recognised yet most fundamental to good programming problem solving.

Another important dimension relevant to this experiment defines how instruction is delivered, which can be described as being implicit, explicit, or a combination of these. *Explicit* instruction involves the instructor openly describing, usually in some documented form, what the student is to learn and how to go about that learning. *Implicit* instruction creates a scenario where a student is expected to undertake new learning, or extend previous learning, without being given a full context for what they are to learn or how. From the results of an experiment conducted by Biederman and Shiffrar (1987), Baddeley (1997) suggested a short period of explicit instruction can be more effective than months of implicit learning. Experiments by Reber (1993) showed students can learn through implicit-only means, but this leads to a poor understanding of the underlying systems involved. Traditional curricula tend to rely on novices acquiring programming strategies implicitly.

A previous study (de Raadt, Toleman, & Watson, 2004) investigated an introductory programming course where novices were expected to learn programming strategies implicitly. Novices who participated in the study were asked to create a solution to a simple averaging problem. Solutions were scrutinised under Goal/Plan Analysis (Soloway, 1986) to measure application of strategies. Only one of 42 novices demonstrated application of all expected strategies. Students~~’~~’ solutions showed flaws in initialising variables, using a correct repetition strategy, guarding against events such as division by zero, and merging strategies that should be achieved together. These flaws implied weaknesses in the curriculum being delivered to the students at the time.

A second study (de Raadt, Toleman, & Watson, 2006) uncovered a model of expert programming strategies at a sub-algorithmic level based on plans described by Soloway (1986). These strategies can be explicitly expressed. This study suggested that the explicit inclusion of programming strategies should be attempted as it may:

- improve outcomes for students,
- establish a vocabulary for programming strategy dissemination, and
- allow students’ programming strategy skills to be assessed.

This current experiment was conducted to discover if programming strategy instruction can be explicitly incorporated into an introductory programming curriculum, and if this is possible, what effects can be observed. Two curricula were designed to allow comparison and isolation of effects. One curriculum included explicit programming strategies while the second relied on implicit learning of programming strategies. Each of these curricula was delivered over a single weekend and followed by a series of one-on-one interviews with participants. No credit was awarded to participants; participants gained the experience of learning programming.

1.1 Research Questions

This experiment was motivated by the following interrelated questions (answered in section 5).

- Can programming strategies be explicitly incorporated into an introductory programming curriculum?
- What is the significance of the time consumed by this additional instruction?
- Can programming strategies explicitly taught in an introductory programming course be assessed?
- What impact does explicit strategy instruction have on students and their problem solving ability when compared to an implicit-only approach?
- Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?

In section 2, details of the experimental curriculum are described. Section 3 describes how the experiment was undertaken. Results of the experiment are displayed and discussed in section 4. Section 5 answers the research questions and concludes with implications and future work.

2 Description of Curricula

A base curriculum was created that contained programming strategy instruction explicitly. This curriculum is described further in this section and is included in full in a working paper (de Raadt, 2007a). From this, a second curriculum was created by identifying and removing programming strategy instruction components.

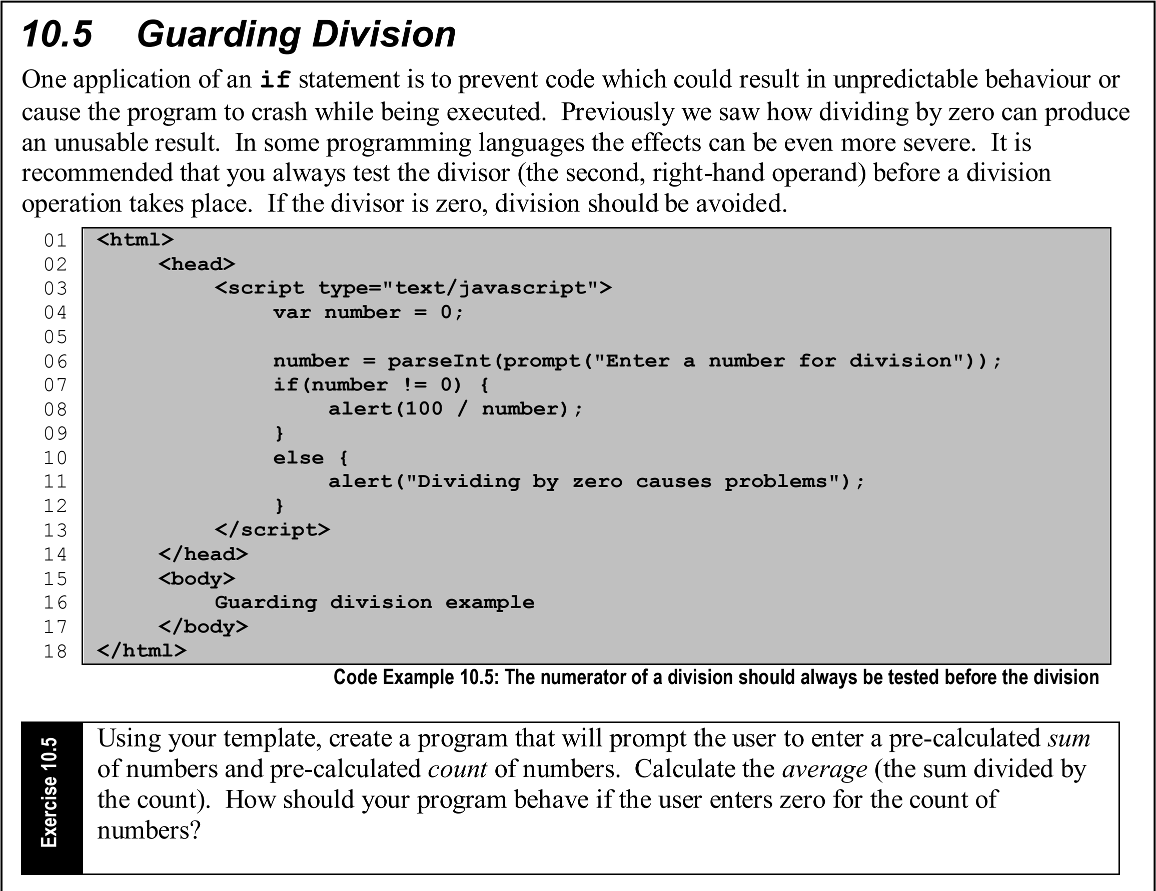
2.1 Incorporating Explicit Programming Strategies

In this experiment Solway’s *plans* were chosen over *patterns*, even though patterns have become more widespread in recent years. Patterns are bound to the Object paradigm and require a pattern language for application. Plans can be used in multiple paradigms, including the Object paradigm. Plans can be expressed simply, particularly at a sub-algorithmic level. In saying this, the focus of this research is not on the type of strategies that are taught but on *how* they are taught, and outcomes for students from that. It is likely that patterns could be used to achieve the same programming strategy understanding for students as plans. From this point on the term *plan* is used is used to represent a specific form of strategy and the term *strategy* is used in its more generic sense.

Programming strategies are explicitly incorporated into the curriculum in a number of ways. These are described in subsections 2.1.1 to 2.1.3.

2.1.1 Identifying Strategies in the Curriculum

A book of written study materials was created and hardcopies were given to participants. Lecture slides were created based on the content of the written study materials. The lecture slides were used during lectures. In these written materials and lectures the strategies incorporated in the curriculum were named, their benefits were explained, and examples of their application were shown. Figure 2.1 shows a section of the written materials provided to students. In this example the Guarded Division Plan is identified. An explanation is given for why this plan is used, including a reference to an earlier mention of the consequences of dividing by zero. The description tells how the strategy is implemented and an example coded implementation of this strategy is shown. As well as introducing strategies, the descriptions also covered the means of integrating these strategies through abutment, merging and nesting (Soloway, 1986, p. 856).


10.5 Guarding Division

One application of an **`if`** statement is to prevent code which could result in unpredictable behaviour or cause the program to crash while being executed. Previously we saw how dividing by zero can produce an unusable result. In some programming languages the effects can be even more severe. It is recommended that you always test the divisor (the second, right-hand operand) before a division operation takes place. If the divisor is zero, division should be avoided.

```
<html>
    <head>
        <script type="text/javascript">
            var number = 0;

            number = parseInt(prompt("Enter a number for division"));
            if(number != 0) {
                alert(100 / number);
            }
            else {
                alert("Dividing by zero causes problems");
            }
        </script>
    </head>
    <body>
        Guarding division example
    </body>
</html>
```

Code Example 10.5: The numerator of a division should always be tested before the division

Exercise 10.5

Using your template, create a program that will prompt the user to enter a pre-calculated *sum* of numbers and pre-calculated *count* of numbers. Calculate the *average* (the sum divided by the count). How should your program behave if the user enters zero for the count of numbers?

Figure 2.1. An extract from the written course materials showing explicit incorporation of a problem solving strategy instruction

2.1.2 Paper Exercises and Practical Computing Tasks

At the end of each module students were asked to complete paper exercises and computer based tasks that reinforced the content delivered in lectures and allowed students to experience the practical implementation of the strategies covered. Instructions for these exercises and tasks were set out in the written materials, such as Exercise 10.5 shown in Figure 2.1. The exercise shown prompts users to explore Guarding Division. In other exercises students are prompted to experiment with the outcome achieved when the strategy is not applied or poorly applied. During the course, as with any normal introductory programming class, the instructor was on hand to answer questions and guide students. In most cases the exercises and tasks given were common to both curricula. In the curriculum without explicit programming strategies students were expected to learn the required programming strategies implicitly.

2.1.3 Assessment of Programming Strategies

At the end of the course, students were asked to complete the same three programming tasks that were given to experts in the previous study with experts (de Raadt, Toleman, & Watson, 2006). These tasks were used as a formal assessment at the end of the course under exam conditions. As well as testing participants' abilities, this was done to explore the potential to assess programming strategies as part of a course. The strategies necessary to solve the final assessment problems had been shown as examples and in exercises and programming tasks.

2.2 Format of the Curriculum

The curriculum is based on a traditional curriculum that reveals parts of a given language in a sequence, with new knowledge of language concepts being dependent on previously covered knowledge. In this format, explicitly incorporating programming strategies depends upon certain underlying knowledge being taught beforehand. For instance, for the Guarded Division plan to be introduced, knowledge of variables, operators and selection must be covered first. Looking at the titles of the modules of the course shown in Table 2.1 gives little clue that explicit programming strategies are involved.

Basing the experimental curriculum on a traditional curriculum allowed the creation of a second curriculum without explicit programming strategies. In a non-experimental setting, the format of the curriculum could change. For instance, the structure of the course could be governed by the strategies themselves instead of the underlying language; in this case strategies could be introduced then underlying language knowledge could be taught. If an objects-first approach is taken, strategies could be used at other stages.

2.3 Philosophy behind the Experimental Curriculum

The curriculum was designed to be short and to allow students to reach programming strategies as soon as possible. The curriculum would not be effective in teaching longer courses, although the ideas used in the explicit incorporation of programming strategies could be applied to longer curricula.

The curriculum focused on programming strategies, with only a minimal covering of the knowledge components on which the covered strategies are dependent. Knowledge content was included if it was fundamentally important for learning the later programming strategies. Later exercises focused on the application of programming strategies. For those who had not been explicitly instructed in programming strategies, this was their opportunity to implicitly learn the needed strategies

Table 2.1. Comparison of the two curricula tested (items with ~~strike through~~ were absent in control curriculum)

Module	Section	Curriculum A (with Explicit PSS)	Curriculum B (without Explicit PSS)
1		**First JavaScript Program**	**First JavaScript Program**
	1.1.	Hello World!	Hello World!
	1.2.	JavaScript and HTML	JavaScript and HTML
	1.3.	Statements	Statements
2		**Calling Functions**	**Calling Functions**
	2.1.	alert()	alert()
3		**Values**	**Values**
	3.1.	Numbers	Numbers
	3.2.	Strings	Strings
	3.3.	Booleans	Booleans
4		**Variables**	**Variables**
	4.1.	What are Variables	What are Variables
	4.2.	Identifier Rules	Identifier Rules
	4.3.	Declaring Variables with var	Declaring Variables with var
	4.4.	Undefined	Undefined
5		**Assigning Values**	**Assigning Values**
	5.1.	Dynamic Typing	Dynamic Typing
	5.2.	typeof	typeof
	5.3.	Initialising Variables	~~Initialising Variables~~
6		**Operations**	**Operations**
	6.1.	Arithmetic Operators	Arithmetic Operators
	6.2.	Division by Zero – infinity	~~Division by Zero – infinity~~
	6.3.	Postfix Operators	Postfix Operators
	6.4.	Relational Operators (incl. Equality)	Relational Operators (incl. Equality)
	6.5.	Logical Operators	Logical Operators
	6.6.	String Operators	String Operators
7		**Abutment**	~~**Abutment**~~
8		**Debugging**	**Debugging**
			~~Exercise 8.3~~
9		**Functions that Return Values**	**Functions that Return Values**
	9.1.	prompt()	prompt()
	9.2.	parseInt() and parseFloat()	parseInt() and parseFloat()
10		**Selection**	**Selection**
	10.1.	The if Statement	The if Statement
	10.2.	The if-else Statement	The if-else Statement
	10.3.	Indenting and Formatting	Indenting and Formatting
	10.4.	"Dangling else"	"Dangling else"
	10.5.	Guarding Division	~~Guarding Division~~
11		**Repetition (Loops)**	**Repetition (Loops**
	11.1.	while Loop	while Loop
	11.2.	Sentinel Controlled Loops	~~Sentinel Controlled Loops~~
	11.3.	for Loop	for Loop
	11.4.	Counter Controlled Loops	~~Counter Controlled Loops~~
	11.5.	Finding the Maximum/Minimum	~~Finding the Maximum/Minimum~~
	11.6.	Nesting and Merging	~~Nesting and Merging~~
12		**Arrays**	**Arrays**
	12.1.	Declaring Arrays	Declaring Arrays
	12.2.	Accessing Array Elements	Accessing Array Elements
	12.3.	Initialising Arrays	Initialising Arrays
	12.4.	Arrays for Values	Arrays for Values
	12.5.	Arrays for Categories	Arrays for Categories
	12.6.	Counting Values in a Set	~~Counting Values in a Set~~

through practical exercises. The assessment at the end of both forms of the course focused on the analysis of programming strategy skills developed through the course. In a non-experimental course the focus of exercises and the weighting of examination questions would be more balanced between knowledge components and programming strategies.

2.4 Language Used with Experimental Curriculum

JavaScript was used as the language to support the instruction of the curriculum. In their essential form, programming strategies are language independent and examples could be given in almost any language. Soloway and his colleagues used Pascal and Lisp to illustrate programming strategies. The authors have used C/C++ to exemplify programming strategies in other work.

JavaScript was chosen for this experiment for the following reasons:

- potential to reach important concepts rapidly;
- simpler to practice than a compiled language;
- attractive to volunteers;
- allows expression of programming strategies with a programming language not previously used for this purpose.

3 Methodology

The method of experimentation began with preliminary demographic, experience and confidence measurements. An examination of programming strategies was conducted at the end of each weekend. In the weeks that followed the two weekend sessions participants were invited to an interview in which they were asked questions about their solutions to gauge their understanding of the strategies that were being tested.

3.1 Volunteer Participants

Participants were volunteers from the student body at the University of Southern Queensland, and were recruited by posters hung around the university campus and by emails sent to former students of two computing concepts courses for non-computing students.

Participants were asked to undertake an initial survey that gathered demographic data, computing experience, past programming experience and a measure of computing confidence.

This initial data was used to filter students who had previous programming experience. Students with *no* previous programming experience were sought in order to set a baseline for all participants. Volunteers with previous programming experience were asked to withdraw.

A number of the volunteers withdrew from the weekend courses, mostly due to personal reasons, giving notice before the start of the experiment. A number of other volunteers failed to attend the course, which was unexpected, and reduced the group of volunteers to eight in two groups of four, divided on a self-selecting basis. One of the participants who attended the first weekend had completed a previous course in computer programming and arrived after being asked by email not to attend. Results were collected from this participant but are not aggregated with other participants in this experiment.

3.2 Setting

The two weekend courses were conducted in a computing lab. This room included facilities for lecturing, computers for students to undertake practical exercises, and desk space between computers for students to complete paper-based exercises.

The two curricula were delivered on consecutive weekends. The curriculum without explicit programming content was delivered first and this was followed the next weekend by the curriculum with explicit programming strategies. The ordering of the two curricula was arbitrary.

The two days of each weekend were divided into sessions; with each session covering one to four modules of the course (see the schedule in section 3.4). Each session consisted of an initial lecture with questions encouraged from students. This was followed by paper tasks and practical programming tasks. Later in the course, tasks that involved programming strategies were used. Students were given breaks between sessions.

3.3 Demographic, Experience and Confidence Measures

A number of demographic, experience and confidence measures were conducted via a web survey presented to students when they volunteered. Participants were asked questions about:

- gender;
- age;
- computing experience;
- previous programming experience; and
- computing confidence.

Details of specific questions are given in de Raadt (2007a). Computing confidence was captured using a test created by Cretchley (2006), which has proven to be a reliable predictor of computing confidence in the past.

3.4 Schedule of Course Delivery

The schedule for both weekends was identical except where programming strategy content was covered. In Table 3.1, content covering programming strategies is highlighted and was covered only in the course with explicit instruction of programming strategies. Participants undertaking the course without explicit programming strategy content were intended to be attempting practical exercises during these times. One of the aims of the experiment was to determine if this additional content would impact on the balance of time allowed for lecture instruction versus exercises and practice. For this reason the schedule was followed as closely as possible on both weekends.

Table 3.1. Schedule for Weekend Courses

Session	Saturday Content	
10:00 – 11:15	Introductions	
	1	First JS Program
	1.1	Hello World
	1.2	JavaScript and HTML
	2	Calling Functions
	2.1	alert()
11:30 – 13:00	3	Values
	3.1	Numbers
	3.2	Strings
	3.3	Booleans
	3.4	Undefined
	4	Variables
	4.1	What are Variables
	4.2	Identifier Rules
	4.3	Creating variables with var
	5	Assigning Values
	5.1	Dynamic typing
	5.2	typeof
	5.3	Initialising Variables
13:30 – 14:45	6	Operations
	6.1	Arithmetic Operators
	6.2	Division by Zero - Infinity
	6.3	Postfix Operators
	6.4	Relational Operators (incl. Equality)
	6.5	Logical Operators
	6.6	String Operators
	7	Abutment
	8	Debugging
	9	Functions that Return Values
	9.1	prompt()
	9.2	parseInt()
15:00 – 16:00	10	Selection
	10.1	The if Statement
	10.2	The if-else Statement
	10.3	Indenting and Formatting
	10.4	"Dangling else"
	10.5	Guarding Division
	Sunday Content	
10:00 – 11:15	11	Loops
	11.1	while Loop
	11.2	Sentinel Controlled Loops
	11.3	for Loop
	11.4	Counter Controlled Loops
	11.5	Finding the Maximum
	11.6	Nesting and Merging
11:30 – 13:00	12	Arrays
	12.1	Arrays for Values
	12.2	Arrays for Categories
	12.3	Counting Values in a Set
13:30 – 14:45	Testing	

3.5 Administering the Final Assessment

After lunch on the Sunday of each weekend course, participants were asked to complete the three programming tasks previously given to experts (de Raadt, Toleman, & Watson, 2006). Each problem was presented on a single sheet of paper with lines below to complete the solutions to the problems (solution sheets are shown in de Raadt (2007a)). Participants were able to use as much time as was needed to complete problems.

Problem 1
Read in 10 positive integers from a user. Assume the user will enter valid positive integers only. Determine the maximum.

Problem 2
Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.

Problem 3
Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.

The solutions produced were examined using Goal/Plan Analysis to test for the presence or absence of expected plans. This was conducted in the same manner as the earlier experiment with experts. The expected strategies and means of integration are given with results.

3.6 Post-Experiment Interviews with Participants

In the 23-day period after teaching, six participants gave verbal, one-on-one interviews. Students' solution sheets were used as a basis for discussion. Interviews were structured, with set questions as listed in de Raadt (2007a). The questions were used as a script, but were intended to encourage discussion that was allowed to continue as long as necessary. The questions used were designed not to be leading. Questions were aimed at discovering participants' interpretations of the problem statements, the strategies understood by participants, the articulation of their solutions and their confidence in their solutions.

4 Results

A number of results were gained from this experiment. First, data gathered during registration are shown. During the experiment both curricula were delivered to students. The potential to succeed in this delivery was judged by the time used to deliver the more extensive curriculum that explicitly incorporated programming strategies within the schedule. At the end of each of these sessions participants were asked to complete a set of problems that were examined under Goal/Plan Analysis. Finally an inspection of post-course interviews provides deeper insights into the programming strategy potential of the participants after the course.

4.1 Data Collected at Registration

The data gathered when participants volunteered for the course are shown in Table 4.1. These data show that the two groups were roughly balanced in gender, age and computing confidence. The two groups differed in responses to computing and web experience self-assessment questions. Experimental group participants showed varying responses to these experience questions. One of the participants indicated they had no previous use of a web browser, even though they used a computer daily. This may have been an error.

Table 4.1. Demographic, experience and confidence data gathered on registration

Group	Participant	Gender	Age Group	Computing Experience	Web Experience	Previous Programming	Computing Confidence 1=low to 5=high
Experimental Group	12	male	Less than 25	Daily use	No use	Never	3.0
	21	male	26 – 35	Daily use	Daily use	Some self-taught	4.6
	29	male	26 – 35	Weekly use	Every few days	Never	3.2
	30	female	Less than 25	Daily use	Daily use	Never	4.4
Average							**3.8**
Control Group	1	male	Less than 25	Daily use	Daily use	Never	3.6
	6	female	Less than 25	Daily use	Daily use	Never	3.5
	13	male	26 – 35	Daily use	Daily use	Never	3.8
Average							**3.6**

Table 4.2: Presence of plans and integration for Problem 1

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Max Initialised					0%				0%	0%
Counter Controlled Loop	Y	Y	Y		75%	Y	Y		67%	71%
Input Plan	Y	Y			50%	Y	Y	Y	100%	71%
Maximum Plan	Y				25%				0%	14%
Output Plan	Y	Y			50%			Y	33%	43%
Input Nested in Counter Controlled Loop	Y	Y			50%	Y			33%	43%
Max Plan Nested in Counter Controlled Loop	Y				25%				0%	14%
Abutment Correct	Y	Y	Y		75%	Y		Y	67%	71%
Overall	88%	63%	25%	0%	**44%**	50%	25%	38%	**28%**	41%

Table 4.3: Presence of plans and integration for Problem 2

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Sum Initialised	Y				33%	Y	Y		67%	50%
Count Initialised	Y				33%	Y			33%	33%
Sentinel Controlled Input	Y	Y			67%				0%	33%
Sentinel Controlled Count	Y				33%		Y		33%	33%
Sentinel Controlled Sum	Y				33%		Y		33%	33%
Guarded Division			Left Early		0%				33%	0%
Output Plan	Y	Y			67%	Y	Y	Y	0%	83%
Loop Plans Merged	Y				33%	Y			100%	33%
Inputs Nested in Sentinel Controlled Loop	Y	Y			67%				33%	33%
Output Nested in Guarded Division					0%				0%	0%
Abutment Correct	Y	Y			67%	Y		Y	67%	67%
Overall	82%	36%		0%	**39%**	45%	36%	18%	**33%**	36%

Table 4.4: Presence of plans and integration for Problem 3

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Counter Controlled Loop (for Initialisation)	Y			Y	67%				0%	33%
Array Initialisation	Y	Y		Y	100%				0%	50%
Sentinel Controlled Input	Y				33%				0%	17%
Count Set Plan	Y	Y			67%				0%	33%
Counter Controlled Loop (for Output)				Y	33%	Y	Y		67%	50%
Output Plan	Y	Y	Left Early		67%				0%	33%
Initialisation nested in Counter Controlled Loop	Y			Y	67%				0%	33%
Inputs nested in Sentinel Controlled Loop	Y			Y	67%				0%	33%
Count Set nested in Sentinel Controlled Loop	Y				33%				0%	17%
Output Nested in Counter Controlled Loop					0%				0%	0%
Abutment Correct	Y	Y		Y	100%	Y	Y	Y	100%	100%
Overall	82%	36%		55%	**58%**	18%	18%	9%	**15%**	36%

One of the intentions in gathering this data was to exclude volunteers who had completed previous formal study in programming. A number of people signed up for the experiment and were rejected because they had studied programming previously. One participant, identified as Participant 14, who was asked not to attend, came along anyway. The results of this participant are not presented here, but their solutions and transcript are presented in de Raadt (2007a) as some of this participant's responses to interview questions were still of interest. One other participant (21) indicated they had some self-taught programming experience. After discussion with the participant this experience was shown to be a limited amount of HTML writing, which was not seen as significant in this experiment.

4.2 Time Load of Explicit Programming Strategy Instruction

During teaching of the curriculum that incorporated explicit programming strategies, added content required additional time to teach, increasing the length of lecture sessions and reducing the time allowed for students to undertake practical work. However, participants undertaking the curriculum with explicit programming strategies were still able to complete the set exercises during the time allocated in the schedule. It was possible for the schedule to be followed in both instances of the curriculum.

4.2.1 Goal/Plan Analysis of Participant Solutions

Tables 4.2 to 4.4 show results of the Goal/Plan Analysis for each problem. Several of the solutions presented by novice participants in this experiment contained English language text that described the code the participant would like to have written in their solution when they were not sure how to implement these ideas. Where this was the case, if the text sufficiently described a plan, it was accepted as being present even if it was not described in code. The participants who used text in their code did not create complete or near complete solutions.

Table 4.2 shows the plans present in each participant's solution to Problem 1. The correctness of the integration of the strategies is also recorded and this included correctness of abutment. Unlike experts studied earlier (de Raadt, Toleman, & Watson, 2006), participants in this experiment did not always apply these integration aspects correctly.

The best problem 1 solution was created by Participant 12 from the experimental group who, despite never previously undertaking programming study, was able to produce a well coded solution that was nearly completely correct. This solution, together with those presented by Participant 21, pushed the overall average correctness level for the experimental group above that of the control group despite the abandoned attempt and non-attempt of their group-mates.

One noticeable aspect was the absence of the initialisation of the maximum variable, which was crucial to the Maximum Plan and is required when using JavaScript. Initialisation was explicitly covered in the curriculum that explicitly included programming strategies. Students undertaking the other curriculum were presented with the opportunity to discover this aspect implicitly. Initialisation was important to the later problems and was applied by a number of participants for those problems. It is not clear why it is absent here.

Table 4.3 shows the strategy correctness of participants' solutions to Problem 2. Participant 29 left after abandoning an attempt at Problem 1, so this participant's solutions were not included in results for this and the next problem.

In this problem again, an outstanding solution was presented by Participant 12 who correctly solved the problem, with the exception of the Guarded Division plan. No participant in either group applied a Guarded Division plan. This suggests that even when it is explicitly incorporated into an introductory programming curriculum, and the consequences of failing to apply the plan are discussed, it is still possible for novice programmers to neglect this particular plan.

This problem was a modified version of the problem given to students in the earlier study (de Raadt, Toleman, & Watson, 2004). Students in the earlier study had completed a semester of instruction under a traditional implicit-only model and achieved an average overall correctness of 57.1% compared to the participants of this experiment who achieved 36%. In the problem statements for Problem 1 and both other problems, students were told they could assume inputs would be valid.

Table 4.4 shows the plan application for the final problem, Problem 3. Again an outstanding solution was presented by Participant 12, who correctly initialised and filled an array to tally user inputs, but failed to output the content of the array using a loop. Participant 30, who did not attempt Problem 1 and presented a confused solution to Problem 2, managed to apply a number of plans for this problem. Participants from the control group showed little ability to demonstrate any of the plans that were needed to solve this problem. This problem is arguably the most complex and, it would appear from these results, it is difficult to implicitly learn the necessary plans required to solve it.

One aspect that was absent in all solutions was the use of a Counter Controlled Loop plan to output the occurrences of numbers. This is not truly surprising as most of the solutions for this problem were incomplete and the only near-complete solution did not apply this particular strategy. Each of the participants from the experimental group applied a counter controlled loop to initialise the array used for tallying.

Table 4.5 shows a comparison of the overall correctness for all problems achieved by each group. There is a

Table 4.5: Overall plan use by each group

	Overall Plan Use
Experimental Group	47%
Control Group	28%
All	38%

distinction in overall results for the two groups with the experimental group, who were exposed to a curriculum that incorporated programming strategies explicitly, achieving a greater result.

Participant 12 produced outstanding solutions to each of the problems. It may be that the incorporation of explicit programming strategies suited this participant, who might have performed better than he would have otherwise. One must wonder if this participant would have done as well in the control group and perhaps reversed the results of the experiment.

With the small number of participants in this experiment no statistically significant evidence can be inferred for the superiority of one curriculum over another. These results are useful as basis for the interviews that followed, which allow a deeper and more personal exploration of the participating students' strategy understandings.

4.3 Interviews

Following the course, participants were asked to attend an interview. Five of the seven participants and Participant 14 (who had previous programming instruction) volunteered to attend interviews.

Each interview was recorded and transcribed. The transcripts of these interviews are presented in de Raadt (2007a).

From an analysis of the transcripts the following observations are made.

4.3.1 Participants Misinterpreted the Validation Simplification Made to Each Problem

Each problem statement contained the text "Assume the user will enter valid integers only." This additional text was introduced to clarify the problems so no attempt at validation would be necessary. This change was made when these problems were used with expert programmers but for this experiment it may have confused participants rather than simplifying the problems. In interviews participants were asked what this sentence in the problem meant. Three of the five participants misinterpreted this simplification; some suggested validation was necessary because of this statement. No participant attempted to validate inputs.

Other parts of the problem statements seemed to be comprehensible to each participant, even if they did not know how to achieve a solution.

4.3.2 Participants Exhibited Understanding of Plans

As well as demonstrating a higher use of plans in their solutions to problems, experimental group participants verbally described plans, for instance Participant 30 described their application of a Set Counting plan as follows: "After you've put a number that isn't in that range it concludes the program and tells the person what numbers you've put into your little boxes. It goes through zero to nine and it tells you how many are in each box."

Rist (1995) showed that novices can expound and apply plans without explicit instruction of programming strategies. Some control group participants did still learn plans through implicit-only means. In an observable instance Participant 6 stated the following, which could be seen as a description of a Set Counting plan using an array: "I've created an array, because I think that for the program to calculate, between 0 and 9, how many times it occurs, it has to have an array for, say if it's zero, then zero; for one it's one, two three, four... So the array for zero is, like, zero, because arrays start from zero, right? Then, so in the box for zero, say the user enters three times it will refer back to this array zero, it will keep repeating itself in the loop, from then on how many times it gets zero in that box it will get the output."

4.3.3 Participants Failed to Learn Some Plans

It was clear that participants did not learn all the plans they were expected to learn. This was true for participants from the control group who were expected to learn strategies implicitly, for example Participant 6 felt there must be a formula that would take care of the task of calculating maximums: "And probably some formula to determine the highest number (which I don't know how)."

Experimental group participants also failed to learn some plans, even though they had been explicitly exposed to them. For example when Participant 30 was asked how a maximum could be determined, responded, "Can you make the program look at the digits I guess, so you could determine the maximum. I don't know." When Participant 21 was asked, "What does it mean by determine the maximum?", responded with, "Perhaps the maximum sum. I'm not really sure."

4.3.4 Experimental Group Participants Used Plan Terminology and Ideas

On a number of occasions participants from the experimental group (who were exposed to plans and related terminology) referred to parts of their code using the terms used to describe plans or attempted to use plan terminology without specific names.

Participant 12, while discussing the integration of counting with input in Problem 2, said "they have to merge with the loop".

Participant 21, discussing loops in Problem 2, could not remember the terminology for a Sentinel Controlled Loop but described it well: "…and then create a loop… get user input outside and inside so that it's, I can't remember the name." Later Participant 21, while interpreting part of the problem statement, recalled the correct terms and said "Which I did recognise as a sentinel loop."

The use of Goal/Plan terminology was not universal by any means. Participants from the experimental group still resorted to syntactical description when describing their code and needed to be prompted further to elicit possible strategy understandings. Participant 12, who delivered perhaps the best result, stated the following syntactical reading of code: "It's a loop, for loop. For counter equals zero. Start from zero again. And counter smaller than numberNum. Counter++. And the message is numArray[counter] equals zero."

4.3.5 Experimental Group Participants showed Confidence in Solutions

One clear finding was that experimental group participants were confident in their solutions, or the ability to correct their solutions if given the chance. This is despite the fact that no participant created a fully correct solution to any of the problems. Participant 21 was confident about all his solutions, even though they were flawed. Participant 30 showed confidence in most of her attempted solutions even though they were flawed; when asked "Does your solution solve the problem?", replied, "…Well my solution in my head did, not like the first one, so yes. I did understand this question so I could go through the steps of doing it."

Participant 12, who was the closest of all participants to solving the problems correctly, was realistic about the correctness of his solution. During discussion Participant 12 saw the flaws in two of his three solutions. Interestingly this participant explains his confidence in one of his problems as being the result of understanding the required strategy: "I'm very confident in doing this question because I know the right way to structure [it]."

4.3.6 Control Group Participants showed a Lack of Confidence

When asked if they believed if their solutions correctly solved each problem, members of the control group almost universally showed a lack of confidence in the solutions they had created.

Participant 1 lacks confidence in all solutions except for Problem 2 solution where he claims more time was needed, even though time was not restricted during the test. When this participant was asked, "Does your solution solve the problem?", answered, "Probably, if I got time to add up more things." This same participant later describes a lack of confidence in their general programming ability: "I'll probably mess it up anyways, because I'm still not sure how to...", and later expresses a typical gap between design and implementation where plans can be applied: "I understand the question. I was thinking through. I got everything right in my head. I just can't put it onto codes."

The other control group participant interviewed, Participant 6, showed some confidence in one solution, believing, correctly, that the remaining solutions were flawed.

5 Conclusions

The research questions posed earlier are answered by the results of this experiment and the observations of the experimenter/instructor in conducting the experiment.

5.1 Explicitly Incorporating Programming Strategies

Can programming strategies be explicitly incorporated into an introductory programming curriculum?

Programming strategies can be explicitly incorporated into an introductory programming curriculum. The curriculum used in this experiment is evidence that this can be done.

5.2 Balance of Lectures and Practice

What is the significance of the time consumed by this additional instruction?

As stated in section 4.2 the additional instruction in the curriculum incorporating programming strategies explicitly did require more time in lecture sessions, but students were still able to complete set exercises by the end of each session. It can therefore be asserted that this additional instruction is balanced by an eased burden on students in completing practical exercises.

This result is useful for our comparison of the curricula, however in regular teaching, lectures and practicals are usually conducted in disjoint time slots; so extending the length of a lecture would not normally impact on practice time.

Having more material in one curriculum over another would increase the burden on student learning with more content to process. This needs to be compared with the effort a student would have to make to develop the needed programming strategies in an implicit-only model.

5.3 Assessment of Programming Strategies

Can programming strategies explicitly taught in an introductory programming course be assessed?

Goal/Plan Analysis of students' solutions is far from new, but it is novel as a means of assessment in a programming course. This experiment showed that programming strategies applied to create solutions can be assessed using Goal/Plan Analysis. A limitation of using Goal/Plan Analysis is that it requires students to generate code before it can be assessed. In early stages, assessing generated code might not be the best method of assessing programming strategies.

5.4 Impact on Problem Solving Ability

What impact does explicit strategy instruction have on students and their problem solving ability when compared to an implicit-only approach?

Through the results shown from Goal/Plan Analysis of participants' solutions and through interviews it appeared that students exposed to a curriculum that incorporated programming strategies explicitly were more likely to understand and apply those strategies than participants who were expected to learn these strategies implicitly.

It was, by no means, guaranteed that participants explicitly shown programming strategies would understand and apply all of these strategies. It was also demonstrated that participants exposed to an implicit-only curriculum can learn programming strategies.

5.5 Other Observed Effects

Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?

Two other observations can be made from the results shown. These are presented in the following subsections.

5.5.1 A Vocabulary for Programming Strategies

Some participants in the experimental group, who were exposed to plan terminology during their instruction, went on to use this terminology during interviews. If this were applied during an ordinary teaching period with multiple weeks of instruction and assessment, it would be beneficial to have students able use a common vocabulary of terms. Instructors would be able to describe the strategies they expect students to apply in tasks. It would be possible to allocate marks for the application of specified strategies. Students would have the potential to describe and analyse code using such terminology.

5.5.2 Confidence in Solutions

A clear contrast is shown in the confidence participants had in their solutions. Participants from the experimental group, who had been exposed to programming strategies explicitly, were confident about the solutions they presented and the understanding of the strategies needed to complete the solutions. Participants from the control group were not so confident. It is not necessarily clear why this is the case. Perhaps because experimental group participants had been exposed to a higher level of programming thought, they may feel that the underlying syntactical implementation is less difficult to achieve. Reber (1993) suggests that students exposed to implicit-only instruction can gain aptitude but fail to gain understanding of underlying systems. This seems to be consistent with the experience of participants exposed to implicit-only instruction of programming strategies in this experiment who were, in some instances, able to produce partial solutions, but appeared to have a general lack of understanding for programming strategies and the programming processes needed to solve the problems presented.

5.6 Flaws in the Experimental Approach

A number of flaws in the experimental approach were realised during and after the experiment.

5.6.1 Size of Groups

The size of the experimental and control groups was sufficient to test the potential to incorporate explicit programming strategy content into an introductory programming curriculum and the timing of that incorporation. It was sufficient to allow a small number of participants to experience these curricula and be interviewed on their understandings that may have developed through this participation in interviews that followed.

Although the Goal/Plan Analysis of participants' solutions showed differences between the groups, the number of participants was too low to statistically infer the superiority of the experimental curriculum. It is not clear that increasing the size of the participant population would produce consistent reproducible results, which appears to be the bane of many explorations in educational settings (Hirsch, 2002).

5.6.2 Absorbing Concepts Rapidly

Participants in the study were diligent students. All students were able to follow the course materials and achieve results in paper exercises and practical computer tasks. However, expecting completely correct solutions in the final assessment, which involved generation of code for novel problems, appears to have been more than could be expected from students at the end of two days of instruction. Although exercises were given to reinforce the concepts covered, these may not have been as effective as if they were completed days or weeks later.

The result of this experiment shows that the strategy ability of participants exposed to the experimental curriculum produced an average overall correctness of 39% for Problem 2 compared to students who had been exposed to a semester long, traditional introductory course in programming, who achieved an average overall correctness of 57% on effectively the same problem.

5.6.3 Generation of Code can be a Poor Measure

The final assessment asked students to generate code to novel problems, the solutions to which should involve the strategies they had learned in the preceding day and a half. Most of the participants were unable to create complete solutions to these problems. This may be attributable to a lag between

1. exposure to a programming strategy,
2. the ability to comprehend that strategy, and eventually
3. the ability to generate an implementation that applies that strategy.

In this case asking participants to generate code at that stage may have been less effective than gauging their programming strategy skill levels by other means, such as comprehension tests or cases involving errors.

5.7 Implications and Future Work

This experiment showed that it is possible to create a curriculum that explicitly incorporates sub-algorithmic programming strategies. The incorporation of such additional instruction does not create an unfeasible burden of time.

There were also noticeable effects on the students participating in the experiment and exposed to this additional instruction. Participants who covered the experimental curriculum appeared more likely to understand and apply the programming strategies they had been exposed to. These students used terms from a programming strategy vocabulary presented in the curriculum, which could be useful in teaching and assessment if applied to a full scale course. Participants who covered the experimental curriculum claimed confidence in the solutions they had created and their understanding of the strategies used to create them, while students not exposed to this curriculum doubted their abilities.

Some instructors may see these outcomes as encouraging enough to adopt teaching of programming strategies in an explicit manner in full introductory programming courses. An evaluation of a real course with explicitly incorporated programming strategies is planned.

Goal/Plan Analysis is a basic tool for analysing student code and detecting deficiencies in student understanding and, in turn, possible weaknesses in curricula. It has been used here to measure student solutions and as a basis for a deeper exploration of novice understanding. But it appears that its use in this experiment, and in the past, is limited and would not be fully appropriate to assess students at all stages of a full introductory programming course. Multiple forms of assessment are needed to go beyond Goal/Plan Analysis in order to accurately and consistently measure a student's strategy skill during and at the conclusion of a course in introductory programming. Assigning marks to use of strategies in assessments will hopefully encourage students to value this component of the curriculum, devoting study time to programming strategies.

6 References

Cretchley, P. (2006): Does computer confidence relate to levels of achievement in ICT-enriched learning models? In *Education and Information Technologies*. New York, USA: Springer.

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

Incorporating Strategies Explicitly into Curricula (Working Paper), de Raadt, M. http://www.sci.usq.edu.au/research/workingpapers/sc-mc-0705.ps. Accessed May 29 2007.

de Raadt, M. (2007b): A Review of Australasian Investigations into Problem-Solving and the Novice Programmer. *Computer Science Education,* **17**(3):201 - 213.

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.

Hirsch, E. D., Jr. (2002): Classroom Research and Cargo Cults. *Policy Review,* **115**:51 - 69.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin,* **36**(4):119 - 150.

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

Rist, R. S. (1995): Program Structure and Design. *Cognitive Science,* **19**:507 – 562.

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.

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