The Impact of a Single Lecture on Program Plans in First-Year CS*

Francisco Enrique Vicente Castro
WPI Dept of Computer Science
fgcastro@cs.wpi.edu

Shriram Krishnamurthi, Kathi Fisler
Brown Dept of Computer Science
{sk,kfisler}@cs.brown.edu

ABSTRACT

Most programming problems have multiple viable solutions that organize the underlying problem's tasks in fundamentally different ways. Which organizations (a.k.a. *plans*) students implement and prefer depends on solutions they have seen before as well as features of their programming language. How much exposure to planning do students need before they can appreciate and produce different plans? We report on a study in which students in introductory courses at two universities were given a single lecture on planning between assessments. In the post-assessment, many students produced multiple high-level plans (including ones first introduced in the lecture) and richly discussed tradeoffs between plans. This suggests that planning can be taught with fairly low overhead once students have a decent foundation in programming.

KEYWORDS

Plan composition, program design, programming pedagogy

1 INTRODUCTION

Given a programming problem, students make multiple choices in crafting a solution. Some choices focus on lower-level concerns such as which language constructs to use (e.g., a `while` loop versus a `for` loop). Higher-level decisions include how to cluster the subtasks of a problem into individual functions or code blocks. The clustering of subtasks is often called a *plan* [18]. Programming involves (among other things) implementing plan components in lower-level constructs and composing those constructs into a solution for the overall problem.

Planning is not an advanced topic only for upper-level CS students. Even casual programmers who write scripts are affected by planning decisions. A student writing scripts to process data for a lab experiment encounters changing data requirements, noisy data, or other situations that get handled through planning, not just low-level construct choices. Thus, planning is relevant even to students in first-year computing courses, including those who may not take later courses.

*Research partially supported by the US NSF under grant 1116539.

Koli Calling 2017, November 16–19, 2017, Koli, Finland

ACM ISBN 978-1-4503-5301-4/17/11…$15.00
https://doi.org/10.1145/3141880.3141897

But can students learn planning that early and, if so, how? Some researchers have made notable efforts at teaching such concepts from the outset [4, 8], but this requires a concerted effort to make it central to the introductory curriculum. Because these courses serve a variety of needs for the rest of the program, doing so requires buy-in and prioritization from all the other faculty, who may have vastly different needs. Thus, we were interested in whether a lightweight approach to teaching planning could have any effect, or whether only a comprehensive overhaul of the courses—which may be impossible given a department's other needs—would suffice.

Concretely, we assigned students (from two universities) programming problems that could be approached through multiple plans; leveraged those to give a single 50-minute lecture on plans and tradeoffs; and then assigned a new set of programming problems. In the post-assessment, we asked students to produce two solutions to each problem, each embodying a different plan. We also asked students to rank their solutions by preference, so we could see what criteria they used. Nearly all students produced two different plans in the post-assessment, often choosing a general plan structure that was first introduced in the planning lecture. When asked to preference-rank among their solutions, many students chose solutions with a different general structure than what they wrote on the pre-assessment. These results suggest that lightweight instruction in planning can have an impact.

2 RELATED WORK

The task of developing and integrating programming plans has been identified as a recurring problem among programming students [3, 17, 18]. While some recent studies show students succeeding at plan composition in specific contexts [6, 16], the pedagogic choices that help students with this task remain poorly understood.

A growing body of research aims at improving planning skills by explicit instruction. Porter and Calder suggest a process for building a vocabulary of common patterns for guiding students through problem decomposition [14]. Muller, Haberman, and Ginat use this same concept to develop pattern-oriented instruction [10]: attaching labels to algorithmic patterns and presenting various problems to students, while encouraging students to look for common patterns across problems. Our work differs in trying a more lightweight approach, in which planning is the focus of a lecture and two assignments rather than the entire curriculum.

De Raadt, Watson, and Toleman make the explicit distinction between programming *knowledge* (language syntax and semantics) and programming *strategies* [4]. Their 'strategy guide' discusses *abutment*, *nesting*, and *merging* as ways for integrating strategies; their assignments require students to apply specific strategies in their solutions. Our work, in contrast, focuses on *problem-level techniques* (such as cleaning data) rather than code-level techniques

(such as merging code). Our work also asks students to discuss design tradeoffs, as a way of triangulating what they understand about patterns.

3 COGNITIVE FOUNDATIONS OF RECALLING SOLUTIONS

Strategies for teaching planning build on results on how people construct programs at a cognitive level. Given a programming problem, programmers (subconsciously) identify solutions to similar problems and adapt them to the problem at hand [12, 13, 15, 19]. Repeated application of a pattern helps programmers form a mental schema for that problem (which can be recalled later for solving other problems); repeated use of a schema strengthens recall [9, 11]. This basic architecture underlies approaches to teaching patterns explicitly. Research has identified a difference between knowledge schemas and strategy schemas (as both de Raadt *et al.* and Caspersen cite [1, 4]): this difference implies that students may require less direct practice to internalize a new (strategic) pattern that builds on already-internalized knowledge schemas.

The distinction between knowledge and strategy schemas frames our experiment because our lecture shows students new ways to cluster subtasks of programming problems. All of our participants had been writing list traversals prior to the experiment. The lecture showed new high-level ways to decompose a problem into (potentially multiple) list traversals. Our study asks whether students would apply these high-level strategies to new problems based just on the single lecture (without us telling them which solution style to produce, as other pattern-based studies have done [4]).

4 STUDY DESIGN

4.1 Study Components

Our study contained three components:[1]

(1) A pre-assessment (section 4.3) in which students produced solutions to 2–3 programming problems. In one course, students also preference-ranked solutions to 3 *different* problems.
(2) A single 50-minute lecture (section 4.4) on planning and design tradeoffs, framed by the pre-assessment problems. The same professor gave the same lecture in both courses.
(3) A post-assessment (section 4.5) in which students were asked to (a) produce two solutions with different plans to each of 4 programming problems and (b) to preference rank between their solutions (with justification).

We did not ask for multiple solutions on the pre-assessment because we couldn't find a way to explain what would distinguish solutions before doing the intervention lecture.

The questions in the pre- and post-assessment were carefully chosen to introduce some broadly-applicable strategies in multi-task programming problems:

- Noisy data that could be *cleaned* prior to the main computation.
- Flattened data that could be *parsed* (or *reshaped*) to a structure that was better suited to the main computation.
- Data that could be *truncated* to a prefix of interest for the main computation.

[1] Actual pre- and post-assessment questions and a technical report with additional analysis and details [2] are at https://github.com/franciscastro/koli-2017.

In addition, the post-work included computations that targeted a *projection* of the data (say to a specific field within an object). We did not emphasize projection in the pre-assessment as students had experience with this idea from other assignments in both courses.

The lecture discussed cleaning, parsing, and truncating in the context of the pre-assessment problems. We also discussed various design tradeoffs that these offered, including impact on run-time efficiency, ability to adapt the solution to a different dataset, and readability and maintainability of the resulting code.

4.2 The Host Courses

We conducted the study in two first-year CS courses at different universities. Each course was taught by one of the authors. Students in both courses had some prior programming experience, though it differed both across and within the populations.

- CrsA is an accelerated CS1 course that compresses much of the first year into a semester. Most students have prior experience, usually with imperative or object-oriented programming in Java or Python. The course is taught in Pyret, a functional language with syntax reminiscent of Python.
- CrsB is a CS2 course on object-oriented programming and data structures, taught in Java. Students feed into the course from one of two CS1 courses taught in functional programming: one for novice programmers (CrsBnvc), and one for non-novices (CrsBexp). Students from CrsBnvc have seen little to no imperative programming prior to CrsB, while students from CrsBexp have prior experience similar to that of CrsA.

Students in both courses had previously learned functional programming with the *How to Design Programs* [5] curriculum. Prior to the pre-assessment, CrsB had covered both kinds of `for` loops for iterating over Java linked lists. In-class examples of `for`-loops consisted of simple list traversals that accumulated answers (such as summing a given field across a list of objects) or filtering out a subset of elements. The pre-assessment was the first assignment in the course on programming with lists and `for`-loops.

Sampled Populations. There were 75 students in CrsA and 290 in CrsB. While all students completed the study, our (manual) analysis uses a sample based on final course grade. We sampled up to 10 students from each passing grade (A, B, and C) in each of the three populations (CrsA, CrsBnvc, CrsBexp). The grade bands were not significant in our analysis, so we do not discuss them further.

4.3 Pre-Assessment

The programming problems for the pre-assessment came from Fisler *et al.*'s recent study [7]. We used slightly different problems in each course to accommodate different student preparation (such as whether they knew string manipulation).

In CrsA, the pre-assessment consisted of programming solutions to *Palindrome*, *Sum Over Table*, and *Adding Machine*. It also asked students to preference-rank given solutions for the *Rainfall*, *Length of Triples*, and *Shopping Cart* problems.

In CrsB, we used *Rainfall* and *Length of Triples*. We did not present ranking tasks due to time constraints.

We briefly summarize some of these problems. *Rainfall* involves noisy data (negative numbers that should not be averaged) and a

single-character delimiter for the relevant data. *Length of Triples* asks for the longest concatenation of three consecutive elements from a list of strings. *Adding Machine* is described as follows:

> Design a program called `addingMachine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

It features flattened data that could be parsed into a list of sublists and a prefix of data (prior to the consecutive zeros) that could be truncated. Typical solution structures include (1) traversing the data once while accumulating the sum of the current sublist and the output, (2) parsing the input into a list of sublists, with a second traversal to sum the sublists, or (3) using a nested loop to compute sums of sublists.

4.4 The Lecture

Within two days after the pre-assessment was due, Fisler lectured about the problems to each course. The lectures were not identical since the questions were different, but they covered similar content.

In CrsA, the class first discussed the ranking tasks and the tradeoffs students considered. Fisler moderated the discussion, raising each of efficiency, aesthetics, maintainability, and code structure. Fisler showed possible solutions to *Adding Machine*, explicitly discussing parsing and truncating as applicable strategies.

In CrsB, Fisler showed multiple solutions to each of *Rainfall* and *Length of Triples*. The former was used to point out cleaning and truncating as strategies; the latter was used to point out parsing. These solutions were posted for later reference. Again Fisler moderated a classwide discussion of the tradeoffs among the solutions.

In both lectures, the instructor described planning as the general task of allocating subproblems to traversals of the data. While this is a somewhat more code-focused definition that we might otherwise like, it was designed to give students a way to assess whether their two solutions would be "different" from the perspective of the post-assessment, though at this point they did not know what they would be asked to do on that assignment. However, the emphasis of the lecture was not on this definition but on concrete strategies, to help them build a vocabulary of planning operations.

4.5 Post-Assessment

For the post-assessment, we sought problems that resembled the pre-assessment ones, and were amenable to the parsing, cleaning, and truncating strategies discussed in the lecture. The post-assessment contained four problems; students were required to (a) submit two solutions (with different structures) for each problem, and (b) state a preference between their two solutions (with justification). This paper focuses on two of the problems, due to their particular similarities with pre-assessment problems:

Data Smoothing Given a list of health records with a numeric `heartRate` field, design a program `dataSmooth` that produces a list of the `heartRate`s but with each (internal) element replaced with the average of that element and its predecessor and successor. E.g., given a list of health-records with heart rates `[95, 102, 98, 88, 105]`, the resulting smoothed sequence should be

```
[95, 98.33, 96, 97, 105]
```

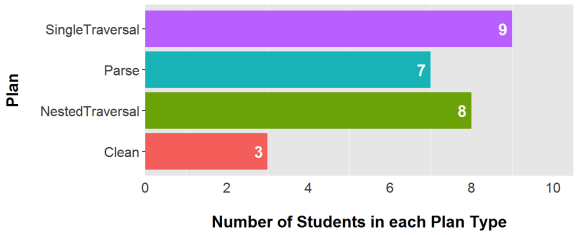

Figure 1: *Adding Machine* structures, pre-, CrsA

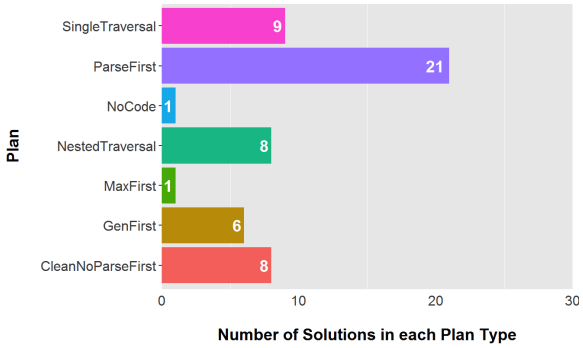

Figure 2: *Earthquake Monitor* structures, post-, CrsA

This problem, like *Length of Triples* from the pre-assessment, looks at 3-element windows within an input list. This problem is a good candidate for parsing. Other strategies include first extracting all the heart-rates from the health records (giving a list of numbers to smooth), or doing the computation in a single input traversal.

Earthquake Monitor Write a program that takes a month and a list of readings from an earthquake sensor. In the input, 8-digit numbers are dates and numbers below 500 are readings for the preceding date. Produce a list of reports containing the highest reading for each date in the given month. E.g., given this list and 10 for the month,

```
[20151004, 200, 150, 175, 20151005, 0.002, 0.03, 20151207]
```

the program should yield `[report(4, 200), report(5, 0.03)]`

Like *Adding Machine*, this problem has sublists within the data, which makes it a candidate for parsing. Like both *Adding Machine* and *Rainfall*, it has a sentinel (in the form of data from a later month). It could be approached with a single traversal that accumulates the max value per date, a cleaning phase that restricts the input data to the desired month, or parsing prior to computing the reports.

5 ANALYSIS

We discuss our findings separately for each course. A direct comparison between the two courses is not meaningful due to differences in the students' backgrounds and the smaller set of questions used with CrsB. We still find value, however, in seeing how two groups of students with some similarities in their backgrounds fared.

5.1 The View from CrsA

The data from CrsA suggest that our lecture had a significant impact on students' planning behavior. Figure 1 and Figure 2 show the

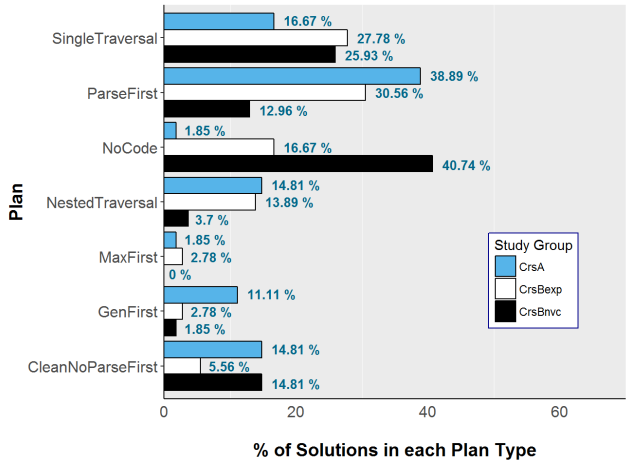

Figure 3: *Earthquake Monitor* structures, post-, all three

structures that CrsA students used in *Adding Machine* on the pre-assessment and *Earthquake Monitor* on the post-assessment, respectively. We contrast these two problems because of their similarity. Students took a variety of approaches in the pre-assessment, with some using parsing. Usage of parsing jumps significantly ($p < .006$ with a McNemar's test) in the post-assessment: about 70% of CrsA students used parsing in one of their two *Earthquake Monitor* solutions. Even in *Data Smoothing*, roughly the same number of students chose to parse as did a single data traversal. Thus, there is strong evidence that CrsA students learned parsing as a strategy.

Significant contrasts also arise when we examine CrsA students' ranking preferences between the two assessments. The following table shows evolution in criteria mentioned per student across the pre- and post-assessments. The table shows that many students both dropped and added criteria.

Criterion	Pre, not post	Post, not pre	Pre and post
Efficiency	31	-	36
Structure	24	9	15
Aesthetics	14	1	35
Maintainability	7	20	3

All in all, the lecture had the impact we hoped for in CrsA: students showed their ability to produce solutions with multiple plans; most students raised more issues when discussing tradeoffs among solutions; and many students changed the solution structures that they preferred in the post-assessment (which is merely a sign that the lecture impacted their thinking, not that their analyses necessarily grew more accurate).

5.2 The View from CrsB

CrsB offers a more nuanced picture of the lecture's impact, which is less significant. We also see interesting differences between CrsBnvc and CrsBexp, and between CrsA and CrsBexp, who had been working in different programming languages despite a fairly common curriculum (and common programming language) just a month or two prior to the study.

Figure 3 contrasts the *Earthquake Monitor* solutions across all three populations in the post-assessment. Two observations jump out. First, a significant percentage of students in CrsB were unable to solve the problem at all (the *"No Code"* group): of the 45 students sampled, 11 turned in no solution (9 from CrsBnvc, 2 from CrsBexp), while another 6 students turned in only one. Of those with only one, half used parsing while the others did a loop-based traversal or nested traversal. In contrast, there was only one *"No Code"* in CrsA. We suspect that the *"No Code"* s came partly from the lack of programming experience in CrsBnvc and partly from students running out of time (*Earthquake Monitor* was the last problem on the assignment, which was due just before a mid-course holiday.)

Setting *"No Code"* aside, the dominant solution structures differ across the populations: *"Parse First"* dominates in CrsA, *"Single Traversal"* dominates in CrsBnvc, while these two are fairly even in CrsBexp. This suggests that parsing strategies may require more programming experience for students to adopt.

On *Data Smoothing*, the two CrsB populations are more similar to each other, with much heavier use of *"Single Traversal"* solutions, especially compared to the dominance of *"ExtractFirst"* solutions in CrsA. Programming language constructs are a likely factor. Most of the CrsA students used a built-in `map` function to extract the heart rates from the health records. While Java 8 provides `map`, it is somewhat clumsy to use and only a handful of CrsB students had been exposed to it. A basic Java `for` loop is straightforward for *Data Smoothing*, so we should hardly be surprised that students used it.

Of the 45 CrsB students, 16 produced two *Data Smoothing* solutions with the same high-level structure. This suggests that many CrsB students didn't really understand the idea of multiple program plans just from the single lecture, or perhaps that the alternate plans for *Data Smoothing* were too subtle for many students. Of the strategies provided by the lecture (cleaning, parsing, and truncating), only parsing applies to *Data Smoothing*; if parsing was indeed too hard for students, they would have been left without named strategies to apply to the problem. The pre-assessment data did not shed light, as all but one student used a *"Single Traversal"* structure to program *Rainfall*. Prior experience is not the explanation either: these 16 students were roughly evenly split between CrsBnvc and CrsBexp. Whether CrsB students would have understood this idea better had they also done ranking tasks on the pre-assessment is a question for future studies.

5.3 Preference Ranking of Own Post Solutions

The post-assessment asked students in both courses to state a preference between the approaches taken in their solutions to each problem. In CrsA, students had a significant preference for parsing first, which suggests at least the ability to recognize the lecture's views on structures that better decomposed plans.

In CrsB, the most interesting finding was the lack of criteria that students mentioned. Of the 45 students sampled: only 15 mentioned any criteria at all; 9 didn't submit a ranking; the other 21 just described their code (6 of these were from CrsBexp, the rest from CrsBnvc). Among the 15 CrsB students who did describe criteria, code structure and aesthetics came up most often (9 and 10 instances, respectively), while efficiency got only 3 mentions.

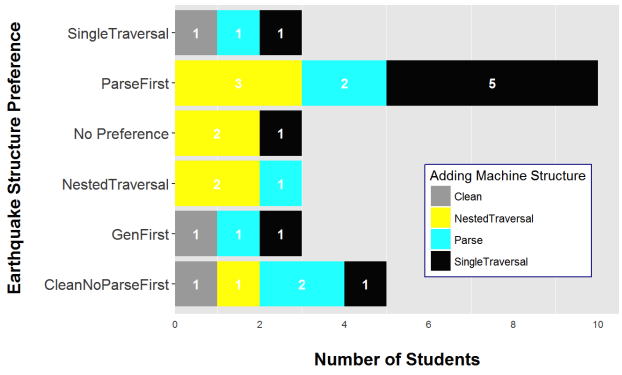

Figure 4: Comparing individual CrsA **students' *Adding Machine* structures (pre) to that of their preferred *Earthquake Monitor* solution (post)**

The contrasts in students' ability to discuss solutions by attributes is striking, and not readily explained. Neither course had practiced this skill, either explicitly or implicitly that the instructors can recall. Both courses had covered rudimentary big-O prior to the pre-assessment, so students at least had "efficiency" in their vocabulary. As such, we cannot yet explain these differences.

5.4 Changes in Solution Structures

Changes in the solution structures that students wrote from the pre- to the post-assessment might indicate that the planning lecture had impact. Since there were two solutions in post-, we compared the structure that students wrote in the pre-assessment with whichever of their solutions they marked as preferred in the post-assessment.

Figure 4 shows the comparison for CrsA, using *Adding Machine* from pre- and the preferred *Earthquake Monitor* solution from post- . Each row shows a structure for the preferred post- binned by their pre- structure. The comparisons are per student. Perhaps most interesting is the lack of a clear internal pattern: students from each *Adding Machine* structure are dispersed across the *Earthquake Monitor* bins, and each *Earthquake Monitor* bin is populated with students from multiple *Adding Machine* structures. Our key takeaway from this graph is that the lecture got CrsA students to think about plans and tradeoffs, with many reconsidering choices that they might have made reflexively during the pre-assessment.

A similar graph for CrsB is not meaningful, since all but one student produced the same structure for *Rainfall* on the pre-assessment. However, we do see diversification in students' preferred solutions in the post-assessment. While *"Single Traversal"* solutions remain the most popular among CrsB students in both the *Earthquake Monitor* and *Data Smoothing* problems on the post-assessment, there is considerable diversity in the *Earthquake Monitor* preferences (which admits more interesting plans). Whether this diversification was caused by the lecture, or by the relatively greater difficulty of *Earthquake Monitor* compared to *Rainfall*, is open to question.

6 DISCUSSION AND FUTURE WORK

As CrsBnvc shows, students seem to need some computing experience before they can embrace planning. We don't know how much or what kind of exposure matters. Some planning strategies may be easier than others for students to appreciate and apply. Does exposure to functional programming also matter? Understanding these nuances would be valuable in creating planning pedagogy.

We do not yet know how to find out what students understand about planning prior to any formal instruction about choosing among solution structures. More broadly, the problem of getting students to discuss solution tradeoffs in terms of criteria rather than code is more general than this study. We wonder whether the ability to comprehend planning correlates with the ability to meaningfully discuss general solution criteria.

This work also proposes a new methodology for studying planning, in the form of writing two different versions of the same program. Broadly, plan-composition seems to be having a resurgence, but the field must start taking a more holistic look at students' planning choices and practices (which in turn could better explain some of the errors that prior studies focused on). We hope this paper will contribute to a larger discussion about research directions in program planning for both majors and non-majors.

REFERENCES

[1] Michael E. Caspersen. 2007. *Educating Novices in the Skills of Programming.* Ph.D. Dissertation. University of Aarhus, Denmark.

[2] Francisco Enrique Vicente Castro, Shriramx Krishnamurthi, and Kathi Fisler. 2017. *The Impact of a Single Lecture on Program Plans in First-Year Computer Science Courses.* Technical Report WPI-CS-TR-17-02. WPI Department of Computer Science.

[3] Michael de Raadt, Mark Toleman, and Richard Watson. 2004. Training Strategic Problem Solvers. *SIGCSE Bull.* 36, 2 (June 2004), 48–51.

[4] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Australasian Computing Education Conference.* Australian Computer Society, Inc., 45–54.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs.* MIT Press. http://www.htdp.org/

[6] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the International Conference on Computing Education Research.* ACM, 35–42.

[7] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* ACM, 211–216.

[8] David Ginat. 2009. Interleaved Pattern Composition and Scaffolded Learning. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education.* ACM, 109–113.

[9] S.P. Marshall. 1995. *Schemas in Problem Solving.* Cambridge University Press. https://books.google.com/books?id=iKSYrsXe6f0C

[10] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education.* ACM, 151–155.

[11] A. Newell, P.S. Rosenbloom, and J.E. Laird. 1989. Symbolic Architectures for Cognition. In *Foundations of Cognitive Science.* MIT Press, 93–131.

[12] Peter L. Pirolli and John R. Anderson. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240–272.

[13] Peter L. Pirolli, John R. Anderson, and Robert G. Farrell. 1984. *Learning to program recursion.* 277–280.

[14] Ron Porter and Paul Calder. 2003. A Pattern-based Problem-solving Process for Novice Programmers. In *Proceedings of the Australasian Computing Education Conference.* Australian Computer Society, Inc., 231–238.

[15] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989), 389–414.

[16] Otto Seppala, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem is?. In *Proceedings of the Koli Calling Conference on Computing Education Research.* ACM, 87–96.

[17] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. *Learning and Teaching in Computing and Engineering* (2013), 130–135.

[18] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858.

[19] James C. Spohrer and Elliot Soloway. 1989. *Simulating Student Programmers.* Morgan Kaufmann Publishers Inc., 543–549.