Problems Encountered by Novice Pair Programmers

BRIAN HANKS
Fort Lewis College

In a study of the types of problems encountered by students that led them to seek assistance, Robins et al. [2006] found that the most common problems were related to trivial mechanics. The students in this study worked by themselves on their programming exercises. This article discusses a replication of the Robins et al. study in which the subjects pair programmed. The types of problems encountered by the pairing students were similar to those of the solo students. The number of problems requiring assistance was much smaller for the pairing students, which suggests that they were able to resolve more problems on their own.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Experimentation, Measurement

Additional Key Words and Phrases: Computer science education, CS1, errors, novice, pair programming

ACM Reference Format:
Hanks, B. 2008. Problems encountered by Novice Pair Programmers. ACM J. Educ. Resour. Comput. 7, 4, Article 2 (January 2008), 13 pages. DOI = 10.1145/1316450.1316452. http://doi.acm.org/10.1145/1316450.1316452.

1. INTRODUCTION

Learning to program is difficult for many students. In an effort to better understand the difficulties encountered by students, Robins et al. [2006] and Garner et al. [2005] examined the types of problems that led students to ask for assistance. They argue that a better understanding of the types of problems that students have can lead to better pedagogy that targets these learning difficulties. Although the students in their study encounter many types of problems,

Author's address: B. Hanks, Computer Science Information Systems, Fort Lewis College, 1000 Rim Drive, Durango, CO 81301. hanks_b@fortlewis.edu.
This paper originally appeared in the Proceedings of the 3rd International Workshop on Computing Education Research, 2007, Atlanta, Georgia, USA.

the most commonly occurring problems are those related to basic mechanics. This includes problems such as missing semicolons, unbalanced braces, or issues related to case-sensitivity.

Students in the Robins et al. [2006] study worked alone on their programs. Recent research on pair programming (in which two students work on their programs together at one computer) has found that students who pair in their introductory programming course are more successful, more confident in their work, and more likely to continue in a computing-related major [McDowell et al. 2006]. Students who pair in Computer Science (CS)1 are also more likely to turn in solutions to their programming assignments and these solutions are more likely to compile [Hanks et al. 2004].

These results suggest that students who pair are able to overcome more problems on their own (there is some anecdotal evidence that supports this conjecture [Nagappan et al. 2003]). One might also expect that the types of problems that cause pairing students to seek assistance would differ from the types encountered by soloing students. Perhaps pairing students are able to overcome most trivial mechanical problems and only seek help on more substantial roadblocks.

It is not known whether the problems reported by pairing students mirror those reported by Robins et al. [2006], or if the distribution differs in some significant way. If, for example, students who pair have many fewer problems with basic mechanics, this would suggest that pair programming helps students progress beyond low-level syntax issues and allows them to focus more on problem solving.

This article describes a replication of the Robins et al. [2006] study , except that the subjects used pair programming instead of working by themselves. The goal of this study was to answer two questions:

(1) do students who pair program have the same types of problems?
(2) do pairing students have as many problems as soloing students?

Two hypotheses were developed to investigate these questions. The first hypothesis postulates that the types of problems on which students need assistance will be more substantive. The second hypothesis predicts that pairing students will be able to resolve more problems on their own.

H1 The proportion of trivial mechanical problems will be smaller for paired students.
H2 Paired students will require assistance on fewer problems.

2. METHODOLOGY

The subjects in the Robins et al. [2006] study were approximately 470 students enrolled in introductory Java programming courses at the University of Otago in New Zealand. These students participated in 25 closed labs.

The subjects of this study were students enrolled in two sections of an introductory Java programming course at Fort Lewis College (FLC) in the 2006 academic year. I was the instructor for both sections. All students in this

course pair programmed. There were 15 pairs in the two sections: eight in the fall semester and seven in the winter semester.

The course is taught using an objects-first approach with the BlueJ IDE. It is presented in a combined two-hour lecture/lab format where the lecture is immediately followed by a closed-lab exercise (supervised by the course instructor), which gives students hands-on experience with the lecture material. In the initial labs, students modify existing classes; later they create classes on their own and finally develop small groups of interacting classes. There were 31 lab exercises which covered the following topics in approximately this order:

—introduction to the BlueJ environment
—accessor and mutator methods
—Boolean expressions and conditionals
—methods and parameter lists
—collections and generics
—iteration
—arrays
—string class
—scanner class
—two-dimensional arrays
—the `main` method
—unit testing and JUnit
—StringBuffer class
—file I/O

Although the approach and sequencing differs, the set of topics covered is similar to that used in the courses investigated by Robins et al. [2006] (hereafter referred to as the Otago study). Material covered here but not in Otago includes unit testing, collections, and the StringBuffer classes. The Otago course included material on GUIs, applets, object hierarchy, and abstract classes that was not covered at FLC.

The FLC students worked in pairs on the 31 lab exercises. Students typically worked with the same partner but would work with another student if their partner was absent. The Otago subjects worked alone on 25 lab exercises which were held in closed labs supervised by teaching assistants. The lab periods were distinct from the lecture.

A procedure similar to that used in the Otago study was followed. Whenever a pair asked for assistance during closed lab, a short summary of their problem was recorded in a log book. At the end of each semester, these problem descriptions were analyzed and a problem code assigned to each one. The problems were categorized using a slightly modified version of the coding scheme used in the Otago study. Two problem codes were added to address some problem types that could not occur in Otago due to the different topics covered. The new codes pertain to problems encountered with Java collections (code S22)

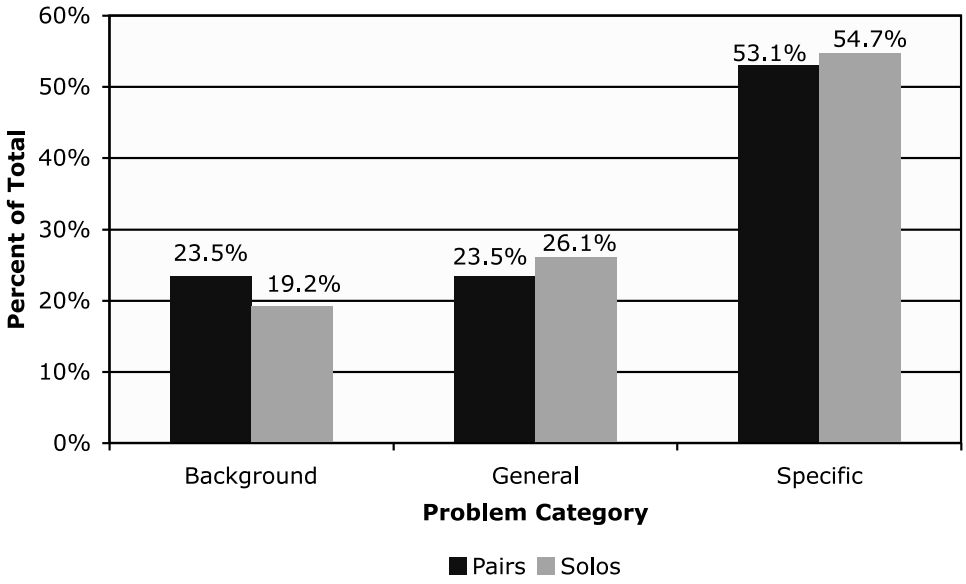

Fig. 1. Problem distribution by category.

or the StringBuffer class (S23). For similar reasons, problem codes S19, S20, and S21 were not used in the FLC study. The coding scheme is reproduced in the Appendix.

Each problem code represents a class of problems that students encountered during the lab exercises. These problems fall into three main categories: (1) background problems, (2) general problems, and (3) specific problems. Background problems include those related to understanding the assignment and using the tools. General problems include basic misunderstanding of object concepts and trivial mechanical problems. Specific problems are those that are associated with particular language constructs such as arrays, loops, or expressions.

3. RESULTS

3.1 Types of Problems

Because the raw data for the Otago study was unavailable, it was necessary to estimate the number of problems of each type. Estimated problem counts (to the nearest 25) were made using the graphs in Figs. 1 and 2 of Robins et al. [2006]. The resulting problem counts ranged from 125 (for codes S16 and S21) to 3,075 (for code G4). The counts for the Otago and FLC students were then converted to percentages to facilitate comparison of the two groups.

Figure 1 shows the problem distributions by category for the paired and solo students. The difference in these distributions was not statistically significant ($\chi^2(2) = 3.92$, $p = .14$).

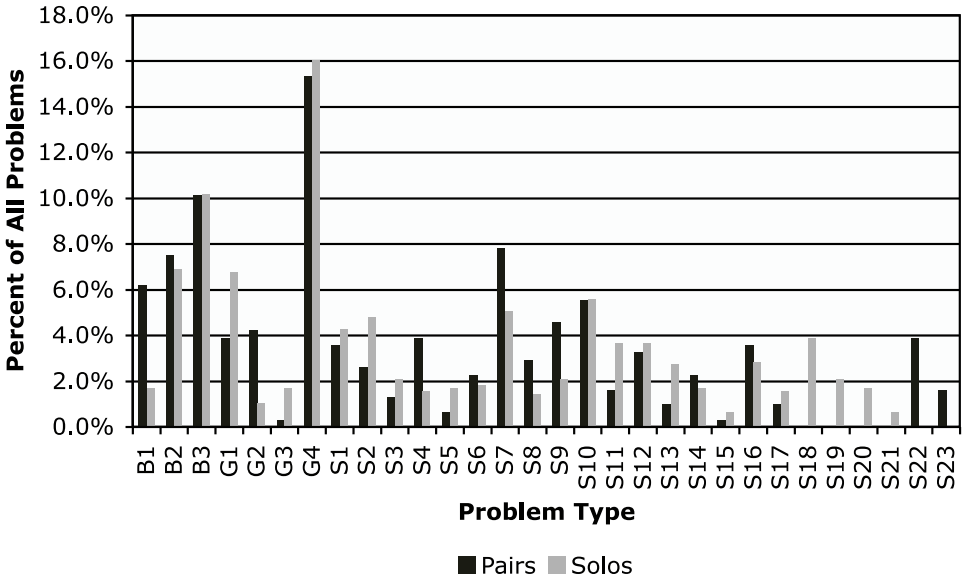

Fig. 2. Problem distribution.

Table I. Problem Summary

	Solo	Pair
n	470	15
Number of Problems	19008	331
Problems per Solo/Pair	40.43	20.73
Problems per (Solo/Pair) per Lab	1.62	0.67

The problem distribution for all problems is shown in Fig. 2. Excluding Problems S19 through S23, which were not relevant to both studies, the distributions did not significantly differ ($\chi^2(24) = 16.32$, $p = .88$). There were also no significant differences between the two studies when comparing the distributions of background, general, or specific problems independently.

Of particular interest are trivial mechanical problems (problem code G4) such as missing semicolons, mismatched braces, and using a single "=" sign instead of "==" to compare two values. These comprised 16.0% of all problems for the Otago students and 15.4% for the FLC students. This was the most commonly occurring problem for both groups.

Another significant source of problems for both groups was not knowing how to get started (Problem B3): This accounted for approximately 10% of all problems requiring assistance. Students understood what the assignment was asking them to do, but had no idea what approach to take toward its solution.

3.2 Number of Problems

The pairing students in this study required assistance on 311 problems. This compares with 19,008 problems encountered by students in the Otago study.

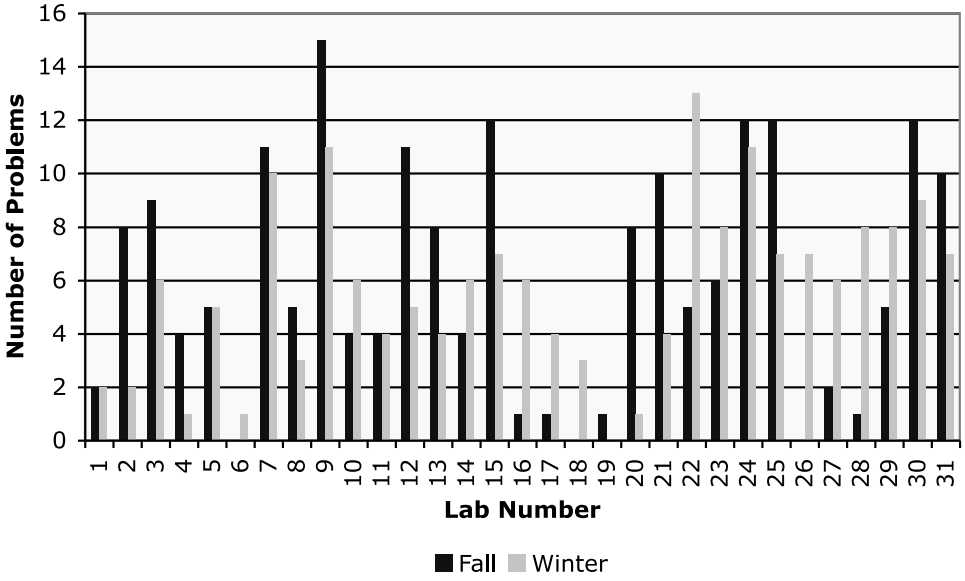

Fig. 3. Problems per lab at FLC.

A typical solo student needed assistance on 40.43 problems over the course of the 25 lab exercises. Pairs encountered an average of 20.73 problems over the course of 31 labs. This means that a solo student had on average 1.62 problems per lab, while pairs encountered 0.67 problems per lab. This is summarized in Table I.

As can be seen in Fig. 3, there was a lot of variability in the number of problems encountered by the pairing students in any of their labs. The difference in distributions between the fall and winter semesters was statistically significant ($\chi^2(30) = 52.15$, $p < .01$), although this result should be interpreted cautiously due to the large number of data points with values less than 5.

Regardless, some of the labs appear to be consistently difficult (such as labs 7, 9, and 24) while others are relatively easy. In the fall semester, the largest number of problems for which students required help in any lab was 12. In the winter semester, there was 1 lab in which students asked for help 13 times. In both semesters, there were labs in which the students did not ask for any help. This occurred twice in the fall semester and once in the winter semester.

4. ANALYSIS

Hypothesis H1 conjectured that the proportion of trivial mechanical problems that led students to seek assistance would be smaller for paired students. This was based on the expectation that paired students would be able to help each other figure out these seemingly simple problems and therefore would only require assistance to resolve more substantial problems.

This hypothesis was not supported by this study. As shown in Figs. 1 and 2, pairing and soloing students exhibited similar problem distributions. Specifically, the percentage of trivial mechanical problems was nearly identical for both groups.

Hypothesis H2 postulated that pairing students would need assistance on fewer problems. This hypothesis appears to be supported by this study. On average, each pair needed help on only 51% as many problems as students who worked alone. On a per-lab basis, pairs needed help on only 41% as many problems. When examined on a *per-student* basis, pairing students needed help approximately 25% as often as soloing students.

On a few of the labs, pairing students only asked a small number of questions (in three cases they did not ask for any help at all). In these cases the pairing students used their collective skills to resolve nearly all problems on their own.

There were a few labs that appear to be more challenging such as labs 7, 9, and 24. Although pairing students may require less help than soloing students, they are not able to overcome all problems. It may be that these labs are more intellectually challenging or require greater cognitive leaps than some of the other labs. These results may also be a sign that these labs are not as clearly defined as some of the others.

4.1 Threats to Validity

There are a number of factors that may affect the generalizability of this study.

The lab exercises differed between the two sites, which is likely to have affected the distribution of problem types. In fact, this is probably a major source of distribution variance for the specific problems codes (S1 to S23). However, it appears that this did not have a major effect on problem categorization—as seen in Fig. 1, students in both groups needed assistance on problems in the background, general, and specific categories in similar proportions.

The difficulty of the labs may have differed between the two studies which could have led to the paired students needing less assistance. However, if this were the case, it seems likely that the category ratios would have differed between the two groups. The number of specific problems would probably have been smaller due to easier problems. On the other hand, the number of trivial mechanical problems would have stayed constant because these problems are related to difficulties with language syntax and not to the problem being solved. Because the two distributions were similar in terms of the proportions of problems encountered in each category, it appears that the lab exercises were of similar difficulty.

Problem codes might have been assigned inconsistently between the 2 sites. An attempt was made to address this by asking one of the Otago study authors for guidance when assigning codes in the first semester. Although this could have affected the distribution of problem types, it has no effect on the number of problems encountered by students.

This study is based on a small number of pairs so the true behavior of pairs may not have been exhibited. Additionally, the variance between the fall and winter semesters in terms of the number of problems per lab may also be associated with the small sample size. It is unclear what effect this may have had on the results.

There were different instructors at the 2 institutions, and teaching quality may have differed. This could have affected both the distribution and quantity of problems requiring assistance.

There could have been different types of students at the two sites, although both groups seem to be representative of novice programmers in general. If one group possessed greater programming skill than the other, this would have had a significant impact on the type and number of problems for which assistance was required.

Although the topics covered in the two courses were similar, the teaching approach differed. An objects-early approach was used at FLC, while a more imperative approach was used at Otago. It is not clear what, if any, difference this would have made on the types or numbers of problems encountered by novices. There is little evidence that one approach is simpler than the other [Lister et al. 2006], which suggests that this was not a factor in the types of problems encountered by students.

5. DISCUSSION

It has been known for a long time that learning to program is difficult. (See, e.g., the collection edited by Soloway and Spohrer [1989].) Recent studies show that students still find it very challenging [McCracken et al. 2001; Lister et al. 2004]. While pair programming has been shown to provide many pedagogical benefits to students who are learning to program, it appears that pairing students still struggle with the same types of problems as students working by themselves.

It seemed plausible that novice pairs would be able to solve more low-level syntax problems on their own, requiring assistance only on more substantive problems. Surprisingly, in this study, the types of problems that led students to ask for assistance were very similar for both paired and solo students. As seen in Fig. 1, the ratio of background, general, and specific problems was nearly identical for both groups. Furthermore, the percentage of all problems that were related to trivial mechanics was very similar for solos and pairs.

On a more positive note, students who pair appear to get stuck less often than students who work alone. The paired students in this study were able to resolve more problems in each problem category but the underlying ratio of problem categories was unchanged. This surprising result leads to some interesting questions: Is there a fundamental ratio of problem types? Do novices get stuck on trivial mechanical problems in a similar ratio regardless of the instructional approach? What is the distribution of problems encountered by experienced programmers?

Pair programming is an effective tool that increases student success, confidence, and retention. This study indicates that pairing students are able to

resolve more problems on their own which seems likely to improve student confidence and self-esteem. However, pair programming is not a panacea; students still need help and it is important to provide an environment in which that help can be given.

APPENDIXES

The problem types/codes listed in this appendix are from Robins et al. [2006] with the addition of specific Problems S22 and S23.

B—Background Problems

B1 Tools
Problems with the Mac, OS X, directories (lost files), jEdit, Applet runner, or other basic tools. Includes being unable to find the resources described in the lab book, but not other kinds of general lab book/text book issues (do not record these). Does not include Java/file naming conventions (Problem G4).

B2 Understanding the Task
Problems understanding the lab exercise/task or its solution. In other words, whatever other problems they may be having, in this case, they don't actually know what it is that the program is supposed to be doing. (Does not include minor clarifications of some detail which do not need to be recorded.)

B3 Stuck on Program Design
They understand the task/solution (it's not Problem B2) but can't turn that understanding into an algorithm or can't turn the algorithm into a program. Cases such as "I don't know how to get started" or "what classes should I have?" or "what should the classes do?".

G—General Problems

G1 Problems with Basic Structure
They have a general design and classes (it's not Problem B3) but are getting basic structural details wrong: for example, code outside methods, data fields outside the class, data fields inside a method/confused with local variables. Meant to capture problems at the class/major structural level—problems specifically with data fields or about or within methods (e.g., mixing up loops) will be some other problem code.

G2 Problems with Basic Object Concepts
Covers very basic problems with creating and using instance objects: for example, how many, what they are for (but not more specific problems, for example, with class versus instance Problem S16, or constructors Problem S18).

G3 Problem Naming Things
They have problems choosing names for things, especially where this seems to suggest that they don't understand the function of the thing that they are trying to name.

G4 Trivial Mechanics
Trivial problems with little mechanical details (where these are not better described by some other problem). Braces, brackets, semicolons. Typos and spelling. Java and file-naming conventions. Import statements (when forgotten or misunderstood, see Problem S12). Formatting output. Tidiness, indenting, comments.

This category covers only trivial problems (e.g., accidentally mismatched {}). When the underlying issue is actually a conceptual one (e.g., they don't understand the structure that the {} describe) use the best matching Specific Problem.

S—Specific Problems

S1 Control Flow
Problem with basic sequential flow of control, the role of the main or init method. Especially problems with the idea of flow of control in method call and return (e.g., writing methods and not calling them, expecting methods to get called in the order they are written). (For issues with parameter passing and returned results, see Problem S7.) Does not include event-driven model, Problem S21.

S2 Loops
Conceptual and practical problems relating to repetition, loops (including for loop headers, loop bodies as blocks).

S3 Selection
Conceptual and practical problems relating to selection, if else, switch (including the use of blocks).

S4 Booleans and Conditions
Problems with Booleans, truth values, Boolean expressions (except Boolean operator precedence, see Problem S13). Problems with loop or selection headers/conditions will have to be judged carefully—is this a problem formulating the Boolean expression (Problem S4) or understanding how the expression/result is relevant to the loop or selection (Problems S2, S3)?

S5 Exceptions, Throw Catch
Problems with exceptions, throw catch.

S6 Method Signatures and Overloading
Problems related to overloading. Failure to understand how method signatures work/which version of a method gets called. (Includes problems with constructors that are really about the signatures of constructors; cf. Problem S18).

S7 Data Flow and Method Header Mechanics
Especially conceptual problems with arguments/parameters and return types/values. Includes problems with method header mechanics (incorrect or mismatching parameter specifications, incorrect return types, or use of void). Includes any other problems with data flow that are not better described by Problem S8.

S8 Terminal or File IO
Problems with terminal or file IO/data flow (not including exception handling Problem S5, or output formatting Problem G4).

S9 Strings
Strings and string functions. Does not include formatting output (Problem G4) or problems relating specifically to strings as reference types (Problem S15).

S10 Arrays
Problems relating to arrays as a data structure, including array subscripts, array contents, array declaration, and initialization (cf. Problem S11). Does not include failing to understand that an array as a whole is itself a reference type or may contain references (Problem S15).

S11 Variables
Problems with the concept of or use of variables. Includes problems with initialization and assignment. (Missing the distinction between a data field and a local/method variable is Problem G1.) Does not include cases more accurately described as problems with reference types (Problem S15) or arrays (Problem S10) rather than the concept of a variable.

S12 Visibility and Scope
Problems with data field visibility, local variable scope (e.g., defining a variable in one block and trying to use it in another, problems arising from unintended reuse of identifiers), and namespace/imported package problems (but not including forgotten import statement, Problem G4). Includes cases confusing data fields and variables of the same name but not where this is better described as a failure to understand this (Problem S15).

S13 Expressions and Calculations
Problems with arithmetic expressions, calculations, notation such as ++ and all forms of precedence (including Boolean operator precedence; cf. Problem S4).

S14 Data Types and Casting
Problems caused by failing to understand different data types and casting for primitive types (reference types are Problem S15).

S15 Reference Types
Problems arising from a failure to understand the concept or use of reference types (references/pointers, “this”, different references to the same object, etc.) or that reference types behave differently from primitive types (when assigned, compared, etc.).

S16 Class Versus Instance
Problems understanding the class object versus instance object distinction, including problems with class and instance data fields (and use of static).

S17 Accessors/Modifiers
Specific problems (cf., e.g., Problem S7) with the concepts of/purpose of an accessor or a modifier method.

S18 Constructors
Specific problems (cf., e.g., Problems S6, S7, S16) with the concept of/purpose of a constructor.

S19 Hierarchies
Problems relating to hierarchical structure, inheritance (extends, overriding, shadowing, super), and issues relating to the use of abstract methods or interfaces.

S20 GUI Mechanics
Problems with GUIs and the use of AWT, Swing etc. Includes problems with specific required methods such as actionPerfomed(), run(), implements actionListener, and so on (but some issues might involve general problems with the concept of interfaces, Problem S19). Does not include the underlying concepts of event-driven programming.

S21 Event-Driven Programming
Problems with the underlying concepts of event-driven programming, and general flow-of-control type issues that arise in the transition from application to applet.

S22 Collections
Problems with the underlying concepts of collections and how they work. Misunderstanding the distinction between the type of collection and the type of its elements. Problems using collection methods to insert/extract elements.

S23 StringBuffer
Problems related to use of StringBuffer objects. Failure to understand the distinction between String and StringBuffer. Problems associated with changeable content of StringBuffer. Problems associated with index values changing as StringBuffer objects are modified.

REFERENCES

GARNER, S., HADEN, P., AND ROBINS, A. 2005. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education (ACE'05)*. 173–180.

HANKS, B., MCDOWELL, C., DRAPER, D., AND KRNJAJIC, M. 2004. Program quality with pair programming in cs1. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)*. 176–180.

LISTER, R., ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPÄLÄ, O., SIMON, B., AND THOMAS, L. 2004. A multinational study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'04)*. 119–150.

LISTER, R., BERGLUND, A., CLEAR, T., BERGIN, J., GARVIN-DOXAS, K., HANKS, B., HITCHNER, L., LUXTON-REILLY, A., SANDERS, K., SCHULTE, C., AND WHALLEY, J. L. 2006. Research perspectives on the objects-early debate. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'06)*. 146–165.

MCCRACKEN, M., ALMSTRUM, V., DIAZ, D., GUZDIAL, M., HAGAN, D., KOLIKANT, Y. B.-D., LAXER, C., THOMAS, L., UTTING, I., AND WILUSZ, T. 2001. A multinational, multiinstitutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'01)*. 125–180.

MCDOWELL, C., WERNER, L., BULLOCK, H. E., AND FERNALD, J. 2006. Pair programming improves student retention, confidence, and program quality. *Comm. ACM 49,* 8, 90–95.

NAGAPPAN, N., WILLIAMS, L., FERZLI, M., WIEBE, E., YANG, K., MILLER, C., AND BALIK, S. 2003. Improving the CS1 experience with pair programming. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'03)*. 359–362.

ROBINS, A., HADEN, P., AND GARNER, S. 2006. Problem distributions in a CS1 course. In *Proceedings of the 8th Australian Conference on Computing Education (ACE'06)*. 165–173.

SOLOWAY, E. AND SPOHRER, J. C., eds. 1989. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Inc., Hillsdale, N.J.

Received October 2007; accepted October 2007