Informatics in Education, 2005, Vol. 4, No. 1, 69–86

Take Note: the Effectiveness of Novice Programmers' Annotations on Examinations

Robert MCCARTNEY
Department of Computer Science and Engineering, University of Connecticut
Storrs, CT 06269, USA
e-mail: robert@engr.uconn.edu

Jan Erik MOSTRÖM
Department of Computing Science, Umeå University
905 86 Umeå, Sweden
e-mail: jem@cs.umu.se

Kate SANDERS
Department of Math and Computer Science, Rhode Island College
Providence, RI 02908 USA
e-mail: ksanders@ric.edu

Otto SEPPÄLÄ
Laboratory of Information Processing Science, Helsinki University of Technology
02015 TKK, Finland
e-mail: oseppala@cs.hut.fi

Received: December 2004

Abstract. This paper examines results from a multiple-choice test given to novice programmers at twelve institutions, with specific focus on annotations made by students on their tests. We found that the question type affected both student performance and student annotations. Classifying student answers by question type, annotation type (tracing, elimination, other, or none), and institution, we found that tracing was most effective for one type of question and elimination for the other, but overall, any annotation was better than none.

Key words: student assessment, tracing, annotation, code reading, test strategies.

1. Introduction

In summer, 2004, a working group at the ITiCSE conference in Leeds, UK, examined the code reading and understanding of novice computer programmers. The analysis was based on data collected from twelve institutions in Australia, Denmark, England, Finland, New Zealand, Sweden, the United States, and Wales. These data were based on a multiple-choice test administered to beginning programming students, and included the student answers for all of the questions. For a subset of the students, interview transcripts

and the actual test forms with any annotations used during the test were also collected. The working group analyzed a broad range of issues: the kinds of questions, the performance of students by institution and quartile, the sorts of annotations used and their general effectiveness, student test-taking strategies observed, and others(Lister *et al.*, 2004).

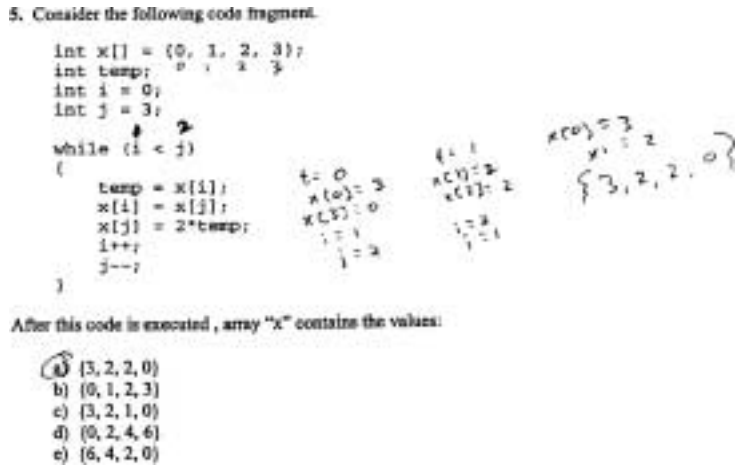
In this paper, we look in detail at a small slice of these issues: the interrelationships observed between the kinds of annotations used by the students, the style and difficulty of the individual questions, and the institutions where the students were tested. In particular, we would like to determine how the likelihood of answering a question correctly is affected by the various kinds of annotations used.

2. Classifying Annotations

An annotation, referred to as a "doodle" by the Working Group, was defined to be any kind of marking by a student on his or her exam paper. In Fig. 1 we can see an example where the user made a number of marks: numbers written over variables, numbers written under array elements, and many assignment statements setting variables to constants. In this case, the assignments show the student keeping track of variables as the loop is executed – a graphical record of the tracing process.

Two of the Working Group members did a data-driven classification of the doodles. This classification was later independently verified by three other members (see (Lister *et al.*, 2004) for details). The classification is given in Table 1. Using this, the researchers classified 56 complete exams: the three from each institution corresponding to the students who were interviewed about their tests, plus 20 others chosen at random from the six researchers who had other tests with them in Leeds. All of the results in this paper are based on these 56 exams.

As the example in Fig. 1 shows, an answer can contain several different doodle categories: N, P, and T for this question.



5. Consider the following code fragment.

```
int x[] = {0, 1, 2, 3};
int temp;
int i = 0;
int j = 3;

while (i < j)
{
    temp = x[i];
    x[i] = x[j];
    x[j] = 2*temp;
    i++;
    j--;
}
```

After this code is executed, array "x" contains the values:

a) {3, 2, 2, 0}
b) {0, 1, 2, 3}
c) {3, 2, 1, 0}
d) {0, 2, 4, 6}
e) {6, 4, 2, 0}

Fig. 1. Example of doodles on test from question 5, showing annotations N, P, and T.

Table 1

Categorization of annotations. % is percentage of questions showing each type

Name	Code	Description	%
Blank page	B	No annotations for this question	38
Synchronized trace	S	Shows values of multiple variables changing, generally in a table.	11
Trace	T	Shows values of a variable as it changes (more than 1 value) or a variable's value is overwritten with new value	32
Odd Trace	O	Appears to be a trace but neither S nor T, such as linking representations with arrows	3
Alternate answer	A	Student changed their answer to the question	4
Ruled out	X	One or more alternative answers crossed out, answer appeared to be selected by elimination	9
Computation	C	An arithmetic or boolean computation (not rewrite of comparison)	4
Keeping tally	K	Some value counted multiple times, variable not identified	1
Number	N	Shows single variable value, most often in comparison	28
Position	P	Picture of correspondence between array indices and values	11
Underlined	U	Part of question underlined for emphasis	7
Extraneous marks	E	Markings that appear meaningless or ambiguous (could not be characterized). Includes arrows, dots, and so forth	13

3. Classifying the Exam Questions

The multiple-choice exam used for this project consisted of twelve questions involving a variety of array-processing tasks, such as comparing two arrays in order to find elements that are contained in both, testing to determine whether an array is sorted, filtering some of the elements of one array into another, searching for a given element in an array, and deleting the element in a given position from an array.

Each question on the test can be classified into one of two categories, *fixed-code* questions, where the student is given a code fragment and asked questions about the result of executing it, and *skeleton-code* questions, where the student is given an incomplete code fragment, and asked to complete it so it will perform a given task. There were seven fixed-code and five skeleton-code questions on the exam; these are given in Appendices A and B respectively.

Fixed-code questions present a single piece of code and ask the student about what is true after the code is executed. The style of all the fixed-code questions is the same: *"Consider the following code fragment:* [code fragment] *What is the value of* [some int or array variable] *after this code is executed?"*, and each of the possible answers is a constant, either the value of an integer variable (as in questions 1, 2, 3, 4, and 7) or a list

Table 2

Percentage of questions answered correctly and percentage of questions with annotations observed, by question. These are based on analysis of 672 questions (56 of each question)

	Fixed-code							Skeleton-code				
	1	2	3	4	5	7	10	6	8	9	11	12
% annotated:	68	79	71	86	88	73	71	41	45	39	45	38
% correct:	68	61	71	57	79	66	68	54	45	68	66	45

of the values in an array (as in questions 5 and 10). Other than the code, they require very little reading.

In contrast, *skeleton-code questions* provide a code fragment containing one or more blanks, a description of what the code is to accomplish, and a set of choices with which to fill the blanks. These choices are all code fragments themselves, as opposed to the numeric values used in the answers to the fixed-code questions. These are longer to read, as there is a description of the intent of the code (see, in particular, questions 8 and 11), and the answers are longer since they are code fragments.

The students' performance was noticeably different on the two types of questions. Table 2 shows the percent of correct answers broken down by question. It shows that students in general did better on the fixed-code questions than on the skeleton-code questions. On the fixed-code questions, from 57% to 79% of all the answers were correct (depending on the question), with a mean score of 67%. On the skeleton code questions, 45% to 68% of the answers were correct, with a mean score of 55% – a difference of 12% on average. The overall average exam score was 62%, which indicates that the students do not have a strong grasp of the basic knowledge in these areas.

4. Analysis

Two issues complicated our analysis of these data. First, the large majority of the annotations were done on the fixed-code questions, as can be seen in Table 2. The difference between the fixed-code and skeleton-code questions is striking: 77% of the fixed-code questions are annotated, as opposed to 41% of the skeleton-code questions. Indeed, if we were to group the questions on the basis of how often they are annotated, without looking at the questions themselves, we would have the same groups: Questions 1–5, 7, and 10, the fixed-code questions (68%–88% annotated) and Questions 6, 8–9, and 11–12, the skeleton-code questions (38%–45% annotated).

Because the two groups were annotated so differently, there was the danger that overall conclusions would be determined by the data from the annotations of the fixed-code questions. Accordingly, we considered the fixed-code and skeleton-code questions both together and as two separate groups.

The second issue that complicates this analysis is that the classifications are not disjoint; with the exception of Blank, any combination of annotations can occur on any

given question. The observed non-blank questions had from 1 to 5 different annotation types represented, with an average of approximately 2. As an example of this problem, consider class N, numbering, which was relatively common (appearing in just over 28% of the questions). 90% of the time that there was numbering, however, there were other classes as well; 77% of the time at least one tracing type was also present.

To resolve this issue, we created four disjoint categories, reclassifying each question as Blank, Some Tracing (S, T, and O, but not A or X), Elimination (A or X), or Other (everything else). The rationale for these categories is that tracing and process of elimination are recognizable strategies, and, with Blank, cover 89% of the observations.

We then counted the number and percentage of time the answers were correct for each category, for fixed-code questions, skeleton-code questions, and for all questions taken together. The results are given in Tables 3 and 4.

These tables illustrate a number of things:

1. Overall, answers showing explicit tracing are the most likely to be correct: 75% are correct, compared with 50% for questions without annotation. We see similar results for FC and SC questions handled separately.
2. Overall, elimination is the second-most effective strategy.
3. Across the board, both overall and for the fixed-code and skeleton-code subsets of the data, any form of annotation, even Other, is better than no annotation at all.
4. The frequency of tracing is much lower for skeleton code questions than fixed code questions.Although tracing is used less, it is still highly effective for the skeleton-

Table 3

Percentages of annotation types for each question type. Numbers in parentheses present the counts (questions having this annotation, all questions)

	fixed-code	skeleton-code	all questions
Blank	23 (92 of 392)	59 (164 of 280)	38 (256 of 672)
Some tracing	61 (238 of 392)	6 (17 of 280)	38 (255 of 672)
Elimination	6 (25 of 392)	21 (59 of 280)	13 (84 of 672)
Other	9 (37 of 392)	14 (40 of 280)	11 (77 of 672)

Table 4

Percentages correct, by question and annotation type. Numbers in parentheses present the counts (correct and total answers)

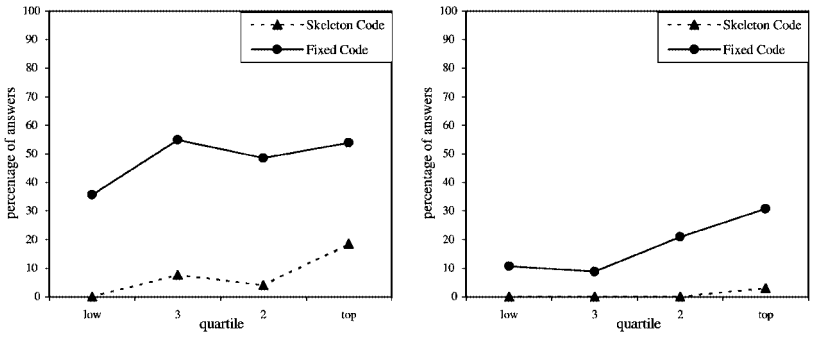
	fixed-code	skeleton-code	all questions
Blank	50 (46 of 92)	50 (82 of 164)	50 (128 of 256)
Some tracing	76 (180 of 238)	65 (11 of 17)	75 (191 of 255)
Elimination	48 (12 of 25)	61 (36 of 59)	57 (48 of 84)
Other	54 (20 of 37)	55 (22 of 40)	55 (42 of 77)
Total	66 (258 of 392)	54 (151 of 280)	61 (409 of 672)

code questions when it is used.

5. The frequency of elimination is much higher for the skeleton code questions (where it is the most common annotation) than for the fixed code questions. Its effectiveness for skeleton code questions is slightly better than tracing, and much better than no annotations.
6. Skeleton-code questions are much more likely than fixed-code questions not to be annotated at all.

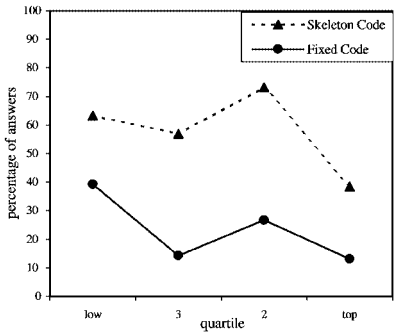
5. Annotations and Overall Performance

In the working group paper (Lister *et al.*, 2004), the authors were able to extract differences between questions by observing what students in different quartiles (based on test score) chose in each question as their answer. We applied similar techniques here, with the expectation that students who do well on the exam would be more likely to use the effective annotation techniques. Results from three annotation types can be found in Fig. 2.



(a) Tracing a single variable

(b) Synchronized tracing



(c) Blank

Fig. 2. Percentage of an annotation type used in different quartiles.

We found that tracing individual variables (T) was used about the same for the top three quartiles for fixed-code questions, but for skeleton-code questions, the top quartile was twice as likely to trace as the second and third. The increased likelihood for the top quartile to use synchronized traces (S) relative to the lower quartiles was even greater, although the overall use was lower than tracing for the top quartile – possibly because not all questions required multiple variables to be traced. The data related to blank questions are less easily explained, however, as the frequency of blank questions does not increase monotonically as we go from the top to the bottom quartile – the second quartile students have relatively more blanks than the third quartile for fixed-code questions, and more than the third or fourth for skeleton-code questions.

6. Annotations and Institutions

While the popularity of annotations varied substantially from institution to institution, they led to higher scores in almost every institution. Table 5 shows the frequency of annotations for the 12 institutions (identified by letter due to confidentiality requirements).

These are large differences. Overall, the percentage of questions with annotations varies from 28 to 92. The percentages of fixed-code questions with annotations range from 36 to 100, and for skeleton-code questions, from 10 to 93.

To try to isolate the performance effects of annotation, we examined the performance difference between "strategic" annotations (tracing or elimination) and no annotations for each institution. (Tracing and eliminations were pooled since the numbers of observations at each institution are rather low). These data are given in Table 6. This table further supports the inference that students who annotate their exams tend to perform better: four of the five highest averages are from institutions in the top five in annotation frequency. In addition, in ten of the twelve institutions, annotated questions were more often correct than "blank" questions, which would indicate that the positive effect of tracing is fairly universal. (It should be noted that the number of blank questions is extremely low for institutions T, N, C, and P (9, 4, 4, and 3 respectively), so the comparisons for those institutions are of dubious value.) An intriguing anomaly is observed with institutions E and J, however: these two had the lowest annotation frequencies, but average-to-above-average scores.

Table 5

Percentage of questions with any annotations, by institution and question type

Question type	Institution											
	E	J	L	S	Q	H	O	A	T	N	C	P
all	28	36	50	51	58	60	69	70	87	89	89	92
fixed code	36	55	62	71	71	89	81	89	98	86	100	100
skeleton code	17	10	33	23	40	20	53	43	73	93	73	80

Table 6

Percentage of questions correctly answered by institution, for Some tracing or Elimination (labelled S.T. or E) and Blank. Average score is based on all questions at the institution

	Institution											
	E	J	L	S	Q	H	O	A	T	N	C	P
S.T. or E.	87	77	87	54	72	59	63	76	78	63	76	58
Blank	62	48	33	66	53	21	45	38	67	100	50	0
Average score	68	57	58	60	61	42	56	63	78	67	75	39

7. Discussion: Making Sense of These Results

Some of the results make obvious sense: for example, tracing through the code on paper helps; the proportion of correct answers where there is tracing is much higher than where there is not. One result seem counter intuitive at first glance: students use tracing less on the harder questions, where such strategies might be expected to be effective. We can offer three possible explanations:

Too much work Answering a skeleton-code question by tracing through the different alternatives would mean four or five times the work compared to a fixed-code question. Many students seem to realize this and instead chose to change their problem solving strategy: they begin by reading the question and forming a hypothesis of how the program should work, then look for boundary values that can be used to rule out different alternatives, and finally they check whether the selected alternative seem to work. This strategy would also explain the increase in eliminations for skeleton-code questions as shown in Table 3.

Too abstract The fixed-code questions are simple in the respect that they only require a knowledge of the syntax/semantics of the language and the ability to carefully trace the values of different variables. The skeleton-code questions are different in that they give a textual description of the problem, which the students have to translate into some problem understanding, they then have to form a hypothesis of a possible solution based on the alternatives, and then evaluate the different alternatives to find the correct one possible by creating one or more test cases (compare this to the software comprehension model described in (Boehm-Davis, 1988)).

Lack of representation Closely related to the explanation above is the lack of representation. As described above the student is required to *understand* the problem description for skeleton-code questions. This might be difficult to do without representing the task at the abstract level, a skill that most novices apparently do not have. It is observed in (Bransford *et al.*, 1999) that novices tend to solve problems concretely (plugging values into equations, e.g.), while experts tend to apply the (correct) abstract principles – it may be that novices cannot represent the abstract version of the task.

From the available data we can not determine which, if any, of these are true, but answering this is an interesting possible research topic.

Regarding the different amounts of doodling at the different institutions, there are a number of possible causes – local culture, the way programming is taught, the way the test was administered, chance, and so forth. It may simply be that the subject pools were fairly homogeneous within institutions.

8. Related Work

Davies (1993) found that novices and experts have different ways of annotating programs, and also that experts spend more time annotating than novices. Both Davies' and our results indicate that annotations are a successful strategy for finding the correct answer.

Thomas *et al.* (2004) found that students who drew object diagrams performed better on tests involving object references, but their attempts to encourage greater use of diagrams were largely ineffective. Indeed, even after the benefits of using diagrams were demonstrated to students, they failed to use them when taking exams.

Hegarty (2004) looks at relations between externally produced diagrams and internal visualizations. One of the relations that she proposes is the use of external visualizations to augment internal visualizations: some of the internal processing and memory is "offloaded" to an external representation. Experimental evidence in (Hegarty and Steinhoff, 1997) showed that "low-spatial" subjects who made annotations on an external diagram performed as well as "high-spatial" subjects when doing problems involving the inference of mechanical component motion, which suggests that the use of external annotation can substitute for keeping track of details internally.

Perkins *et al.* (1989) identify tracing (which they refer to as "close tracking") as a fundamental skill required by programmers: even for novices, it can be used to avoid, diagnose, and repair bugs in programs. They also observed that students fail to trace their code effectively, and identify a number of reasons for such failure:

1) students do not understand that tracing can be useful, or are not confident that they can trace effectivly,
2) students do not understand the programming language primitives,
3) students "project intentions" onto the code, and reason from these rather than from the code itself, and
4) students have differing cognitive styles.

We explicitly observed the third reason, students "recognizing" what a code fragment does and reasoning from that level, and suspect that the other three also occurred in this study.

More generally, many previous relevant studies, specifically regarding novice programmers, use of strategies, and distinguishing comprehension and generation, are cited in a survey by Robins *et al.* (2003).

9. Conclusions

This project used both fixed-code and skeleton-code questions to test concepts and identify misconceptions in introductory programming students. Performance in both improves when students annotate their tests, particularly by tracing on paper. There are differences, however: fixed-code questions require only understanding of how each expression works. Skeleton-code questions are closer to writing code, and can require more abstract reasoning. As they require code to meet a specification, they require students to reason about aggregate function, determine proper test cases, and so forth. Other strategies, such as process-of-elimination, may be more appropriate here, as the space of four or five different pieces of code and a number of possible test cases may make tracing too tedious.

In addition to the questions as to why students annotate harder questions less, this work suggests some areas for further study:

Novice/expert annotation. Do the annotation patterns used by individuals change over time? Some previous work (Davies, 1993) suggests both the number and the type of annotations differs between novices and experts. Can similar differences be seen between first and last year students and what would these differences mean?

Institutional differences. Are institutional differences as large as these data (Table 5) would indicate? Do these differences reflect differences in how programming is being taught, or culture, or simply differences in the way the data were collected?

MCQs in practice. The results in the paper shows a clear difference between fixed-code and skeleton-code questions which makes them appropriate for different stages in a first programming course. How might we best exploit these differences to improve the student learning experience?

Acknowledgments

Thanks to all of the working group participants (see (Lister *et al.*, 2004)), and the local arrangements people at Leeds who provided a great work space and plenty of tea and coffee. Thanks to the organizers, reviewers and participants of *Kolin Kolistelut*: the organizers who provided an intellectually stimulating environment, and the reviewers and participants who asked good questions and offered good ideas on how to improve this paper and build on these results. Finally, thanks to Sally Fincher, Marian Petre, and Josh Tenenberg, whose Bootstrapping and Scaffolding workshops (supported by NSF grants DUE-0122560 and DUE-0243242) had a large positive influence on this work.

Appendix A. The Fixed-Code Questions

1. Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 3;
int i = 0;
int sum = 0;

while ( (sum<limit) && (i<x.length) ) {
     ++i;
     sum += x[i];
     }
```

   What value is in the variable "i" after this code is executed?

   a) 0
   b) 1
   c) 2
   d) 3

2. Consider the following code fragment.

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;

while ((i1 > 0 ) && (i2 > 0 )) {
     if ( x1[i1] == x2[i2] ) {
        ++count;
        --i1;
        --i2;
        }
     else if (x1[i1] < x2[i2]) {
        --i2;
        }
     else {
        // x1[i1] > x2[i2]
        --i1;
        }
     }
```

   After the above while loop finishes, "count" contains what value?

   a) 3
   b) 2
   c) 1
   d) 0

3. Consider the following code fragment:

```
int [] x = {1, 2, 3, 3, 3};
boolean b[] = new boolean [x.length];

for ( int i = 0; i < b.length; ++i )
    b[i] = false;
```

```
for ( int i = 0; i < x.length; ++i )
    b[ x[i] ] = true;

int count = 0;
for (int i = 0; i < b.length; ++i ) {
   if ( b[i] == true ) ++count;
   }
```

After this code is executed , “count” contains:

a) 1
b) 2
c) 3
d) 4
e) 5

4. Consider the following code fragment.

```
int[ ] x1 = {0, 1, 2, 3};
int[ ] x2 = {1, 2, 2, 3};
int i1 = 0;
int i2 = 0;
int count = 0;

while ((i1 < x1.length) && (i2 < x2.length )) {
    if ( x1[i1] == x2[i2] ) {
        ++count;
        ++i2;
        }
    else if (x1[i1] < x2[i2]) {
        ++i1;
        }
    else {
        // x1[i1] > x2[i2]
         ++i2;
        }
    }
```

After this code is executed, “count” contains:

a) 0
b) 1
c) 2
d) 3
e) 4

5. Consider the following code fragment.

```
int[ ] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;

while (i < j) {
    temp = x[i];
    x[i] = x[j];
    x[j] = 2*temp;
    i++;
    j--;
    }
```

After this code is executed , array "x" contains the values:

a) {3, 2, 2, 0}
b) {0, 1, 2, 3}
c) {3, 2, 1, 0}
d) {0, 2, 4, 6}
e) {6, 4, 2, 0}

7. Consider the following code fragment:

```
int[] x = {2, 1, 4, 5, 7};
int limit = 7;
int i = 0;
int sum = 0;

while ( (sum<limit) && (i<x.length) ) {
    sum += x[i];
    ++i;
    }
```

What value is in the variable "i" after this code is executed?

a) 0
b) 1
c) 2
d) 3
e) 4

10. Consider the following code fragment.

```
int[] array1 = {2, 4, 1, 3};
int[] array2 = {0, 0, 0, 0};
int a2 = 0;
for ( int a1=1 ; a1<array1.length ; ++a1 ) {
    if ( array1[a1] >= 2 ) {
        array2[a2] = array1[a1];
        ++a2;
        }
    }
```

After this code is executed , the array "array2" contains what values?

a) {4, 3, 0, 0}
b) {4, 1, 3, 0}
c) {2, 4, 3, 0}
d) {2, 4, 1, 3}

Appendix B. The Skeleton-Code Questions

6. The following method "isSorted" should return true if the array is sorted in ascending order. Otherwise, the method should return false:

```
public static boolean isSorted (int []x) {
//missing code goes here
}
```

Which of the following is the missing code from the method "isSorted" ?

a)
```
boolean b = true;
for (int i=0 ; i<x.length-1 ; i++) {
```

```
            if ( x[i ] > x[i+1] )
                b = false;
            else b = true;
            }
        return b;

b)      for ( int i=0 ; i<x.length-1 ; i++ ) {
            if (x[i ] > x[i+1] ) return false;
            }
        return true;

c)      boolean b = false;
        for (int i=0 ; i<x.length-1 ; i++) {
            if (x[i] > x[i+1] ) b = false;
            }
        return b;

d)      boolean b = false;
        for (int i=0;i<x.length-1;i++) {
            if (x[i ] > x[i+1] ) b = true;
            }
        return b;

e)      for (int i=0;i<x.length-1;i++) {
            if (x[i ] > x[i+1] ) return true;
            }
        return false;
```

8. If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array “x” that contains the following six numbers:

   4 5 6 2 1 3

   There are 10 inversions in that array, as:

```
x[0]=4   >   x[3]=2
x[0]=4   >   x[4]=1
x[0]=4   >   x[5]=3
x[1]=5   >   x[3]=2
x[1]=5   >   x[4]=1
x[1]=5   >   x[5]=3
x[2]=6   >   x[3]=2
x[2]=6   >   x[4]=1
x[2]=6   >   x[5]=3
x[3]=2   >   x[4]=1
```

   The skeleton code below is intended to count the number of inversions in an array “x”:

```
     int inversionCount = 0;
     for  ( int i=0 ; i<x.length-1 ; i++ ) {
         for  xxxxxx  {
              if (  x[i] > x[j] ) ++inversionCount;
             }
         }
```

When the above code finishes, the variable "inversionCount" is intended to contain the number of inversions in array "x". Therefore, the "xxxxxx" in the above code should be replaced by:

```
a)   ( int j=0   ; j<x.length   ; j++ )
b)   ( int j=0   ; j<x.length-1 ; j++ )
c)   ( int j=i+1 ; j<x.length   ; j++ )
d)   ( int j=i+1 ; j<x.length-1 ; j++ )
```

9. The skeleton code below is intended to copy into an array of integers called "array2" any numbers in another integer array "array1" that are even numbers. For example, if "array1" contained the numbers:
   array1: 4 5 6 2 1 3
   then after the copying process, "array2" should contain in its first three places:
   array2: 4 6 2
   The following code assumes that "array2" is big enough to hold all the even numbers from "array1":

```
int a2 = 0;
for ( int a1=0 ; xxx1xxx ; ++a1 ) {
    // if array1[a1] is even
    if ( array1[a1] % 2 == 0 ) {
        // array1[a1] is even, so copy it
        xxx2xxx;
        xxx3xxx;
        }
    }
```

   The missing pieces of code "xxx1xxx", "xxx2xxx" and "xxx3xxx" in the above code should be replaced respectively by:

```
a)    a1<array1.length
      ++a2
      array2[a2] = array1[a1]

b)    a1<array1.length
      array2[a2] = array1[a1]
      ++a2

c)    a1<=array1.length
      array2[a2] = array1[a1]
      ++a2

d)    a1<=array1.length
      ++a2
      array2[a2] = array1[a1]
```

   Hint: in all four options above, the second and third parts are the same, just reversed.

11. Suppose an array of integers "s" contains zero or more different positive integers, in ascending order, followed by a zero. For example:
    `int[] s = {2, 4, 6, 8, 0};`
    or `int[] s = {0};`
    Consider the following "skeleton" code, where the sequences of "xxxxxx" are sub-

stitutes for the correct Java code:

```
int pos = 0;
while ( (xxxxxx) && (xxxxxx) ) ++pos;
```

Suppose an integer variable “e” contains a positive integer. The purpose of the above code is to find the place in “s” occupied by the value stored in “e”. Formally, when the above “while” loop terminates, the variable “pos” is determined as follows:

1. If the value stored in “e” is also stored in the array, then “pos” contains the index of that position. For example, if e=6 and s = {2, 4, 6, 8, 0}, then pos should equal 2.
2. If the value stored in “e” is NOT stored in the array, but the value in “e” is less than some of the values in the array then “pos” contains the index of the lowest position in the array where the value is larger than in “e”. For example, if e=7 and s = {2, 4, 6, 8, 0}, then pos should equal 3.
3. If the value stored in “e” is larger than any value in “s”, then “pos” contains the index of the position containing the zero. For example, if e=9 and s = {2, 4, 6, 8, 0}, then pos should equal 4.

The correct Boolean condition for the above “while” loop is:

```
a)    (pos < e)     &&  (s[pos] != 0)
b)    (pos != e)    &&  (s[pos] != 0)
c)    (s[pos] < e)  &&  (  pos  != 0)
d)    (s[pos] < e)  &&  (s[pos] != 0)
e)    (s[pos] != e) &&  (s[pos] != 0)
```

12. This question continues on from the previous question. Assuming we have found the position in the array “s” containing the same value stored in the variable “e”, we now wish to write code that deletes that number from the array, but retains the ascending order of all remaining integers in the array. For example, given:

```
s = {2, 4, 6, 8, 0};
e = 6;
pos = 2;
```

The desired outcome is to remove the 6 from “s” to give:

```
s = {2, 4, 8, 0, 0};
```

Consider the following “skeleton” code, where “xxxxxx” is a substitute for the correct Java code:

```
do {
    ++pos;
    xxxxxx;
    } while (s[pos] != 0 );
```

The correct replacement for “xxxxxx” is:

```
a)    s[pos+1] = s[pos];
b)    s[pos]   = s[pos+1];
c)    s[pos]   = s[pos-1];
d)    s[pos-1] = s[pos];
e)    None of the above
```

References

Boehm-Davis, D.A. (1988). *Handbook of Human-Computer Interaction*, Ch. 5 (Software Comprehension). Elsevier.

Bransford, J.D., A.L. Brown, R.R. Cocking (Eds.) (1999). *How People Learn*: *Brain, Mind, Experience, and School*. National Academy Press, Washington, D.C.

Davies, S. (1993). Externalising information during coding activities: effects of expertise, environment, and task. In *Empirical Studies of Programmers*: *5th Workshop*. Ablex, Norwood, NJ, pp. 42–61.

Hegarty, M. (2004). Diagrams in the mind and in the world: relations between internal and external visualizations. In: A. Blackwell, K. Marriot, A. Shimojima (Eds.), *Diagrams 2004*, *LNAI*, **2980**. Springer-Verlag, Berlin Heidelberg, pp. 1–13.

Hegarty, M., K. Steinhoff (1997). Use of diagrams as external memory in a mechanical reasoning task. *Learning and Individual Differences*, **9**, 19–42.

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

Perkins, D.N., C. Hancock, R. Hobbs, F. Martin, R. Simmons (1989). Conditions of learning in novice programmers. In E. Soloway, J.C. Spohrer (Eds.), *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 261–279.

Robins, A., J. Rountree, and N. Rountree (2003). Learning and teaching programming: a review and discussion. *Computer Science Education*, **13**(2), 137–172.

Thomas, L., M. Ratcliffe, and B. Thomasson (2004). Scaffolding with object diagrams in first year programming classes: some unexpected results. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. Norfolk, USA, pp. 250–254.

R. McCartney is an associate professor in the Department of Computer Science and Engineering, University of Connecticut, USA. His research interests include computer science education, diagrammatic reasoning, and cooperative robotics.

J.E. Moström is currently a lecturer at the Department of Computing Science, Umeå University, Sweden. He has more than 15 years of experience teaching programming, human computer interaction, realtime systems, operating systems, etc. His current research interests include computer science education reseach, psychology of programmers and human computer interaction.

K. Sanders is an associate professor in the Math and Computer Science Department at Rhode Island College, USA. Her research interests include automated legal reasoning and computer science education.

O. Seppälä is a researcher and a DSc student at Helsinki University of Technology. His research interest lies with the use of automatic programming visualization and debugging tools in CS education.

Pradedančiųjų programuotojų komentarų apie testavimą efektyvumas

Robert MCCARTNEY, Jan Erik MOSTRÖM, Kate SANDERS, Otto SEPPÄLÄ

Straipsnyje nagrinėjami dvylikos institucijų pradedančiųjų programuotojų atsakymų į pasirenkamojo pobūdžio testus suvestiniai rezultatai. Ypatingas dėmesys kreipiamas į studentų pateikiamus komentarus apie testus. Iš to daromos išvados, jog pats klausimo pobūdis turi įtakos ir studentų darbams, ir jų pateikiamiems komentarams. Suskirstydami studentų atsakymus pagal klausimų pobūdį, komentarų tipą (trasavimas, eliminavimas, kita arba be komentarų) bei jų atstovaujamas institucijas, išsiaiškinome, jog trasavimo metodas laikomas efektyviausiu pateikiant vieno tipo klausimus, o eliminavimas ûkito tipo klausimus, tačiau bendru atveju bet koks komentaras kur kas geriau negu jokio.