On the Differences Between Correct Student Solutions

Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, Se-Young Yu
Dept. of Computer Science
The University of Auckland
Auckland, New Zealand
{andrew,paul,diana,ewan,seyoung}@cs.auckland.ac.nz

ABSTRACT

We know that students solve problems in different ways, but we know little about the kinds of variation, or the degree of variation between these student generated solutions. In this paper, we propose a taxonomy that classifies the variation between correct student solutions in objective terms, and we show how the application of the taxonomy provides instructors with additional insight about the differences between student solutions. This taxonomy may be used to inform instructors in selecting examples of code for teaching purposes, and provides the possibility of automatically applying the taxonomy to existing solution sets.

Categories and Subject Descriptors

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

General Terms

Design, Human Factors

Keywords

variation, taxonomy, novice, programming, syntax

1. INTRODUCTION

Examples are a critically important part of learning to program. Almost all instructors of introductory programming courses use examples to illustrate the use of syntax to solve simple programming problems, and numerous textbooks emphasise the importance of examples in the learning process (for example, see [16]).

One common use of examples is to illustrate the different approaches (and/or programming styles) that may be used to solve a given problem. This process of asking students to consider the differences between solutions is characteristic of pedagogies such as peer review [15], and is underpinned by variation theory [10, 14] and social cognitive theory [3].

Advocates of pedagogies that involve reviewing of peer generated code claim numerous benefits for students, including the development of self-assessment skills, critical thinking, communication skills and content knowledge [13, 7]. We believe that exposing students to a variety of solutions is particularly important in programming, since students must learn to read and understand code produced by others if they are to be successful.

However, if there are many solutions, it is not clear *which* of them to expose to students. Ideally we would like to identify a small number of solutions that usefully convey the variation that is possible while still being correct. But to do so, we need a good understanding of what variety is possible. Unfortunately, there are few reports that focus on the nature of the variation between different successful student-generated solutions. Studies conducted in this area have tended to focus on variation in high-level concepts, such as novice programmers' conceptions of "object" and "class" [5], rather than characterising the variation in syntax and semantics present in different student solutions. A systematic way of classifying solutions that captured their variation would help identify interesting solutions to use as examples.

Categorizing correct student solutions may provide a number of additional benefits to teachers, students and developers of automated learning systems. It may provide instructors with some insight into the programming choices made by their students. This opportunity to gain feedback about how students are solving problems is invaluable, since "the most powerful feedback occurs when feedback is to the instructor: about how well they have taught, who they taught well, what they have or have not taught well" [6]. Students may be provided with objective feedback about how their solution relates to other solutions submitted by their peers. Automated learning systems such as those involving peer review [9] could use this categorisation to determine which learning resources should be distributed to a student to ensure exposure to a variety of solution types.

Numerous previous studies have investigated how students solve simple programming problems [12, 11]. Taxonomies such as SOLO have been used to categorize the degree to which solutions are correct, in order to tease out the different cognitive skills involved in programming [8], but these studies have focused on the correctness of code rather than trying to capture the differences between correct solutions.

In this paper, we examine a set of successful student solutions to simple programming problems and propose a taxonomy that classifies the variations between the code used in those solutions (i.e. the relationships between different solutions to the same problem). Additionally, we apply the top level of this taxonomy to categorise a set of solutions submitted by novice programmers.

We investigate only "correct" solutions that have passed a set of test cases defined for each problem. That is, for

ITiCSE'13, July 1–3, 2013, Canterbury, England, UK.

each problem we have a set of solutions that are functionally equivalent with respect to the test cases. Each student solution consists of a single method, so the taxonomy proposed here is intended to categorize only variation of code within methods and does not necessarily capture variation in the design of larger systems.

The major contributions of this paper are:

- describing a taxonomy that captures the variation between correct student solutions, and
- showing the potential benefits of automatically applying the top level of the taxonomy to a real data set.

2. METHODOLOGY

Code can vary in many ways, some possibly important to helping understand what variation is interesting, some probably not. Coming up with a systematic classification of code variation that can be objectively applied and is simple enough to be automated has been a challenge.

We began by examining the solutions to a set of short programming exercises in Java developed by students in a second programming course. Each exercise required students to complete a method that fulfilled the problem specification. The methods were tested for correctness using an automated test suite and students were given feedback on the test cases they passed or failed. Students were able to keep submitting until their code passed all the tests.

While we expected to see variation, we were surprised by the amount of variation we saw to one of the simplest problem specifications, shown in Figure 1.

> *Write a method that will take two ints as parameters and return their sum. The method header has been provided below:*
>
> `int sumTwoInts(int a, int b)`

Figure 1: The SumTwoInts method.

Even with this very simple exercise, we observed variations of code that involved use of parentheses, use of white space, variable declaration and initialization, simple variable declaration, choice of identifier names, and use of assignment statements. Figure 2 illustrates some of the observed variation among student solutions.

```
return a + b;
```

```
return (a + b);
```

```
return a+b;
```

```
return (a+b);
```

```
int result = a + b;
return result;
```

```
int result = (a + b);
return result;
```

```
int c;
c = a + b;
return c;
```

```
int sum;
sum = (a + b);
return sum;
```

```
int result;
return (result = a + b);
```

Figure 2: Examples of student solutions to a simple programming exercise.

To formally analyse the solutions, we used a process known as *thematic analysis* [4] to identify patterns in the data and organize those patterns into themes. Braun and Clarke [4] describe six phases of thematic analysis as follows:

1. Familiarizing yourself with the data
2. Generating initial codes
3. Searching for themes
4. Reviewing themes
5. Defining and naming themes
6. Producing the report

Initially, we read through the set of student solutions, making notes of any initial observations and discussing particular examples in detail. After familiarization with the data, one of the researchers generated codes that described various features of the solutions. The process of coding started with the first student solution, and progressed by adding codes as new features were identified in subsequent solutions. The codes were distributed to other researchers and checked against a sample of student solutions for content validity. After the entire set of codes was generated, the data was rechecked for consistency and related codes were grouped together to identify a set of themes. These themes were refined and described in terms of objective compiler concepts to reduce the level of ambiguity between categories.

3. REVIEW OF COMPILER CONCEPTS

We found that the themes we identified were most clearly defined in terms of some standard compiler concepts, which we now review. The first step in the compilation process is typically to represent the source code as a sequence of *tokens*. A token is an atomic unit of the language, for example the operator `>=` or the identifier `cokesPurchased`. The token stream will usually remove all information regarding formatting (use of whitespace, layout, and so on).

The next step is to parse the token stream according to the language grammar. The grammar generally refers to token *classes* rather than individual tokens, so that it will treat `cokesPurchased` and `count` as both members of the token class IDENTIFIER.

Some compilers operations (such as optimisation and other forms of analysis) represent the code as a *control flow graph*. This is a graph in which each vertex is a basic block, and edges between vertices indicate control flow. A basic block is a sequence of statements with a single entry point at the beginning (i.e. there are no jumps to anything other than the first statement), and a single exit point (no possibility of control to leave other than the last statement). More details are available in any standard text on compilers, such as Aho et al.[1].

4. TAXONOMY

We characterise the variation observed in the data as belonging to three distinct themes: *Structure, Syntax* and *Presentation*. In this taxonomy, the themes are hierarchical in nature. Structural variation is the highest level of variation, followed by Syntax, with Presentation being the lowest level. If two solutions have the same structure, then we may consider how the syntax varies between the two solutions. If two solutions have the same structure and the same syntax, then it is possible to consider how they might differ in presentation.

For each theme, we illustrate the variation with examples of solutions to an exercise in which students were asked to calculate the number of free Coke cans awarded for a given purchase during a promotion. The rules of the promotion state that for every 20 cans purchased, 3 are given away for free. The method signature provided to students when solving this exercise is shown below:

```
int freeCoke(int cokesPurchased)
```

4.1 Variation of structure

We define the structure of the code as being expressed by the control flow graph of the code. Solutions to a given problem that have different control flow graphs illustrate structural variation.

An example of this structural variation is shown in Figures 3(a) and 3(b). In Figure 3(a), the student has calculated and returned the result in a single expression. In Figure 3(b), the student has used a conditional statement to handle the case when the input value is less than 20. Although this is not strictly necessary as the integer division would evaluate to 0 when the input value is less than 20, this does result in a different structure to the solution in Figure 3(a).

```
return (cokesPurchased/20)*3;
```

(a) A simple solution to the `freeCoke` problem

```
if (cokesPurchased < 20) {
    return 0;
} else {
    int coke = cokesPurchased/20;
    return coke*3;
}
```

(b) A more complex solution to `freeCoke` problem that uses a conditional statement

Figure 3: Solutions (a) and (b) illustrate structural variation because they have different control flow graphs.

4.2 Variation of syntax (within blocks)

The second theme is the variation of code that occurs within basic blocks (i.e. code with a single entry point, single exit point, and no internal branching). We use the name *Syntax* to characterize these sequences of tokens that occur within basic blocks. We acknowledge that the term carries other connotations, but we believe that the proposed taxonomy classifies variation between solutions at the appropriate levels, and the term Syntax is adequate for characterising the variation within blocks. In this taxonomy we only classify the variation of syntax between solutions that have the same structure. As Figure 2 illustrates, even simple problems have numerous solutions with the same structure but different code within the structure.

The code fragments shown in both Figures 4(a) and 4(b) are examples of basic blocks, that is, series of sequential statements without any branches. Although both blocks exhibit the same structure, the approach taken in calculating the result is different in each case. In Figure 4(a), the result is calculated and returned in the same expression without the use of separate variables. In Figure 4(b), two separate variables are declared to store intermediate results, and the return value is a single variable.

4.3 Variation of presentation

The third and final theme captures the variety of ways that the code is presented. Two solutions that vary only in presentation will have the same sequence of token classes, even though the token values may vary. In other words, presentation is about the use of whitespace, identifier names and other superficial elements of style.

The code fragments shown in Figures 5(a) and 5(b) are both basic blocks, and both feature the same token classes

```
return (cokesPurchased/20)*3;
```

(a) A simple solution to the `freeCoke` problem

```
int temp = cokesPurchased/20;
int numOfFreeCoke = temp*3;
return numOfFreeCoke;
```

(b) A solution that uses intermediary variables to calculate the result

Figure 4: Solutions (a) and (b) illustrate variation of syntax because they have the same structure, but different statements within basic blocks.

in the same order. The only variation in these solutions are the identifier names and the spacing used (a blank line is included in Figure 5(b)).

```
int temp = cokesPurchased/20;
int numOfFreeCoke = temp*3;
return numOfFreeCoke;
```

(a) A solution that uses informative variable names

```
int f = cokesPurchased/20;
int free = f*3;

return free;
```

(b) A solution that uses less informative variable names, and contains a blank line

Figure 5: Solutions (a) and (b) illustrate variation in presentation because they have the same syntax, but the identifiers and code layout vary.

5. EVALUATION

In order to evaluate the feasibility of applying the proposed taxonomy, we developed an Eclipse plugin to automatically categorise solutions at the structural level. The plugin generates a control flow graph for a target method based on the open-source project *Control Flow Graph Generator* [2].

A collection of Java source files is accepted as input to be analysed using this plugin. The plugin tool compares the control flow graph of a given method within a Java source file with each graph already generated. If a match is obtained, then the analysed code joins the existing set with the identical structure. However, if the control flow graph doesn't match any existing graph, then it becomes the first entry of a new category. Once all solutions have been analysed, the plugin produces a summary of the categories, and the source files that belong to each category.

We used this tool to analyse a new data set of solutions. Students in a typical CS1 course were asked to produce solutions to 10 simple exercises. The first 5 exercises were designed to be straightforward, and may be solved without the use of a loop. The second set of 5 exercises were more challenging, with 4 of the 5 exercises (all except for "magicNumbers") requiring a loop to solve. Students were only eligible to attempt the second set of 5 exercises if they

successfully completed (i.e. submitted a correct solution to) all 5 of the first set of exercises. Table 1 lists all 10 exercises.

We generated the control flow graphs for each of the student solutions to a given problem, and grouped the solutions according to these graphs. That is, all the student solutions that had the same structure were grouped together into a single category.

6. RESULTS

Table 1 shows the results of classifying the student solutions to the 10 exercises according to the structure the solution has. The exercises are ordered in increasing difficulty as determined by the instructor (Paul Denny). The N_p column gives the total number of passing student solutions that have been classified for the corresponding exercise. The N_c column gives the number of different categories into which the passing solutions for the exercise have been classified. The next 5 columns show the number of student solutions in the 5 most populous categories for the given exercise. Note that the most popular category for one exercise is not necessarily the most popular category for any other.

Over all passing solutions to the 10 exercises, there were 278 different categories. Of these 166 contained only one solution. The most popular category (consisting of a single block followed by a return statement) had 189 solutions. The 5 most popular categories had 186, 106, 106, 102, and 76 solutions respectively.

Name	N_p	N_c	5 Most Popular Cat.				
replaceCharAtPos	186	19	134	33	2	2	1
productIsEven	208	16	55	43	32	28	25
swapEnds	183	58	19	16	14	11	11
containersNeeded	188	53	27	17	16	15	13
weeklyPay	179	58	18	12	11	11	10
countOdds	140	17	98	9	9	5	3
posOfValInArray	135	34	57	10	10	8	5
reverse	132	6	101	17	8	4	1
magicNumbers	138	53	30	16	9	7	5
sumValues	133	56	27	19	11	7	6

Table 1: Analysis of the variation in structure of correct student solutions to 10 short programming exercises.

The underlined numbers in Table 1 show the category that the instructor's solution was classified as. For "weeklyPay", the instructor's solution was in the 17th most popular category (with 5 student solutions). For "sumValues", the instructor's solution was in the 8th most popular (3 students). For "magicNumbers", the instructors solution was in one of the 29 categories that had only one student solution.

Figure 6 shows the distribution across all categories (not just the 5 most popular) for 3 exercises. The values plotted are the *proportion* of student solutions in each category. To show separation between the three curves, the largest value for "reverse" (0.77) is omitted. The exercises chosen are shown in bold type in Table 1.

7. DISCUSSION

Considerable differences in the distribution of category sizes exist between exercises. The "reverse" exercise stands out as having solutions with very few structural variations. This exercise required students to return an array of integers in which the elements were in the reverse order to the

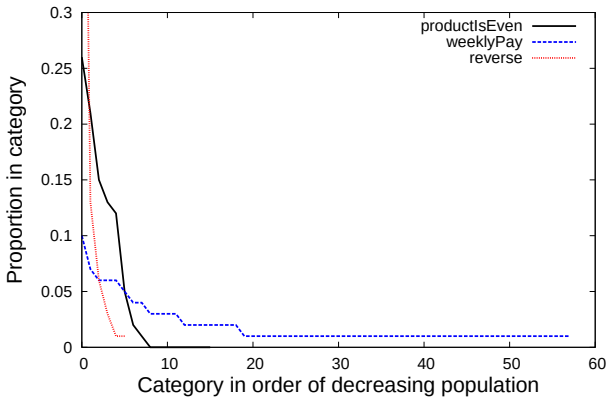

Figure 6: Number of different structural categories for the *product, weekly* and *reverse* exercises

original input array. It is remarkable that there were only 6 variations of structure for this exercise, when some simpler exercises exhibited more than 50 variations. The structure of the most common solution to this exercise consisted of a basic block (to create the new array), followed by a for loop (to assign elements to the new array), followed by a single statement to return the result. The next most common structure category, with 17 entries, did not include the initialization block as it reversed the elements in place by swapping pairs of elements working from the outer indices towards the middle of the array, and then returning a reference to this modified array. Students were not required to preserve the order of the elements in the input array, so this is a perfectly valid solution to the problem.

At the other extreme, the "weeklyPay" exercise required students to calculate the weekly pay of an employee with an age-based pay-scale. The inputs to the exercise were the number of normal hours worked (normalHrs), the number of overtime hours worked (overtimeHrs) and the age (age) of the worker. The actual wording of the question was as follows:

> *The base pay rate for all workers is $15 per hour. On top of the base rate, each worker over the age of 20 earns an extra $1 per hour for every year their age exceeds 20. So, for example, a worker who is 25 years old, will receive a base pay rate of 15 + 5 = $20 per hour. However this additional, age-based bonus is only valid up until the age of 40. So, for example, a 40 and a 45 year old will earn the same base rate. Finally, any overtime hours are paid at twice the base rate.*

The instructor's solution to this exercise, shown in Figure 7, had a very simple structure consisting of just a simple block followed by a return statement. Of the 179 correct solutions submitted to this exercise there were 58 distinct structure variations. One student solution, which was in a category of its own, is shown in Figure 8.

The tool used to collect the data for our study (CodeWrite [5]) included a feature where students were able to view all other successful submissions to an exercise once they had made a successful submission of their own. In large classes, each exercise may have many successful submissions associated with it, making it impractical to expect a student to

```
int base = 15 + Math.min(20, Math.max(0, age-20));
return normalHrs * base + 2 * overtimeHrs * base;
```

Figure 7: The instructor's solution to the "weeklyPay" exercise

```
int money = 0;
int basePay = 15;

if (age <= 20){
    int normalPay = basePay * normalHrs;
    int overtimePay = (2 * basePay) * overtimeHrs;
    money = normalPay + overtimePay;
} else if (age > 20 && age <= 40){
    int olderBasePay = (age-20) + basePay;
    int normalPay = olderBasePay * normalHrs;
    int overtimePay = (olderBasePay * 2) *
        overtimeHrs;
    money = normalPay + overtimePay;
} else if (age > 40){
    int normalPay = (basePay + 20) * normalHrs;
    int overtimePay = (2 * (basePay +20)) *
        overtimeHrs;
    money = normalPay + overtimePay;
}
return money;
```

Figure 8: One student's solution to the "weeklyPay" exercise

view them all. One potential application of the taxonomy presented here would be as a filter so that a given student would not be shown two different solutions if there was no variation between those solutions. The type of variation permitted could be determined by selecting an appropriate level of the hierarchy. Filtering at the highest level of the hierarchy, the structure level, would remove solutions that did not vary by structure. Taking an exercise such as "replaceCharAtPos" for example, this would reduce the number of solutions a student would potentially see from 246 down to 19.

Conversely, in the context of a marking activity in which the set of class submissions are divided amongst multiple markers, it might be preferable to reduce the structural variation in the code that a given marker is allocated in order to simplify the grading process.

The taxonomy may also be of use to instructors who are designing their own examples (possibly for discussion in class or as the basis for questions on a test) as a way of helping them think about the different kinds of variations that are typical in student code. As an example, consider the "sumValues()" function in which, depending on the value of a boolean input ("positivesOnly"), either all of the elements or only the positive elements in the array ("values") should be summed. The instructor solution to this exercise, given in Figure 9(a), uses a single loop and a single complex condition. Students answering the same exercise were less inclined to define complex boolean conditions. The most common structure amongst student submissions (seen in 30 of 153 submissions) involved first iterating over the array elements and, for each, using an if/else construct to handle the positivesOnly condition (see Figure 9(b)). The second most common structure (with 21/153 submissions) involved first using an if/else statement to handle the "positivesOnly" condition and then using a loop in each branch to iterate over the array elements (see Figure 9(b)). In fact, only 3 of the 153 student submissions to this exercise used the same structure as the instructor. This may suggest a useful code re-writing exercise for this cohort, as part of a test or exam, could involve rewriting code provided in one of the more common structures so that only a single condition is used.

```
int sum = 0;
for (int i = 0; i < values.length; i++) {
    if (values[i] > 0 || !positivesOnly) {
        sum += values[i];
    }
}
return sum;
```

(a) Instructor solution to "sumValues"

```
int sum = 0;
for (int i = 0; i < values.length; i++) {
    if (positivesOnly ==true) {
        if (values[i] >=0) {
            sum += values[i];
        }
    } else {
        sum += values[i];
    }
}
return sum;
```

(b) The most popular student solution, in terms of structure

```
int sum = 0;
if (positivesOnly) {
    for(int i=0;i<values.length;i++) {
        if(values[i] > 0) {
            sum+= values[i];
        }
    }
} else {
    for(int i = 0;i<values.length;i++) {
        sum+= values[i];
    }
}
return sum;
```

(c) The second most popular student solution, in terms of structure

Figure 9: The instructor's solution (a) to the "sumValues" exercise, along with the two most common structural approaches (b, c) taken by students

Finally, application of the taxonomy to a set of student solutions may assist instructors in identifying misconceptions or poor code design that often appears in student submissions that are the only example in their structural category for a given exercise. As an example, consider the three student submissions to the "countOdds" exercise (see Figure 10) in which the number of elements in an array that are odd should be calculated. Of the 153 submissions to this exercise, 40 had a unique structure and these three examples have been selected from this group. The set of solutions with unique structures for a given exercise tended to be a rich source of examples that contained redundant or unusual code that an instructor may wish to address.

8. CONCLUSIONS

The objective nature of the taxonomy makes it a feasible candidate to be automatically applied to large data sets of solutions, thereby allowing both staff and students viewing those solutions to concentrate on the differences between

```
int oddCount = 0;
for(int i = 0; i < values.length; i++) {
    if (values[i] % 2 > 0) {
        oddCount++;
    } else {};
}
return oddCount;
```

(a) Inclusion of an empty else clause and redundant semicolon

```
int numOdds = 0;
for(int i =0;i<values.length;i++){
    if(values[i]==1){
        numOdds++;
    }else if(values[i]%2!=0){
        numOdds++;
    }
}
return numOdds;
```

(b) Redundant check for the literal value "1" being present in the array

```
int k = 0;
for(int i = 0; i<values.length; i++){
    int j = 0;
    if(values[i]%2 != 0){
        k++;
    } else if(values.length == 0){
        k = 0;
    } else{};
}
return k;
```

(c) Redundant check for an empty array performed on each loop iteration

Figure 10: Student solutions to the "countOdds" exercise that were the only examples in their structural categories

them. We have shown how student solutions may be categorised at the structural level, and the potential insight it may provide to instructors.

In future, we intend to use this taxonomy to perform more detailed analysis of student-generated solutions. Having identified different categories of solutions, we plan to investigate the relationships that exist between categories of solutions and code comprehension. Although we do not currently know which kinds of variation provide the best learning opportunities for students, the taxonomy described here provides an initial framework with which such questions may be answered.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] A. Alimucaj. Eclipse control flow graph generator. Retrieved from `https://eclipsefcg.svn.sourceforge.net/svnroot/eclipsefcg/org.flowChartPlugin/` on 9/01/13.

[3] A. Bandura. *Social foundations of thought and action: A social cognitive theory*. Prentice-Hall, 1986.

[4] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006.

[5] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 471–476, New York, NY, USA, 2011. ACM.

[6] J. Hattie. The black box of tertiary assessment: An impending revolution. In L. H. Meyer, S. Davidson, H. Anderson, R. Fletcher, P. Johnston, and M. Rees, editors, *Tertiary Assessment & Higher Education Student Outcomes: Policy, Practice & Research*, pages 259–275. Ako Aotearoa, Wellington, New Zealand, 2009.

[7] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan. Integrating pedagogical code reviews into a CS 1 course: an empirical study. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pages 291–295, New York, NY, USA, 2009. ACM.

[8] R. Lister, T. Clear, Simon, D. J. Bouvier, P. Carter, A. Eckerdal, J. Jacková, M. Lopez, R. McCartney, P. Robbins, O. Seppälä, and E. Thompson. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bull.*, 41(4):156–173, Jan. 2010.

[9] A. Luxton-Reilly. A systematic review of tools that support peer assessment. *Computer Science Education*, 19(4):209–232, Dec 2009.

[10] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Associates, 1997.

[11] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33(4):125–180, Dec. 2001.

[12] E. Soloway and J. C. Spohrer. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1988.

[13] H. Sondergaard. Learning from and with peers: the different roles of student peer reviewing. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 31–35, New York, NY, USA, 2009. ACM.

[14] E. Thompson, J. Suhonen, J. Davies, and Kinshuk. Applications of variation theory in computing education. In R. Lister and Simon, editors, *Koli Calling 2007. Proceedings of the Seventh Baltic Sea Conference on Computing Education Research*, volume 88, pages 217–220. Australian Computer Society Inc, 2008.

[15] K. Topping. Peer assessment between students in colleges and universities. *Review of Educational Research*, 68(3):249–276, 1998.

[16] C. Walnum. *Java by Example*. Que, 1996.