Chapter #7

EVALUATING PROGRAMMING COMPETENCE FROM EXPLANATIONS

Edward Brown
Memorial University of Newfoundland, Canada

ABSTRACT
With a prolific amount of computer code available on the web, students are able to use the Internet as a compendium of solutions to computer programming problems. Web search is not only a problem-solving strategy with which students are familiar with and have highly developed skills from years of practice and implicit cultural knowledge, it is also an approach anticipated in real-world contexts for computer programming (Treude, Barzilay, & Storey, 2011). This chapter is an account of the author's transition from pedagogy disallowing the use of solutions copied from web pages to a pedagogy which encourages students to incorporate found solutions into their work. Instead of penalizing students for "cheating" when they adopt other programmer's solutions to computer programming problems, emphasis is instead placed on student's explanations of the solutions they provide regardless of their origin. The effectiveness of this approach is predicated on the idea that the ability to produce comprehensive explanation of a programming solution is a good indicator of programming competency. Otherwise, there is no reason to think adopting someone else's code is a valid learning activity. There is literature to support the idea that explaining and studying (sometimes characterized as reading) existing solutions and program code significantly improves students learning and development of problem-solving strategies (Corney et al, 2014). This chapter suggests similar benefits may accrue from code that is not selected as part of the curriculum, but found by the individual students. More speculative aspects of the approach are the absence of specific instruction in specific problem-solving skills, and absence of a requirement that students eventually shift to independent composition of code as a later stage of demonstrating programming competence. Emphasis is shifted away from the text of computer code solutions, towards student description and assessment of computer code solutions. Students provide their descriptions in a combination of natural language and Unified Modeling Language. Thus design and implementation is separated as advocated by Falkner, Vivian, and Falkner (2014), and the Internet no longer serves as a compendium of pre-packaged solutions. Informal observations regarding a one semester application of this approach conclude the chapter. This chapter is an extended version of Brown (2015).

Keywords: Problem-solving, Programming, Pedagogy

1. THE INTERNET SEARCH STRATEGY

Students have been practicing Internet skills for most of their lives; whereas learning a new problem-solving strategy and trying to apply it to assigned school work may be an inherently bad approach from the student's point of view: difficult, time-consuming and prone to failure. Searching the Internet for a completed answer that is a close enough fit to the problem is quicker, more efficient and likely to be more successful. Circumventing new intended or incidental learning may not be terribly relevant if the student's personal objective is effective and efficient solution to an assigned problem rather than learning or practicing new skills.

Computer programming is one discipline with a heavily problem-oriented curriculum. Programming problems are often assigned to encourage students to develop their programming skills. Curricula, at least at collegiate level, have been criticized for not providing specific problem-solving instruction, instead relying on sequencing of problems to progress the curricular topics. (Guzdial, 2015). However, if we expect students to regulate their own learning strategies, appropriate domain-specific knowledge about solving problems is a prerequisite (Winne, 1995). Veenman, Elshout, and Meijer (1997) note that novices are "restricted by a poor working method which stems from a lack of domain-specific knowledge" (p. 188) and they resort to a cycle of 'impasses and local repairs' without specific strategies to tackle conceptually difficult problems. They tend to delay problem solving until absolutely necessary, resulting in a need to rely on a general mega-strategy that can solve their entire problem in one operation. Given that students without substantial tutelage in solving programming problems are reduced to adopting a general strategy, it should not be surprising that students would retreat to using a familiar Web search strategy for solutions in preference to attempting to decompose and analyzing an unfamiliar programming problem.

In introductory computer programming, the particular strategy for producing a program is seldom assessed: instead, the quality of software submitted is the measure of student performance. An analogy in a math class might be to only assess the final answer, without considering the validity of a student's "workings" that get them to a particular answer. We might tell our students not to copy answers from the Internet, that solving the problem independently is important learning exercise, but seldom review what programming skills the student actually has acquired. No one objects if students seek alternative explanations of course topics, or review elements of a problems already discussed, yet the same students are expected to curtail the kind of investigation they undertake and assistance they accept at some vague point with respect to assigned problems. This contrasts dramatically with industrial or working environments, replete with on-line communities that provide solutions and colleague helpers. Programmers in industry are expected to use these resources (Treude et al., 2011).

There are advocates of unconstrained access to the Internet. Professor Sugata Mitra, (Mitra, 2015) for example, has popularized the concept of minimally invasive education. This includes encouraging students to develop their own learning strategies in an environment enriched with materials (particularly Internet access) and with minimal instruction.

The pedagogical shift advocated in this chapter is substantially a question of evaluation. If performance is about a good problem solution, then access to the Internet should be a non-issue; only the quality of the student solution is relevant. If the solution or answer is used to assess the student's independent competence or comprehension, then allowing the student to tap a community of knowledge is problematic. However unique the assigned problem, the global community can eventually produce a high level of assistance, through user groups, social networks and contact with students in similar programs. This is poignant in Computer Science, which relies on canonical problems that are easily recognized and paired with solutions on the Internet. The proposal advocated in this chapter is to stop attempting to deny students access to a global community of help, instead to shift the emphasis in evaluating their performance.

2. PROGRAMMING CURRICULA ARE A SEQUENCE OF PROBLEMS

An early problem in computing curricula is *sorting*; it is part of the implicit canon of knowledge in Computer Science, and incorporates both programming and algorithm design, two problem-solving contexts within the discipline. A typical statement of the sorting problem is: "*Write a program which takes an arbitrary sequence of integers as input, and outputs the same integers in ascending sequence*". As a problem in algorithm design, solution strategies for sorting should be compared and contrasted. This will often be undertaken as lecture or presentation material in computing curricula; but some variant of the sorting problem might be given to students as an algorithm design and/or programming assignment.

Explanations, solutions and program code for solving this problem are extensively available on the Internet. An instructor might try to avoid the more popular solutions by assigning a more obscure approach for the student to examine. However, even less popular solution strategies (such as the ShellSort algorithm, cf. Sun Microsystems, 2008) have exhaustive source material and solutions on-line. A student can easily produce a computer program without studying or understanding the problem; they can usually by-pass the explanation part of the solution material and simply copy the program code. (While also true regarding textbook sources, searching tools make the Internet a more attractive option.)

Mark Guzdial (2015) characterizes typical curricula with "most of the learning is expected to occur through the practice of programming", and that there is an assumption that students benefit most by being forced to construct their own solutions, despite longstanding evidence that this is not an effective way to learn programming. Drawing on Kirschner, Sweller and Clark (2006), Guzdial suggests the "minimal instruction" approach denies the student direct instruction on how experts program, while expecting them to develop expertise independently.

Kirschner, Sweller and Clark (2006) is in part a rejection of the constructivist approach to teaching computer programming. Leveraging off the notion that knowledge is personally constructed, constructivism views the students' programming activities as an opportunity to diagnose and correct student misconceptions about programming (Ben-Ari, 1998). Delivery of knowledge in the form of a exiting solution to a problem (according to constructivism) would preclude that opportunity. Since this chapter proposes that student explanations of solutions (which they may acquire from sources such as the Web) might replace student code composition as assigned work, it excludes the use of programming as constructivist activity.

3. THE MEANING OF PROBLEM-SOLVING

It appears that problem-solving skills are typically missing as content from curricula focused on having students practice writing programs. One possible reason for this oversight is the characterization of problems and solutions in computer science, which creates ambiguity around the term *problem-solving*.

In the example offered previously, the problem was sorting, and one possible solution to this problem is embodied in a program. This is typical of the definition of a problem in computing: a problem is defined as creation of a specific algorithm or program to transform of a sequence of symbols (input) to another sequence of symbols (output). A solution (that is, a program that is a solution to the problem) holds no semantic value in the sense that humans might attribute to a problem solution; to give them meaning, input and output

symbols require interpretation, which may be provided by humans looking at the data, or by connecting them to actuators, sensors or robots that interact with the real world. The symbols may indeed mean something – a bank balance, facts, or other abstract concepts - but it requires a human interpreter to infer this meaning from symbols; it is not part of the algorithm "solution" itself. Nor does the program "understand" the input or output, or even its own solution, in any manner that a human might consider "understanding". In fact, when computer scientists talk about program semantics, they are referring to the symbol manipulation that is done by the program, not the "meaning" of the inputs or outputs, the latter being *semantics* as a human might use the word.

As a consequence, a *problem* in computer science is what a program is intended to solve, and the term *solution* is applied both to the program, and the output from a particular run of the program (which solves one instance of the problem represented by one particular set of program input); a *solution strategy* is the algorithmic approach used to solve the problem, and may involve techniques that go by familiar names to expert programmers, such as backtracking, dynamic programming, greedy, branch and bound, and so on. (cf. Cormen, Leiserson, Rivest, & Stein, 2009).

This is distinguished from what we conventionally mean when we talk about problem-solving by humans (or students). Programming or algorithm instructors may wish students to acquire knowledge and expertise of algorithmic solution strategies (such as backtracking, and so on), which may be discussed and illustrated by examples. But this is where confusion in terminology may obscure the curricular objectives. General skills for learning how to create programs from such algorithmic strategies are generally not part of computing instruction. What we want to induce is a process competence for addressing new situations or unseen problems and coming up with solutions (Sternberg, 1995). In the context of learning to program, *problem-solving* is this student's process of creating a program, a *learning strategy* is a particular human skill supporting acquisition of competence in that process, one or more *problem-solving strategies* may be applied during the human's problem-solving process, and a *solution* includes not only the program, but the human interpretation of the semantics of the program.

In computer science jargon, a program *solves* a *problem*, but does not use the *problem-solving* process or strategies engaged by the student to write programs. We don't want the student to simply be able to do what the computer algorithm they create does – a semantically vacuous kind of problem solving that manipulates symbols. We want the student to be able to create new programs with new semantics. As far as that process may involve learning or problem-solving strategies, they would be entirely distinct from strategies employed by an algorithm/program.

The distinction is fairly obvious when it is attended to, but can be obscured in particular context. Exacerbating the situation is the fact that programming problems are often posed using natural language, and then the task of the student is to reduce the problem to symbol manipulation, and then produce the correct symbol-manipulating algorithms as a solution. In producing the algorithm that solves a problem, following instruction in algorithmic strategies for solving problems, the students are doing a different kind of problem solving which involves different kinds of strategies and skills. The terminology is confounded.

Another relevant distinction between these two meanings of "problem-solving" is that the algorithmic solution strategy is observable in the program code. By looking at and studying the program, the solution strategy may be extracted and studied. The skills and strategies for learning to create a program, however, are not self-documenting: they are not

present in the program code, but have hopefully been induced in the student's skill set by making the student write programs. It would be natural for a programming expert, when discussing solutions or methods, to refer to the code as an illustration of the solution technique, rather than to refer to programming skills that are not part of an inspectable software artifact. Thus, relevant skills may be overlooked. This makes it difficult to distinguish the artifact (code or algorithm) from the act, process or skill of creating the artifact. For example, an evaluator or grader will be aware that a student's program solution to the sorting problem needs to solve all possible sorting sequences, not just a specific sorting sequence (what computer scientists call a problem instance). If a student has produced such a correct program, a grader may infer the student has applied and internalized an abstract problem-solving strategy: otherwise, (goes the faulty reasoning) the student could not have produced the program.

Further conflating the problem-solving needed to create programs with the program solution are learning theories that attempt to relate the two (cf. Robillard, 1999). Wing (2006), for example, posits that learning skills can be structured in an algorithmic manner using the moniker "Computational Thinking" (Michaelson, 2015). Literature on the psychology of programming also brings together the psychological aspects of programming and the computational aspects of psychology (cf. Coles, & Ollis, 2015). While including skill acquisition, more of this work relates to modeling expert behavior than to learning.

The type of learning strategies advocated by learning theorists are easily distinguishable from what computer programs do. Falkner et al. (2014) adduce self-regulatory skills computing students need to learn computer programming, such as planning, time management, identifying sub-goals, problem decomposition, task difficulty assessment, knowledge building, as well as meta-strategies such as strategy assessment. They also cite the ability to separate program design from program coding activities as a computer programming skill. Bergin, Reilly, and Traynor (2005) provide evidence of connection between general meta-cognitive strategies and computer programming performance, while Lichtinger and Kaplan (2011) claim self-regulated learning strategies are domain specific. (Caruso, Hill, VanDeGrift, & Simon, 2011) provides evidence that successful student programmers develop their own individualized and idiosyncratic application of problem-solving and learning strategies. None of these are discussing algorithmic strategies embedded in the code of introductory computer programs.

4. INSTRUCTIONAL APPROACH

We address the concern regarding student finding assignment solutions online, in part by recognizing that the programming strategies themselves do not represent program *creation* or problem-solving skills. As pedagogy, this draws on observations that studying (or "reading" code) produces many of the learning outcomes conventionally pursued by assigning programming tasks to students (e.g. Corney et al, 2014; Kirschner, Sweller, & Clark, 2006; Guzdial, 2015).

Direct instruction of meta-cognitive learning skills, self-regulatory skills and problem-solving strategies is advocated by some researchers, and they develop explicit instructional materials for those objectives. Allwood (1986), for example, suggests scaffolding techniques to help students to explore new problem-solving strategies instead of relying on familiar approaches.

However, adoption of such instruction places two important demands on instruction: a particular theory and approach must be selected among competing learning theories, yielding particular strategies to be included; and scheduled time must be available to include new instruction on problem-solving strategies in the course work. The approach discussed here avoids explicit instruction in learning skills or problem-solving strategies, but still attempts to move away from reliance on programming practice as the means to develop programming expertise. A main objective is to reduce the utility of students copying solutions "from the web" without comprehension of the underlying code.

The adopted approach requires students to explain their program-code solutions in a different form than they can readily find on the web. This intentionally separates the comprehension or understanding of a solution from its expression as program code. The following or a similar address to students is repeated during the course: "*Unlike other programming courses you may have done, I don't care how you get the code you use, as long as you have permission and you attribute it appropriately in compliance with copyright and other legal requirements. While you might benefit more from composing your own, you can get code from friends, off the Internet or even purchase it. The quality of the code you submit is important, but your evaluation will be based on the understanding and explanation of the code provided in your submission. I will be calling this aspect of submission the* code design."

By shifting evaluation from the quality of the program code to the competency reflected in the individual student's explanation, we attempt to effect two things: (1) eliminate students obtaining credit for code they can acquire but do not understand and (2) force some (perhaps implicit) attention by the student and the instructor on how the student develops an understanding of the program code. In other words, we attempt to engage problem-solving strategies even if they are not explicitly discussed.

In itself the emphasis on design or an abstract description of program code is not innovative. Descriptions of software design are essential in advanced computer subjects such as software architecture. The innovation here is using design descriptions rather than implemented program code as evidence of programming competency, even in more introductory courses where it is not usual.

Work in concept mapping (Novak and Gowin, 1984) supports the belief that students are forced to engage and reify new knowledge when they have to express textual material in a different (diagrammatic) form. Here we pursue an analogous outcome using program design descriptions as the expression of the text of computer program code. Concept mapping has been applied for diagnosing student misconceptions (Sanders, Boustedt, Eckerdal, & McCartney, 2008) and for teaching programming concepts using Unified Modelling Language (UML) (Ferguson, 2003; Tabrizi, Collins, Ozan, & Li, 2004). Essentially, the claim is that in representing program code in a different form (UML), students will have to form a deep understanding of the source material.

Using UML to create abstract models of software systems would be a skill introduced to students at some point in an academic degree stream for computer programming. The difference here is to treat tools such as UML as a way to establish student comprehension of other academic material, rather than as part of the material themselves. To illustrate the difference in using these representations, consider the JAVA program code snippet shown in figure 1.

Figure 1. Taken from openjdk JAVA 6b14 listing for AbstractCollection.contains method.

```
public boolean contains(Object o) {
Iterator<E> e = iterator();
if (o==null) {
while (e.hasNext())
if (e.next()==null)
return true;
} else {
while (e.hasNext())
if (o.equals(e.next()))
return true;
}
return false;
}
```

Figure 1 is an example of a code segment provided with the standard JAVA libraries used by virtually all JAVA programmers. In this case, it provides an example of *iteration*, a sequential programming construct easily expressed in consecutive lines of code. It is difficult to imagine a more concise representation of iteration than the original source code. Depicting this concept in UML would be less concise or expressive than the original code.

Figure 2. UML representation of JAVA interfaces in the Collections framework.

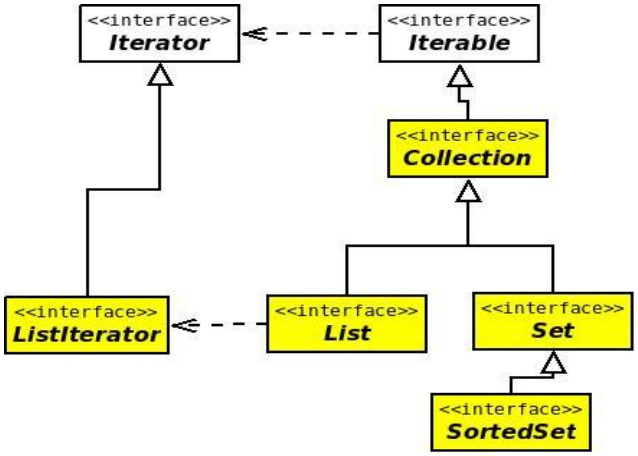


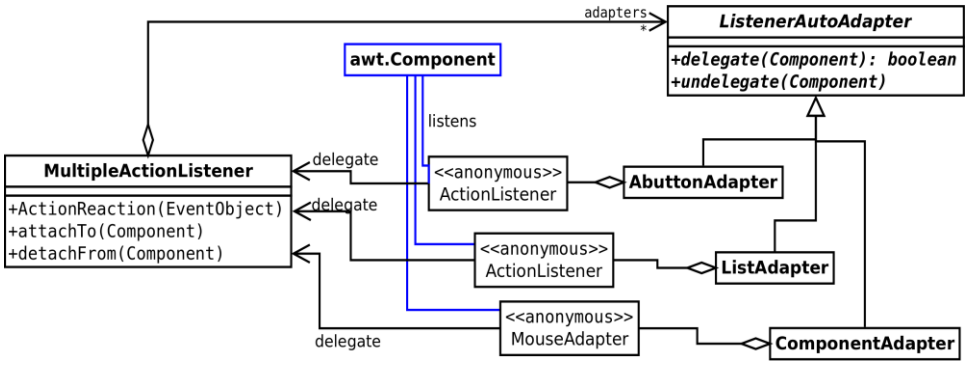
Figure 2, in contrast, uses a notation called a UML class diagram to depict relationships between a set of code constructs called *interfaces.* In the JAVA code, these relationships are scattered throughout the standard JAVA libraries, in lines of code that are non-consecutive and span may pages. It requires some minimal competence to find and conceptually synthesize the relationships depicted in figure 2. UML is much more concise than the original code in presenting this design aspect of the program code.

Our conceit is that requiring students to translate or express aspects of their program solution into UML encourages understanding and thinking about the program code. Efficient expression of programming concepts is not the point, although that helps motivate student use of the notation. Forcing the students to produce correct representations and

make choices about what code aspects to convey means they have to study and understand the code, even if it is not their own original composition. Rather than simply mimic or cut-and-paste from a compendium of coding solutions, students have to internalize the solution well enough to express it in a radically different notational form. The struggle to acquire new understanding is forced on them in a way that cannot be circumvented by generalized learning strategies. It is not the graphical depiction that is important, any more than it is the running program that is important: it is the students' understanding that is engendered by the task of producing an original explanation of the code that is important.

Although our hope and belief is that students adopt successful problem-solving strategies without them being explicitly taught or evaluated, the need to model the use of notational explanations for the students is clear. Figures 3 and 4 are typical classroom materials involved in such modeling. Figure 3 depicts some aspects of a programming solution to a posed problem. In addition to discussing and critiquing the solution depicted, the choices made in constructing the diagram are also discussed with the students: what elements are included, and what is left out for clarity. What unimportant details are omitted. What structures are important to include and which are not. How additional concepts might be included. Certainly such discussion implicitly engages *some* relevant domain-specific problem solving skills, such as problem decomposition.

Figure 3. A UML diagram of a code solution that was critiqued with students. Diagram illustrates one particular design concept used in the coded solution. Students use similar diagrams in their own submissions.



Since UML is poor at presenting certain aspects of programming solutions, (as discussed with respect to figure 1), programming code is still discussed and analyzed as part of class presentations where they are the better choice for representing solution concepts. Figure 4 shows some code used in class presentation to discuss different aspects of the same solution depicted in figure 3. For other aspects, natural language is more effective than either UML or th programming language, and students are directed to consider what is best conveyed by these alternatives: UML, program code excerpts, and natural language. Choice of these alternatives representational choices is also incorporated into class presentations and discussion, impressing on the students that these choices also reflect their understanding and the quality of their work submissions.

Figure 4. Part of the code solution represented in figure 3.

```
/**
 *
 * This adapter sticks to any component and captures mouse click events.
 * This should be the last one tried.
 * @author brown
 *
 */
class ComponentAdapter implements ListenerAutoAdapter {
        MouseAdapter adapterToDelegateListener;
                public ComponentAdapter() {
                adapterToDelegateListener = new MouseAdapter() {
                                @Override
                                public void mouseClicked(MouseEvent e) {
                                        MultipleActionListener.this.actionReaction(e);
                                }
                };
        }
        @Override
        public boolean delegate(Component j){
                j.addMouseListener(adapterToDelegateListener);
                return true;
        }
        @Override
        public boolean undelegate(Component j){
                j.removeMouseListener(adapterToDelegateListener);
                return true;
        }
}
```

5. CONCLUSION/DISCUSSION

The described approach was piloted in a third year course titled “COMP3718: Programming in the Small” at Memorial University, Canada during the first semester January-April of 2015. The primary changes from previous course offering were (1) the emphasis on analyzing and understanding program code rather than composing code; (2) evaluation based on students' original explanations of programs rather than the quality of student-authored programs; and (3) the provision that student may incorporate code recruited from any legitimate source rather than having to compose their own.

The principle distinctions from pedagogy reported in the literature which also emphasize student analyzing and reading (as opposed to only composing) program code are (1) there was no explicit instruction related to problem-solving skills or learning strategies other than algorithmic-programmatic solutions to software problems (2) while coding standards were part of the course, the code explained in student assignments was sourced (chosen or created) by the students themselves, instead of being selected by the instructor for its instructional merit (3) there was no staging that required students to advance past a “code reading and comprehension” stage to a nominally advanced “compose your own” stage in learning programming (Guzdial, 2015).

No formal data collection occurred during this pilot, but some general observations are offered. Compared to previous offerings, both student grades and institutional student evaluations of course instruction were slightly improved. Without controls over other instructional factors, improvement cannot be attributed to any particular cause; however,

there is at least a subjective impression that the modifications discussed in this chapter did no noticeable harm to student performance.

One informal observation is that many students had difficulty adjusting to the shift in evaluation criteria. At mid-term, (after half the assignments were completed), some students' comments on their assignment evaluations reflected conventional expectations. A paraphrase of one comment is “My code works perfectly, so I don't understand how I can have a fail on the assignment”. Similarly, a student who lost marks for inadequate explanation wrote “How do you know I didn't do that?” The point that the scheduled marks were for explanation of the code, not for code quality, had not been understood. In personal communication reviewing this issue with some students, several agreed with the suggestion that their expectations were driven by experience in prior courses. An alternative suggestion, that students enjoy programming code and not creating descriptions, was generally rejected as an explanation of student expectations by those interviewed. These comments warrant additional effort to address student expectations regarding evaluation. The following modifications are planned for future instruction: (1) some assignments which involve NO code submission, only description of existing code, will be used to prime student expectations, (2) specific introduction to evaluation schemes, (3) an example of a completed and graded assignment will be provided early in the semester, (4) lecture time modeling the practices of describing and evaluating code will be increased and further integrated into the programming elements of the curriculum.

A further unprompted comment from several students was that the course was challenging in ways they did not anticipate, and in particular that they had difficulty “finding examples on the Internet”. This provides some hope that the approach is achieving its material purpose of removing the Internet as a compendium of solutions, forcing students to apply cognitive effort in study of their submissions.

A parenthetical observation is that although students were not required to author their code submissions, a substantial amount (more than half) of submitted code was indeed authored by the students. One speculative suggestion is that it is easier to explain your own code than to study, understand and explain someone else's code. It is also commonly asserted that programmers enjoy and feel accomplishment in the act of producing their own programs.

This project could now move past a pilot stage, entertaining data collection for several outstanding questions. A comparison to approaches that adopt specific learning theories and explicit instruction in specific problem-solving skills (e.g. Falkner et al., 2014) could be assessed by identifying the problem-solving strategies students are adopting under this approach. It would also be interesting to determine how students are sourcing the computer program code used in their submissions, to assess how code creation and associated programming skills are impacted.

If generalization of this approach to subject matter other than computing is warranted, a difficulty is analogizing the act of describing code to some activity in another discipline. The author tentatively suggests that student self-critique and self-evaluation of solutions may provide similar constructs, and the general literature on student self-regulated learning (cf. Lichtinger & Kaplan, 2011) could be interpreted as supporting this view.

REFERENCES

Allwood, C. (1986). Novices on the computer: a review of the literature. *International Journal of Man-Machine Studies, 25*(6). 633–658.

Ben-Ari, M. (1998). Constructivism in computer science education. *Proceedings of SIGCSE '98 the twenty-ninth SIGCSE technical symposium on computer science education*, 257-261.

Bergin, S., Reilly, R., & Traynor, D. (2005) Examining the role of self-regulated learning on introductory programming performance. *Proceedings of ICER'05 the first international workshop on Computing education research*, 81-86.

Brown, E. (2015). Problem solving as program code description *Proccedings of International Conference on Education and New Developments*, *2015*, 13-17.

Coles, M., & Ollis, G. (2015). Message from the chairs. *Proceedings of the Psychology of Programming Interest Group Annual Meeting, 2015*.

Caruso, T., Hill, N., VanDeGrift, T., & Simon, B. (2011) Experience report: Getting novice programmers to think about improving their software development. *Proceedings of SIGCSE'11 42nd ACM technical symposium on computer science education*, 493–498.

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). Introduction to Algorithms (3rd Ed.). Massachusetts Institute of Technology.

Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., & Murphy, L. (2014) Explain in Plain English questions revisited: data structure problems. *Proceedings of SIGCSE'14 45th ACM technical symposium on Computer science education*, 591-596.

Falkner, K., Vivian, R., & Falkner, N. (2014) Identifying computer science self-regulated learning strategies. *Proceedings of the 2014 conference on Innovation & technology in computer science education, ITiCSE '14*, 291-296

Ferguson, E. (2003) Object-oriented concept mapping using UML class diagrams. *Journal of Computing in Colleges, 18*(4). 344-354.

Guzdial, M. (2015) What's the Best Way to Teach Computer Science to Beginners? *Communications of the ACM, 58*(2), 12-13.

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006) Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. Educational Psychologist, *41*(2), 75-86.

Lichtinger, E., & Kaplan, A. (2011) Purpose of engagement in academic self-regulation. *New Directions for Teaching and Learning, Special Issue on Self-Regulated Learning, 2011*(126), 9–19.

Michaelson, G. (2015) Teaching Programming with Computational and Informational Thinking. J*ournal of Pedagogic Development, 5*(1), 51-66.

Mitra, S. (2015). Minimally invasive education: Pedagogy for Development in a Connected World. In Rothermel, P. (Eds.), *International Perspectives on Home Education* (pp. 254-277). New York: Palgrave-McMillan.

Novak, J., & Gowin, D. B. (1984) *Learning how to Learn*. Cambridge University Press.

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM, 42*(1), 87-92.

Sanders, K., Boustedt, J., Eckerdal, A., & McCartney, R. (2008) Student understanding of object-oriented programming as expressed in concept maps. *ACM SIGCSE Bulletin, 40*(1), 332-2336.

Sternberg, R. J. (1995). Conceptions of expertise in complex problem solving: A comparison of alternative conceptions. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 295-321). Hillsdale, NJ: Lawrence Erlbaum Associates.

Sun Microsystems Inc. (2008). *Shell Sort Implementation in Java*, Retrieved May 4, 2015 from http://www.java-tips.org/java-se-tips/java.lang/shell-sort-implementation-in-java.html

Tabrizi, M. N. H., Collins, C., Ozan, E., & Li, K. (2004) Implementation of object-orientation using UML in entry level software development courses. *CITC5 '04 Proceedings of the 5th conference on Information technology education*, 128-131.

Treude, C., Barzilay, O., & Storey, M. (2011). How do programmers ask and answer questions on the Web? *Proceedings of ICSE'11 33Rd International Conference on Software Engineering*, 804-807. Waikiki: Honolulu.

Veenman, M., Elshout, J., & Meijer, J. (1997). The generality vs. domain-specificity of metacognitive skills in novice learning across domains. *Learning and Instruction, 7*(2), 187–209.

Wing, J. M. (2006) Computational Thinking, *CACM Viewpoint*, March 2006 (pp. 33-35).

Winne, P. (1995). Inherent details in self-regulated learning. *Educational Psychologist, 80*(4), 284–290.

AUTHOR(S) INFORMATION

Full name: Edward Brown
Institutional affiliation: Memorial University of Newfoundland
Institutional address: Elizabeth Avenue, St. John's. NL, CANADA, A1C 5S7
Short biographical sketch: Dr. Brown is an Associate Professor of Computer Science and holds a Ph.D. in Education from the University of Toronto, and a J.D. in Law from the University of Victoria. Canada. His research interests revolve around software law, privacy, and mobile applications.