DeepCode: An Annotated Set of Instructional Code Examples to Foster Deep Code Comprehension and Learning

Vasile Rus(✉), Peter Brusilovsky, Lasang Jimba Tamang, Kamil Akhuseyinoglu, and Scott Fleming

The University of Memphis, Memphis, TN 38152, USA
vrus@memphis.edu

Abstract. We present here a novel instructional resource, called DeepCode, to support deep code comprehension and learning in intro-to-programming courses (CS1 and CS2). DeepCode is a set of instructional code examples which we call a codeset and which was annotated by our team with comments (e.g., explaining the logical steps of the underlying problem being solved) and related instructional questions that can play the role of hints meant to help learners think about and articulate explanations of the code. While DeepCode was designed primarily to serve our larger efforts of developing an intelligent tutoring system (ITS) that fosters the monitoring, assessment, and development of code comprehension skills for students learning to program, the codeset can be used for other purposes such as assessment, problem-solving, and in various other learning activities such as studying worked-out code examples with explanations and code visualizations. We present here the underlying principles, theories, and frameworks behind our design process, the annotation guidelines, and summarize the resulting codeset of 98 annotated Java code examples which include 7,157 lines of code (including comments), 260 logical steps, 260 logical step details, 408 statement level comments, and 590 scaffolding questions.

Keywords: Code comprehension · Intelligent tutoring systems · Self-explanation

1 Introduction

Code comprehension, i.e., understanding of computer code, is a critical skill for both learners and professionals. Students learning computer programming spend a significant portion of their time reading or reviewing someone else's code (e.g., code examples from a textbook or provided by the instructor). Furthermore, it has been estimated that software professionals spend at least half of their time analyzing software artifacts in an attempt to comprehend computer source code. Reading code is the most time-consuming activity during software maintenance, consuming 70% of the total lifecycle cost of a software product [5, 8, 32]. O'Brien [24] notes that source code comprehension is required when a programmer maintains, reuses, migrates, reengineers, or enhances software systems.

S. Crossley and E. Popescu (Eds.): ITS 2022, LNCS 13284, pp. 36–50, 2022.
https://doi.org/10.1007/978-3-031-09680-8_4

Therefore, offering support to enhance learners' source code comprehension skills will have lasting positive effects for their academic success and future professional careers.

Our goal is to explore instructional strategies that promote deep code comprehension and learning. To this end, we present here a novel instructional resource, called DeepCode, which is a set of annotated code examples to support code comprehension and learning activities in intro-to-programming courses (CS1 and CS2). Indeed, the design of DeepCode was driven by our larger efforts on exploring instructional strategies that foster the development of code comprehension skills and the construction of accurate mental models and learning in conjunction with advanced education technologies such as conversational intelligent tutoring systems (ITSs; [33, 34, 45]) that use scaffolded self-explanations to foster the monitoring, assessment, and development of code comprehension skills for students learning to program. It is important to note that ITSs are a particular category of adaptive instructional systems (AISs) that offer both micro-level and macro-level adaptivity, as explained later.

While we developed DeepCode to serve our goal of developing an ITS for code comprehension and learning of programming concepts, the resulting codeset of annotated code examples can be used for other purposes such as assessment, problem-solving, and in various other learning activities such as studying worked-out examples or code visualizations as well as for research purposes such as exploring other instructional strategies, e.g., asking students to Explain-in-Plain-English target code examples (EiPE; 9, 22, 41).

The code examples included in the DeepCode dataset cover the vast majority of topics in a typical intro-to-programming course (CS1) - see the list of topics later. While the DeepCode instructional dataset contains Java code examples, the design principles and annotation guidelines are generally applicable to any programming language. The annotation guidelines were based on code comprehension theories, self-explanation theories, and the micro-macro adaptivity framework used by ITSs.

We are not aware of any similar resources available for code comprehension and learning activities that have been publicly released and which cover the vast majority of topics in CS1. Furthermore, one unique feature of our annotated code examples is the theory-driven annotation guidelines.

Work in the area of code comprehension targets a subset of CS1 topics and usually do not release code examples for use by others [2, 3, 27, 28]. Recent work on Explain-in-Plain-English (EiPE; 9, 24, 42) use large sets of code examples, e.g., Chen and colleagues [9] use 52 code examples scattered across various courses and course related activities: a CS1 for engineers' course (8 homework questions), 5 homework and exam problems in a 'CS1 for CS majors' course, 26 exam problems of which each student was assigned one problem on each of 5 exams, in a data structures course, and 13 additional questions as part of a paid survey that was offered to sophomore-level CS students. It should be noted that in EiPE tasks, learners/readers are asked to provide a high-level natural language (e.g., English) description of the code which is different from asking them to self-explain as detailed later. To the best of our knowledge, there was no principled way, theory-driven selection and annotation of the 52 code examples.

Other notable efforts to create questions/problems in CS Education fall into two broad categories: (1) creating traditional test questions, usually in multiple-choice format such as Canterbury Bank [31] and (2) creating more advanced learning content such as explained worked examples [7], codecasts [36], code animations [20, 38], Parson's problems [22], and code construction problems with automatic assessment [17, 39]. All these kinds of learning content are based on meaningful code examples, which could be explained, animated, or presented in a form of a problem. Surprisingly, research in this direction predominantly focused on creating authoring tools that allow end users create advanced content items leaving it to the users to create useful collections of examples or problems. Moreover, no theory-based guidelines for creating problems or examples are usually offered. In this context, our work bridges the gap between these two research direction. Similar to the motivation of question-bank developers, we focus on a collection of quality content. However, in contrast to relatively simple content in question banks, we focus on complete meaningful augmented code examples. This code examples could be used as-is in its original form or serve as a basis for creating collections of quality examples and problems using the advanced content authoring tools mentioned above.

The DeepCode codeset has the following main features meant to promote deep comprehension of code and learning of computer programming concepts:

- Explanations in the form of logical step comments capturing the domain model;
- Explanations in the form of logical step details comments capturing the program model and linking the program model and the domain model, i.e., the integrated model;
- Explanations of new concepts being introduced by each code example.
- Scaffolding hints in the form of questions for the domain model, program model, and the integrated model as well as for the new concepts (enables micro-adaptation).
- Topic ordering based on input from CS1 and CS2 instructors and prior research on programming concept difficulty and importance (enables macro-adaptation and implementation of learning strategies such as mastery learning and spacing effects).

2 Related Work

The development of the DeepCode set of instructional code examples has been guided by a number of theories and frameworks and recent advances in code comprehension and text comprehension research of which the following are the most important: reading/code comprehension theories [6, 16, 18, 19, 21, 26, 35, 37, 44], cognitive load theory [41], cognitive engagement theory [13, 14] and the ICAP framework [13], self-explanation theory [10, 12], and the intelligent tutoring framework that offers macro- and micro-adaptive instruction [45] with a focus on conversational tutoring that implement scaffolded self-explanation strategies [33] and related efforts such as the conversational tutor for program planning ProPL [24].

For instance, according to Pennington's theory of code comprehension [25] involves the building of a domain model and of a program model, as detailed later. Accordingly, the DeepCode annotation guidelines specify adding comments, which we call logical step comments, describing the code from a domain perspective, i.e., describing the domain model. For each logical step, details of how those logical steps are implemented in a chunk of code are needed as well. This is meant to describe the program model as well as link the domain model to the program model. By the same token, based on cognitive load theory, each code example introduces only one new topic or concept and only individual statements that refer to the new concept that learners are supposed to learn are annotated with statement level comments. Concepts in other lines of code are supposed to have been mastered earlier while working on code examples corresponding to topics introduced earlier in the sequence of topics. Indeed, code examples are sequenced based on a priori defined sequence of topics and each code example is supposed to use only concepts related to previously mastered topics and the new topic, as detailed later. Scaffolding questions were also annotated for the logical step and logical step details comments in order to help with the development of scaffolding tutorial dialogues.

Of particular interest to our larger goals of building ITSs that scaffold learners' code comprehension processes are self-explanation theories and the micro-/macro-adaptive framework for tutoring. Self-explanation theories [10, 12] indicate that students who engage in self-explanations, i.e. explaining the target material to themselves, while learning are better learners, i.e. learn more deeply and show highest learning gains. The positive effect of self-explanation on learning has been demonstrated in different science domains such as biology [11] and physics [15], math [1], and programming [3]. Furthermore, research has shown that guided self-explanation is effective too [1]. Self-explanation's effectiveness for learning is attributed to its constructive nature, e.g., it activates several cognitive processes such generating inferences to fill in missing information and integrating new information with prior knowledge, and its meaningfulness for the learner, i.e., self-explanations are self-directed and self-generated making the learning and target knowledge more personally meaningful, in contrast to explaining the target content to others [30]. Several types of self-explanation prompts have been identified and explored such as justification-based self-explanation prompts [15] and meta-cognitive self-explanation prompts [11]. The code examples in DeepCode could be used with various prompts and we intend to do so in order to elicit from students a variety of responses to capture as much about their mental models and mental model construction processes as possible.

As already noted, our larger goal is to build an ITS for code comprehension in intro-to-programming courses. The behavior of any ITS, conversational or not, can be described using VanLehn's two-loop framework [45]. According to VanLehn, ITSs can be described in broad terms as running two loops: the outer loop, which selects the next task to work on, and the inner loop, which manages the student-system interaction while the student works on a particular task. The outer loop provides macro-adaptivity, i.e., selects appropriate instructional topics and tasks for a learner to work on, e.g., based on their mastery level. In order to offer outer loop support, DeepCode is based on a sequence of intro-to-programming topics and instructional tasks per topics as explained later.

The inner loop of an ITS monitors students' performance through embedded assessment, updates its model of students' levels of understanding (that is, the student model), and uses the updated student model to provide appropriate scaffolding in the form of feedback and other scaffolds. Accordingly, each code example in DeepCode is annotated with hints in the form of questions which are meant to scaffold learners' comprehension processes. Conversational ITSs can use the instructional code examples and annotations, e.g., the hints, to interactively monitor, assess, and scaffold learners' comprehension and learning. This should lead to best learning outcomes, according to the Interactive, Constructive, Active, and Passive (ICAP; [13]) framework of cognitive engagement according to which interactive learning leads to best cognitive engagement.

As a way to illustrate how these various theories provided the foundations for our work, we will use a concrete example and present step by step the annotation guidelines and process with references to the underlying theories.

3 A Working Example

In order to better illustrate the guiding principles and theories underlying the development of the DeepCode annotation guidelines, we will make use throughout the paper of a concrete example related to the widely played game of Bingo - a simplified version of the game to be precise. The game is played with disposable paper boards which contain 25 squares arranged in five vertical columns and five rows. Columns are labelled 'B', 'I', 'N', 'G', 'O'. The cell in the middle is empty. Random numbers from 1 to 75 are used in the game and there are some restrictions on the range of values that can occur in each column, e.g., the 'B' column only containing numbers between 1 and 15 inclusive whereas the 'I' column containing only 16 through 30. We will work with a simplified Bingo game in which all 25 cells may contain any number between 1 and 75. Players must match rows, columns, or diagonals in randomly generated Bingo board.

Our task is to solve computationally, i.e., with the help of a computer program, the following problem: Automatically generate random boards for the (simplified) game of Bingo.

The Java code implementing the solution for this Bingo board generation task is shown in Fig. 1. Details about the annotations, i.e., the explanations added in the form of comments are provided next together with the underlying theoretical foundations. Figure 1 does not show all the annotations added because of space reasons. The missing annotations are exemplified throughout the narrative of the paper.

4 The DeepCode Annotation Guidelines

We now present a summary of the annotation guidelines for developed DeepCode, exemplifying the various elements and the underlying theories that form the foundations for those elements.

Our entire team was involved in the development of the DeepCode codset. Each team member was assigned a number of topics for which to create and annotate code examples. They were instructed to either identify code examples in various sources (textbooks, websites, etc.) and use them as they are (i.e., just focus on annotating them with comments/explanations) or modified, or create their own examples from scratch. In case a particular source was used, the annotators were supposed to specify the source as detailed later. All annotators went first through a training period and had access to a detailed annotation guidelines manual and a cheatsheet.

The general steps to create and annotate code examples for the DeepCode codeset were:

1. Create at least 4 code example for a target topic. We create 4 code examples for each topic to allow students who struggle to practice the same topic again following the master learning principle (Bloom, 1981).
2. Add metadata.
3. Identify major logical steps and add corresponding logical step comments.
4. Add logical step details describing how the logical step is carried out using programming concepts.
5. Generate statement level comments for specific lines of code referring to the newly introduced topic.
6. For each logical step and new concept references (captured by statement level comments), generate a sequence of instructional hints in the form of questions which can be used to help students understand and articulate the logical step and its details.
7. Add misconception detection information and remedial feedback.

Step 1: Code Example Creation. The general guidelines given to annotators to create code examples are given below:

– create code examples for the topics you were assigned make sure you are aware of the topic/concept ordering for the whole CS1/CS2 sequence as it is important for the code example creation and annotation. The list of topics and their order is: Preliminary Topics (Variables + Expressions + Constants + Primitive data type), Input, Math, Class, Strings, Logical Operators, If, If-else, Switch, While Loops, Do While, For loops, Nested Loops, Arrays, Two Dimensional Arrays, Array Lists, Classes + Objects, Methods, Inheritance, Exception Handling, Recursion, Sorting, and Searching.
– Each code example should focus on the topic for which the example is targeted and may rely on concepts/topics covered earlier in the ordered list of topics. This is meant to reduce the cognitive load on novices trying to learn programming. According to cognitive load theory [41], humans have a limited capacity working memory and an unlimited long-term memory. Therefore, during learning activities instructional strategies should minimize the short-term memory load and encourage the construction of knowledge structures, i.e., schemas, in long-term memory. To this end, each code example introduces one new concept/topic or subconcept/subtopic.

- The examples should have deterministic output if at all possible so that an intelligent tutoring system would be able to assess the correctness of student predictions. This may not always be possible, e.g., when input is required from the user or when a random process is involved.
- Each code example should be nicely edited using a Java/code editor that can format.
- Each code example should be built around a story, i.e., a real-life application of the code should be thought of that is meaningful to students – something they can relate to from their own life experience. This guideline plays a motivational role because using such relatable stories could lead to a more meaningful effort on learners' part when trying to understand computational solutions to relatable challenges (as opposed to unfamiliar or abstract ones). Furthermore, using real-life applications which students can relate to it through their own life experience, minimizes the need for domain knowledge to understand the code examples (i.e., general world knowledge would suffice) should reduce the cognitive load on the learners allowing them to allocate cognitive resources on the core programming concepts to be learned. Examples of real-life applications or problems are feet to meter conversion, leap year detection, or Bingo boards.
- Code examples should compile and run as expected. Once a code example was created, compiled, and executed without errors, it needs to be augmented with metadata and instructional comments as detailed below.

Step 2: Metadata. Each code example is annotated with a header that specifies the annotator/author, topic(s), subtopic(s), source (if any), goal (what the code does, i.e., the problem it solves), input (if any), and output. The granularity of topic/subtopic is an issue in itself. The Java example in Fig. 1 does not offer all the corresponding metadata due to space reasons. We plan to publicly release the fully annotated examples as a GitHub repository once the publication of this work is being accepted.

Step 3: Identification of Logical Steps. This step is about annotating the code examples with high-level explanations describing the logical steps of the problem being solved. This guideline is based on program comprehension theories proposed over the past 50 years or so [6, 16, 19, 25, 28, 32, 34, 36, 39). A major problem with the traditional program comprehension models is that they were the result of analyzing expert programmers' comprehension processes as opposed to novices', i.e., individuals with no or almost no relevant knowledge. More recently, there is work addressing this issue such as Schulte and colleagues [34] who proposed an education comprehension model.

While the various models of code comprehension differ in what their main focus is, they all share the following majors components [25, 34]: an external representation – external views or aids assisting the programmer in comprehending the code, a knowledge base – the programmers' knowledge, a situation/mental model – programmer's current understanding of the code and which is constantly updated through the assimilation process, and an assimilation process – the process through which the situation model is being updated based on the knowledge base, external representation, and the current situation model. The knowledge base and the situation model are sometimes conflated together under a broader cognitive structures label/category [34].

Given that learners lack (most of or all of) the much needed knowledge base, we face a catch-22 challenge:

In order to read and understand code the reader (learner in our case) needs a knowledge base; however, if the reader is someone who just starts learning programming then the knowledge base is empty or almost empty which means the learner needs to build their knowledge base which can be done by "looking at", i.e., reading, code examples which means the reader must understand them which is what they are trying to achieve in the first place bringing them back full-circle. In order to transform this vicious circle into a virtuous one, external support in the form of scaffolding offered by a human or a 24/7 computer-tutor is critical, which is our larger goal.

To this end, the DeepCode codeset and the corresponding pedagogical comments were designed to compensate for the lack of a 'knowledge base' of students in intro-to-programming courses and offer necessary support when needed to both help students understand target code examples and learn newly introduced programming concepts and techniques.

Of particular importance to our work presented here is the distinction between the program model, the domain model, and the situation model [26]. The program model is some representation of the control-flow of the program or what we call a direct mental equivalent of the code. The domain model captures the function or goals of the program from a target domain perspective, i.e., it describes the domain problem and the solution being implemented by the code by referring mostly to objects and relations and processes and approaches of the domain and of the problem being solved. The situation model in our view captures an integrated view of both the program and domain model with an emphasis on cross-references between the two models, i.e., it contains information which is not being captured by the individual program and domain models. In fact, there is evidence that the best code readers are those who can build such an integrated situation model by seeking to understand and infer cross-references between the program and domain models.

Accordingly, we have focused on pedagogical comments that correspond to the domain model (logical level comments) and program model (logical step implementation details) and situation model (logical step details provide cross-references between the program and domain models).

The logical steps and corresponding comments describe the logical steps of the overall algorithm implemented in the code. Logical steps are meaningful, higher-level steps in the overall solution/algorithm implemented by the code. It is not necessary to describe in detail how the step is being implemented but simply indicate the meaningful purpose/functionality of each such logical code chunk in the context of the overall goal/purpose of the code. A logical step comment should be a concise sentence referring mostly to objects and relations of the domain/problem being solved as shown below (see also Fig. 1 which shows all the logical step comments).

logical_step_2: Generate 25 random numbers in the range of 1 to 75 and populate the Bingo board.

Cross-references/usage of concepts from the 'program model', i.e., implementation, should be avoided or kept at a minimum. The logical_step_details field links the problem/domain model to the implementation.

Step 4: Logical Step Details. Whereas the logical step is a very high level explanation meant to link the code to the problem/story at a very high level, the logical_step_details explanation provides details about how the logical step is being carried out.

The need for the logical_step_details is to provide a link between logical details and implementation details while keeping the logical step description high level and short (one short or medium size sentence in plain language - minimal programming language specific lingo, domain knowledge lingo is acceptable, e.g., soccer lingo, but should be kept at a minimum, if at all possible, so that all students can understand the problem and the solution. Additional explanations of domain knowledge concepts should be added if needed).

logical_step_details: Two loops are used to scan all the cells on a Bingo board. The outer loop accounts for the rows and the inner loop for all the cells in one row. For each scanned cell on the Bingo board, a random number is generated and stored in the cell.

Step 5: Statement Level Comments. Statement level comments focus on individual statements and emphasize the elements of the statement relevant to the new concept being taught. It refers more to the 'program model', i.e., to concepts, steps, and functions related to implementation. The statement level comment and related question should not necessarily be about the general function of the statement but rather focus on the parts related to the new concept.

stm_comment: Declare an array variable called myNumber of type integer and size 11 and allocate memory for it.

It should be noted that as another way to reduce cognitive load on learners, our guidelines are based on a code comprehension scaffolding strategy that focuses on eliciting explanations at the logical level of code understanding and of statements that refer to the new concept/topic being introduced by each code example - other statements, while important for understanding, are not explicitly explained as they refer to prior concepts which the learners should have mastered previously when those concepts were introduced previously in the sequence of concepts.

Step 6: Adding Hints in the Form of Questions and the Corresponding Answers. In order to support the development of advanced education technologies such as ITSs that provide micro- and macro-level adaptation through interactive scaffolding, for each logical step comment and logical step details comment, we added a sequence of hints in the form of questions meant to help learners think and articulate about the logical steps and logical step details. That is, the goal is to use those hints to scaffold students' self-explanations of the logical steps and logical step details and statement level explanations. The first question in the sequence elicits the logical step (domain model) whereas the subsequent questions should prompt learners key aspects of the logical step details (program and integrated models). For each question the corresponding answer is provided as well in order to facilitate the automated assessment of student responses to those

hints, e.g., by comparing the student responses to these ideal responses we provided for each hint using automated semantic similarity methods [33]. Figure 1 does not show the questions for any of the comments for space reasons. We illustrate below the kind of question sequences for logical step and logical step details comments.

/**
* logical_step_2: Generate 25 random numbers in the range of 1 to 75 and populate the Bingo board.
* logical_step_details: Two loops are used to scan all the cells on a Bingo board. The outer loop accounts for the rows and the inner loop for all the cells in one row. For each scanned cell on the Bingo board, a random number is generated and stored in the cell.
* question_1: What does the following code block do?
* answer_1: Generate 25 random numbers in the range of 1 to 75 and populate the Bingo board.
* question_2: How many times does the outer loop execute?
* answer_2: The outer loop iterates 5 times.
* question_3: How many times does the inner loop execute?
* answer_3: The inner loop executes 5 times.
*/

Questions and corresponding benchmark answers were generated as well for statement level comments.

/**
* stm_comment: Print element of the Bingo board at position indicated by row i and column j.
* question_1: Which element of the array bingoBoard is being displayed?
* answer_1: Element of the Bingo board at position indicated by row i and column j is being displayed.
*/

Step 7: Annotating Misconceptions and Corresponding Remedial Feedback. A key instructional goal for any instruction effort, computer-based or otherwise, is to identify students' misconceptions and provide remedial feedback immediately. For our running code example, a typical misconception is the index of the last row and column of the matrix representing the Bingo board. We can trigger a question to prompt for an answer to discover the presence of the misconception in any or all the lines of code where the matrix is being referred, e.g., immediately after the bingoBoard matrix is declared or when the matrix is being scanned.

Misconception: The index of the last row of the bingoBoard matrix is 5.
Remedial feedback: The index of the last row of the bingoBoard matrix is 4 as indices run from 0 to the number of rows minus 1.
Triggering questions: What is the index of the last row of the bingoBoard matrix?

```
/**
* SEE METADATA SECTION
*/
import java.util.Random;

public class twoDimensionalArraysBingoBoard {
  public static void main(String[] args) {

    /**
    * logical_step_1: Declare variables needed to represent the Bingo board and generate random numbers.
    */
    int[][] bingoBoard = new int[5][5];
    Random rand = new Random();

    /**
    * logical_step_2: Generate 25 random numbers in the range of 1 to 75 and populate the Bingo board.
    */
    for ( int i = 0 ; i < 5 ; i++ )
    {
     for ( int j = 0 ; j < 5 ; j++ )
     {
      while ( (bingoBoard[i][j] = rand.nextInt (75)) == 0 ) ;
      /**
      * stm_comment: Print the element of the Bingo board at position indicated by row i and column j.
      */
      System.out.print( "board square [" + i + ", " + j + "]" + " = " + bingoBoard[i][j] + "\n") ;
     }
     System.out.println( "" ) ;
    }
    /**
    * logical_step_3: Print the Bingo board.
    */
    for ( int i = 0 ; i < 5 ; i++ )
    {
     for ( int j = 0 ; j < 5 ; j++ )
     {
      /**
      * stm_comment: Print element of at row i and column j on the Bingo board.
      */
      System.out.print(bingoBoard[i][j] + " ") ;
     }
     System.out.println( "" ) ;
    }
  }
}
```

Fig. 1. A working example to illustrate the kind of annotations we added to all 98 of Java code examples covering all CS1 topics.

5 Discussion and Conclusions

The resulting DeepCode instructional codeset consists of 98 annotated Java code examples (at least 4 code examples per topic). More details about the codeset are shown in Table 1 in terms of total lines of code (with and without comments), total number of logical step comments and corresponding logical step details comments, total number of statement level comments, and the number of hints in the form questions.

Table 1. Descriptive statistics about the DeepCode codeset ($n = 98$ annotated Java code examples).

Metric	Total	Average
Total lines of code without comments	1631	16.81
Total lines of code with comments	7157	73.78
Logical steps	260	2.68
Logical step details	260	2.68
Statement level comments	408	4.20
Number of questions for logical steps	590	6.08

The design of the DeepCode instructional codeset was based on strong theoretical foundations which is a unique feature of it. The corresponding annotation guidelines offered as much details for the annotators as possible. Due to space reasons, we have not provided all the guidelines such as the need for the example authors to spellcheck all the comments. Furthermore, it should be noted that the guidelines are just that, guidelines. That is, they are not supposed to and in fact they cannot capture all possible cases that annotators may encounter during their code example creation and annotation.

Additionally, the guidelines leave some concepts vaguely defined, for instance, what exactly constitute a logical step. Nevertheless, we hope that the development and release of DeepCode will foster new developments in terms of additional resources and advanced educational technologies for deep code comprehension and learning of complex programming topics and ultimately help learners become successful computer professionals.

Acknowledgments. This work was supported by the National Science Foundation under award 1822816. All findings and opinions expressed or implied are solely the authors'.

References

1. Aleven, V., Koedinger, K.R.: An effective metacognitive strategy: learning by doing and explaining with a computer-based cognitive tutor. Cogn. Sci. **26**(2), 147–179 (2002)
2. Alhassan, R.: The effect of employing self-explanation strategy with worked examples on acquiring computer programming skills. J. Educ. Pract. **8**(6), 186–196 (2017)

3. Bielaczyc, K., Pirolli, P.L., Brown, A.L.: Training in selfexplanation and self-regulation strategies: investigating the effects of knowledge acquisition activities on problem solving. Cogn. Instr. **13**(2), 221–252 (1995)
4. Bloom, B.S.: All Our Children Learning - A Primer for Parents, Teachers, and Other Educators. McGraw-Hill, New York (1981). ISBN 9780070061187
5. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001)
6. Brooks, R.: Towards a theory of the comprehension of computer programs. Int. J. Man Mach. Stud. **18**, 543–554 (1983)
7. Brusilovsky, P., Yudelson, M.: From WebEx to NavEx: interactive access to annotated program examples. Proc. IEEE **96**(6), 990–999 (2008)
8. Buse, R.P.L., Weimer, W.R.: A metric for software readability. In: International Symposium on Software Testing and Analysis, pp. 121–130 (2008)
9. Chen, B., Azad, S., Haldar, R., West, M., Zilles, C.: A validated scoring rubric for explain-in-plain-English questions. In: The 51st ACM Technical Symposium on Computer Science Education (SIGCSE 2020), Portland, OR, USA, 11–14 March 2020. ACM, New York (2020). 7 pages. https://doi.org/10.1145/3328778.3366879
10. Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P., Glaser, R.: Self-explanations: how students study and use examples in learning to solve problems. Cogn. Sci. **13**, 145–182 (1989)
11. Chi, M.T.H., DeLeeuw, N., Chiu, M.-H., LaVancher, C.: Eliciting self-explanations improves understanding. Cogn. Sci. **18**(3), 439–477 (1994)
12. Chi, M.T.H.: Self-explaining: the dual processes of generating inference and repairing mental models. In: Glaser, R. (ed.) Advances in Instructional Psychology: Educational Design and Cognitive Science, vol. 5, pp. 161–238. Lawrence Erlbaum Associates Publishers (2000)
13. Chi, M.T.H., Wylie, R.: The ICAP framework: linking cognitive engagement to active learning outcomes. Educ. Psychol. **49**, 219–243 (2014)
14. Chi, M.T.H., et al.: Translating the ICAP theory of cognitive engagement into practice. Cogn. Sci. **42**, 1777–1832 (2018)
15. Conati, C., VanLehn, K.: Further results from the evaluation of an intelligent computer tutor to coach self-explanation. In: Gauthier, G., Frasson, C., VanLehn, K. (eds.) ITS 2000. LNCS, vol. 1839, pp. 304–313. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45108-0_34
16. Détienne, F.: Software Design - Cognitive Aspects. Practitioner Series. Springer, London (2002). https://doi.org/10.1007/978-1-4471-0111-6
17. Edwards, S.H., Murali, K.P.: CodeWorkout: short programming exercises with built-in data collection. In: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2017), pp. 188–193. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3059009.3059055
18. Graesser, A.C., Singer, M., Trabasso, T.: Constructing inferences during narrative text comprehension. Psychol. Rev. **101**, 371–395 (1994)
19. Good, J.: Programming paradigms, information types and graphical representations: empirical investigations of novice program comprehension. Ph.D. thesis, University of Edinburgh (1999)
20. Guo, P.J.: Online Python tutor: embeddable web-based program visualization for cs education. In: Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE 2013), Denver, Colorado, USA, pp. 579–584. Association for Computing Machinery (2013)
21. Kintsch, W.: Learning from text. Cogn. Instr. **3**(2), 87–108 (1986)

22. Kumar, A.N.: Epplets: a tool for solving parsons puzzles. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE 2018), pp. 527–532. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3159450.3159576
23. Lane, H.C., VanLehn, K.: A dialogue-based tutoring system for beginning programming. In: Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS), pp. 449–454. AAAI Press (2004)
24. Lopez, M., Whalley, J., Robbins, P., Lister, R.: Relationships between reading, tracing and writing skills in introductory programming. In: Proceedings of the Fourth International Workshop on Computing Education Research, pp. 101–112. ACM (2008)
25. O'Brien, M.P.: Software comprehension – a review & research direction. Department of Computer Science & Information Systems University of Limerick, Ireland. Technical report (2003)
26. Pennington, N.: 1987. Comprehension strategies in programming. In: Soloway, E., Iyengar, S. (eds.) Empirical Studies of Programmers: Second Workshop, pp. 100–113. Ablex, Norwood (1987)
27. Recker, M.M., Pirolli, P.: A model of self-explanation strategies of instructional text and examples in the acquisition of programming skills (1990)
28. Rezel, E.S.: The effect of training subjects in self-explanation strategies on problem solving success in computer programming (2003)
29. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: a review and discussion. Comput. Sci. Educ. **13**(2), 137–172 (2003)
30. Roy, M., Chi, M.T.H.: The self-explanation principle in multimedia learning. In: The Cambridge Handbook of Multimedia Learning, pp. 271–286 (2005)
31. Sanders, K., et al.: The Canterbury QuestionBank: building a repository of multiple-choice CS1 and CS2 questions. In: Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports (ITiCSE -WGR 2013), pp. 33–52. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2543882.2543885
32. Rugaber, S.: The use of domain knowledge in program understanding. Ann. Softw. Eng. **9**(1–4), 143–192 (2000)
33. Rus, V., Sidney, D., Xiangen, H., Graesser, A.C.: Recent advances in conversational intelligent tutoring systems. AI Mag. **34**(3), 42–54 (2013)
34. Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.: An introduction to program comprehension for computer science educators. In: Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE, pp. 65–86 (2010). https://doi.org/10.1145/1971681.1971687
35. Shaft, T.M.: The role of application domain knowledge in computer program comprehension and enhancement. Unpublished Ph.D. thesis, Pennsylvania State University (1992)
36. Sharrock, R., Hamonic, E., Hiron, M., Carlier, S.: CODECAST: an innovative technology to facilitate teaching and learning computer programming in a C language online course. In: Proceedings of the Fourth ACM Conference on Learning @ Scale (L@S 2017), pp. 147–148. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3051457.3053970
37. Shneiderman, B., Mayer, R.: Syntactic/semantic interactions in programmer behaviour. Int. J. Comput. Inf. Sci. **8**(3), 219–238 (1979)
38. Sirkiä, T.: Creating and tailoring program animations for computing education. J. Softw. Evol. Process **30**(2) (2018)
39. Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., Padua-Perez, N.: Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In: ITiCSE 2006, pp. 13–17 (2006)

40. Soloway, E., Spohrer, J.C.: Studying the Novice Programmer. Lawrence Erlbaum Associates, Hillsdale (1989)
41. Sweller, J., VanMerrienboer, J.J.G., Paas, F.: Cognitive architecture and instructional design. Educ. Psychol. Rev. **10**, 251 (1998). https://doi.org/10.1023/a:1022193728205
42. Whalley, J., et al.: An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In: Eighth Australasian Computing Education Conference (ACE 2006), January 2006
43. Woolf, B.P.: Building Intelligent Interactive Tutors: Student-Centered Strategies for Revolutionizing E-learning. Morgan Kaufman Publishers, Burlington (2009)
44. Zwaan, R.A., Radvansky, G.A.: Situation models in language comprehension and memory. Psychol. Bull. **123**(2), 162 (1998)
45. VanLehn, K.: The behavior of tutoring systems. Int. J. Artif. Intell. Educ. **16**(3), 227–265 (2006)