Decoding Source Code Comprehension: Bottlenecks Experienced by Senior Computer Science Students

Pakiso J. Khomokhoana and Liezel Nel(✉)

Department of Computer Science and Informatics, University of the Free State, Bloemfontein, South Africa
nell@ufs.ac.za

Abstract. Source code comprehension (SCC) continues to be a challenge to undergraduate CS students. Understanding the mental processes that students follow while comprehending source code can be crucial in helping students to overcome related challenges. The 'Decoding the Disciplines' (DtDs) paradigm that is gaining popularity world-wide provides a process to help students to master the mental actions they need to be successful in a specific discipline. In focusing on the first DtDs step of identifying mental obstacles ("bottlenecks"), this paper describes a study aimed at uncovering the major SCC bottlenecks that senior CS students experienced. We followed an integrated methodological approach where data were collected by asking questions, observations, and artefact analysis. Thematic analysis of the collected data revealed a series of SCC difficulties specifically related to arrays, programming logic, and control structures. The identified difficulties, including findings from the literature as well as our own teaching experiences, were used to compile a usable list of SCC bottlenecks. By focusing on senior students (instead of first-year students), the identified SCC bottlenecks point to learning difficulties that need to be addressed in introductory CS courses.

Keywords: Computer programming · Source code comprehension · Students' learning bottlenecks · Decoding the Disciplines

1 Introduction

Despite the continuous efforts of committed instructors to share the intricacies of their academic disciplines and their students' desperation to succeed, many students still struggle to master course material [32]. The specific points where students' learning gets interrupted can be referred to as *bottlenecks* [11,28]. A bottleneck typically occurs when students are unsure about how to approach a problem and consequently follow inappropriate strategies [32]. In an attempt to assist instructors in addressing students' learning bottlenecks, Middendorf and Pace devised the *Decoding the Disciplines* (DtDs) paradigm [28]. One of the underlying principles of this paradigm is that each discipline has unique

B. Tait et al. (Eds.): SACLA 2019, CCIS 1136, pp. 17–32, 2020.
https://doi.org/10.1007/978-3-030-35629-3_2

ways of thinking [28]. Students who fail to master the required 'ways of thinking' are unlikely to succeed in their higher-level studies. In the DtDs paradigm, instructors are therefore encouraged to identify discipline-specific learning bottlenecks that could prevent students from mastering the basic disciplinary ways of thinking. Subsequently, specific strategies to address the bottlenecks are identified, implemented and evaluated [32]. Despite the recent uptake in decoding research in other disciplines [40,43], limited information about DtDs research in Computer Science (CS) is available.

However, during the past three decades numerous investigations have been launched to gain better understanding of the various difficulties that computer programming students experience [3,4]. One such difficulty—which has been studied extensively—relates to the way in which students (also referred to as 'novice programmers') interpret pieces of source code [9,23]. This activity—commonly referred to as *source code comprehension* (SCC)—is regarded a vital skill that novice programmers have to master [39]. Most of the previous SCC studies, however, focused on the evaluation of difficulties that students enrolled for introductory programming courses experience [26,38]. Pace points out that a student's inability to master certain basic concepts may not necessarily lead to his/her failure of an introductory course [32]. However, it is likely that such a student's confusions will continue to accumulate, thus causing diminishing performance of basic tasks. As such, it is possible for students to progress to advanced courses while they are still experiencing bottlenecks related to basic concepts. Their failure to grasp these basic concepts can have a negative impact on their ability to complete their degrees. This paper therefore attempts to answer the following two *questions:*

1. What are the major SCC difficulties experienced by senior CS students**?**
2. How can knowledge of these difficulties be used to identify SCC bottlenecks that should ideally be addressed in introductory programming courses**?**

In the remainder of this paper, a review of relevant background literature is presented in Sect. 2. This is followed by a discussion of the research design and method in Sect. 3, and a presentation and interpretation of the results in Sect. 4. The identified SCC bottlenecks are presented in Sect. 5, and conclusions and recommendations for future research in Sect. 6.

2 Related Work

The first step of the seven-step DtDs framework [28] is to identify students' learning bottlenecks. The identification of discipline-specific bottlenecks allows instructors to identify specific areas in a module where they need to intervene more strongly in order to facilitate better learning [29,32]. In identifying a learning bottleneck, the instructor must ensure that the bottleneck is 'useful'. A bottleneck is 'useful' if it affects the learning of *many* students, interferes with the major learning in a module, is relatively focused, and does not involve a large number of disparate operations. It must also be defined clearly without jargon [32]. In the DtDs paradigm [28], instructors can take various ways to identify bottlenecks.

2.1 Bottleneck Identification Approaches

In one of the popular approaches [29], instructors themselves identify bottlenecks based on specific student problems they discover during their teaching of a specific module [34]. Instructors can also identify bottlenecks by focusing on a single assignment. In History, for example, Pace identified a specific difficulty while grading a writing assignment [32], whereas Shopkow was alerted to a specific difficulty as a result of questions voiced by her students regarding the specifications of an assignment [40].

In most of the limited number of decoding studies in the CS discipline to date, researchers have also identified specific bottlenecks based on personal teaching experiences. For a Database Design and Data Retrieval module, the authors of [19] identified creating Entity Relationship diagrams, reasoning in MySQL and dualism as the main student learning bottlenecks. Menzel used her experience in teaching an introductory CS module to identify recursion —a threshold concept in CS [38]— as her students' main bottleneck [27]. For a follow-up module, German focused his decoding study on the challenges of program debugging [14].

Bottleneck identification for a specific module can also be facilitated by an outsider (e.g. a pedagogical advisor). In [43], module-specific bottlenecks were identified by asking seven participants representing five disciplines (Engineering, Chemistry, History, Social Sciences and Electronics) to each write a 10-line description of two or three bottlenecks they could think of for the modules they were teaching. In an attempt to identify the top bottlenecks experienced by Accounting students in their Taxation modules, Timmermans and Barnett first asked instructors to identify potential bottlenecks [42]. Their eventual selection of the top bottlenecks was based on the responses of 4th-year Taxation students who were asked to rate the 40 potential bottlenecks w.r.t. level of understanding and importance.

When the goal is to identify common bottlenecks in a specific discipline, the collective experiences of a group of instructors can also be a valuable source. In this regard, various researchers from the History discipline have used individual interviews with instructors to identify common discipline-specific bottlenecks [11,41]. Wilkinson chose a peer dialogue strategy where Law instructors collectively established that the reading of case law was their students' major learning bottleneck [46]. For bottleneck identification in Political Science, Rouse (et al.) based their selection of literature reviews as the major bottleneck on the experiences of both instructors and students (from different year levels) as well as on the findings of other studies [37]. Thus an instructor's insight often is the main source for bottleneck identification. However, the role of students in bottleneck identification should not be ignored. Further justification for the seriousness of specific bottlenecks can also be found by linking bottlenecks to discipline-specific learning difficulties identified in other non-decoding studies (such as [31]).

2.2 SCC Difficulties

As mentioned above, numerous previous studies have attempted to uncover the specific difficulties experienced by novice programmers in comprehending source

code. Although none of these studies were specifically conducted in the DtDs framework, Middendorf and Shopkow suggest that relevant literature can also be used to identify bottlenecks [29].

In an investigation of the programming competency of students enrolled for CS1 and CS2 courses, McCracken (et al.) stated that many students still do not know how to program at the end of their introductory programming courses [26]. This problem was further explored in the BRACElet project which confirmed students' lack of programming skills [45]. In an attempt to expand understanding of the difficulties experienced by students, [26] refers to the potential role that in-depth analysis of narrative data collected from students can play.

The ITiCSE 2004 working group study [23] was conducted as a follow-up on the McCracken (et al.) study. A set of 12 multiple-choice questions (MCQs) was used to test students' ability on two tasks: firstly, to predict the output of executing the given fragments of source code; secondly, to select a piece of source code (from a small set of options) that would correctly complete a given near-complete code snippet. Although many students were found to be lacking the skills required to do both tasks, the latter was found to be the most challenging. The final ITiCSE 2004 report states that students were unable to *"reliably work their way through the long chain of reasoning required to hand-execute code, and/or... to reason reliably at a more abstract level to select the missing line of code"* [23] (p. 132).

All 12 questions used in [23] focused strongly on the concept of *arrays*. In a study aimed at improving students' learning experiences, Hyland and Clynch found arrays to be the most challenging topic for first- and second-year students [18]. In an attempt to record all the difficulties that students experience during practical computer programming sessions, Garner (et al.) found arrays to be featuring among the top three difficulties [13]. Other studies, too, have identified arrays as a challenging concept for novice programmers [2,24].

All ITiCSE 2004 questions [23] included basic *control structures* such as conditionals (e.g. `if`, `if-else`), loops (e.g. `while`, `for`), or a combination thereof. According to [30], many novice programmers struggle to comprehend basic control structures. Various studies have described the specific difficulties that students experienced while interpreting looping (repetition) structures [6,16,18,24]. Garner (et al.) mention that most of the difficulties associated with loops originate in students' incorrect comprehension of either the header or body of the looping structure [13].

Although logic generally is regarded as a mathematical field, it has grown more relevant to CS especially w.r.t. its applications [17]. Programming logic involves executing statements contained in a given piece of code one after another in the order in which they are written. Though still logical and correct, there are some programming control structures that may violate this execution order [10]. It is therefore not surprising that students struggle with logical reasoning in solving computer programming related problems [6]. The logical flow of the source code statements is closely related to the control flow of such statements [13]. This implies that for a programmer to fully comprehend a computer program, he/she must skilfully combine the programming logic with the control flow

of the program. Students are more likely to logically work (or trace) through a piece of source code if they have adequate knowledge of the semantics of the programming language and can keep track of changes made to variable values [23]. Therefore novices especially struggle to follow a program's execution [4,36] and control flow [13].

As the proponents of the DtDs paradigm argue that bottlenecks directly relate to difficulties hindering *many* students' learning [28], the previously identified difficulties can serve as a baseline for the identification of common and useful SCC bottlenecks. The exact nature of some of these difficulties, however, remains unclear: Where exactly are students getting stuck? Why are they getting stuck? What are they doing wrong? Which strategies do they resort to when they get stuck? Better knowledge about the nature of these difficulties can thus be valuable in determining teaching and learning gaps related to SCC.

3 Research Methods

3.1 Design

Within the scope of a DtDs-based research design, we followed an approach based on Plowright's Frameworks for an Integrated Methodology (*FraIM*) [35]. Thereby, our focus was on collecting narrative and/or numeric data by means of observations, asking questions, and/or artefact analysis. The study population consisted of final-year undergraduate CS students (referred to as 'senior students' in this paper) from a South African university. The empirical part of our study comprised two phases. The aim of Phase 1 was to identify specific senior CS students having trouble in comprehending short pieces of source code. In Phase 2 we wanted to detect specific points or 'places' [28] where these students were experiencing SCC difficulties, with the goal of identifying common and useful SCC bottlenecks.

3.2 Phase 1

Participants, Data Collection and Analysis. The sample for Phase 1 consisted of 40 students registered for a 3rd-year Internet Programming module. The selection of this sample was both 'purposeful' and 'convenient' [33]. The sample was purposeful because the students had already completed four earlier programming modules. However, they could still be regarded as 'novice' programmers since they did not yet have any professional programming experience. The sample was also convenient since we had easy access to the participants because the lecturer responsible for this module agreed to make available one of her scheduled class sessions for our research activity.

For the research activity of Phase 1, participants were given a test consisting of the 12 questions of [23]. For each of these questions, participants had to work through a short fragment of source code, and then either predict the execution output of the code fragment or select (from a small set of options) the relevant piece of code needed to complete the given fragment. The questions of [23]

were chosen for two reasons: Firstly, all of them contained source code fragments that students had to comprehend before they could answer the related question. Secondly, these questions had already been tested with a large population of students from several universities in the USA and other countries.[1] Since the original questions were formulated in Java, we had to convert their code fragments to C# because this is the programming language familiar to our chosen population.

The participants' answer sheets (regarded as 'artefacts') were our primary data source for Phase 1. After 'grading' the artefacts, the performance data for each participant were captured into a spreadsheet, and descriptive statistics were used to rank the questions in order of difficulty (based on the number of participants who incorrectly answered such question). The three apparently most difficult questions (Q3, Q6, Q8) were chosen for further use in Phase 2.

3.3 Phase 2

Data Collection. Based on the student performance data collected in Phase 1, 15 students were invited to take part in Phase 2. These were the students who provided wrong answers to all three of the questions identified in Phase 1. Ten of the 15 invited students agreed to partake in Phase 2. The research activity in Phase 2 consisted of individual sessions during which each participant had to verbally expose his/her thinking process(es) in a form of 'thinking aloud' [7] while answering anew the same three above-mentioned SCC questions. This data collection strategy can be regarded as a means of 'asking questions'.

Time slots of 45 min were scheduled for each individual session. However, the participants were informed that they could take as much time as they needed to complete the tasks. Since none of the participants had prior experience with the required think-aloud technique, this technique was first demonstrated to each participant on an unrelated SCC question. Thereby, one of us played the role of the 'interviewer' by asking probing questions when required (i.e. no progress or silence). Where necessary we also recorded some observations, as an additional means of data collection, after the permission for audio-recording was obtained by the corresponding participant.

Data Analysis. To transcribe and analyse the audio recordings from the individual think-aloud sessions, we followed the approach of [8]. Upon data transcription we 'cleansed' the data by searching for faults and by repairing them accordingly [47]. Since the participants had to verbalise their thoughts as part of the think-aloud process, the transcripts contained numerous illogical and repeated statements. We therefore decided to use 'fuzzy validation' (instead of strict validation which requires the complete removal of invalid or undesired responses) [47]. In fuzzy validation we are allowed to correct some data if there is a reasonably 'close match' to a known right answer. Thereafter we familiarised ourselves with the data [25] by listening and re-listening to the audio records numerous

[1] *Benchmark* for international comparability.

times as well as by intensively and repeatedly reading the transcripts. This helped us to devise a coding plan in which the analysis would be guided by the data related to our research questions. At this stage, the 10 validated transcripts were imported into the Nvivo 12 Professional tool. Thereafter, codes were developed (by creating several nodes) for each SCC difficulty identified in the data.

For coding, Klenke recommends the use of 'units of analysis' [22]. These can be words, sentences or paragraphs. Accordingly, we coded the data by highlighting and/or underlining text (from which the SCC difficulties could be extracted) within the domain of the stated units of analysis. Then we 'populated' the created codes by associating the corresponding texts with them. During this process of refinement, the names of the codes were continuously revised until *relevant themes* began to emerge. For each emerging theme, its Nvivo-generated frequency of occurrence was taken into account.

4 Results and Interpretation

Given the large amount of data collected during Phase 2, the results discussion only focuses on the participants' comprehension of Question 3 (Fig. 1, with line numbers **1**–**12** inserted in aid of this discussion). This question was selected since its related 'think-aloud' data revealed most clearly the numerous difficulties that can be directly associated with SCC. Q3 also tested students' comprehension of arrays and basic control structures—i.e. the concepts previously identified as challenging for novice programmers (see above). In the following discussion, the eight most common SCC difficulties identified are grouped into *three categories:* arrays, programming logic, and control structures.

4.1 Array-Related Difficulties

Analysis of the Q3 think-aloud data revealed the following *four* major array-related difficulties experienced by the participants.

Array Index. An array index refers to a non-negative integer number that identifies the position of an element stored in an array. Four participants had difficulties to interpret simple array indices, with a total of nine occurrences identified. Participant 1 (P1) had the most difficulties in this regard, with three occurrences identified. In her interpretation of `b[i]`, she regarded `i` as a value contained *in* array `b` instead of a position *of* an element. Participant P8 confused the square brackets indicating the array index with a multiplication operator when he interpreted `b[i]` as `b` multiplied by `i`: "*`int i` is equal to `0`* [Line 8], *and then for* ***this times that****, it is equal to* ***`true`*** [Line 10] *then increment the* ***`counter`*** [Line 11], ***that times that*** *is equal to* ***`true`*** *... it is a difficult one but then ...* ***that times that*** *is* ***`true`*** *and* ***that times that*** is ***`true`***". Thus, both participants were challenged by the *notation* [20] of the array index.

Array Length. The length of an array refers to the maximum number of values that can be stored in it. Three participants struggled to determine the length of

Question 3
Consider the following source code fragment:

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

3.  for (int i = 0; i < b.Length; ++i)
4.      b[i] = false;

5.  for (int i = 0; i < x.Length; ++i)
6.      b[x[i]] = true;

7.  int count = 0;

8.  for (int i = 0; i < b.Length; ++i)
9.  {
10.     if (b[i] == true)
11.         ++count;
12. }
```

After this source code is executed, `count` contains:

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

Fig. 1. Question 3 according to [23]

the arrays in Q3. P1 had no idea how to determine the length of the Boolean array `b` and remarked: *"I do not know what is the length of array `b`"*. Also P6 was unable to correctly determine the length of the array. He interpreted the Boolean array `b` to have the length of 4 (instead of the correct length 5): *"So now is `0` less than `4` because our `b` value is `4`"* while looking at the condition of the `for` loop in [Line 3].

Boolean Array. A Boolean array is an array of which the elements can only contain the values *true* or *false*. Five occurrences of Boolean array difficulties were identified, whereby P7 was most challenged (with three identified occurrences). Overall, the identified difficulties ranged from the declaration of the Boolean array to basic understanding regarding the effects of operations performed on such arrays.

P7 got stuck at the Boolean array declaration in [Line 2] and skipped the question: *"Do I understand what I am doing? ...it is a Boolean array, array is a Boolean, what does it mean? ...* (pause) *... I am not sure about this one yet, let me..."* (turning the page to see the next question). When P7 returned to this question later, his confusion regarding Boolean arrays became even more apparent as he regarded the index value of `1` as the Boolean equivalent of `true`: *"Once it gets to the `if` statement, `i` is now equal to `1` and `1` is equal to `true`"* [Line 10].

P9 was under the impression that since `b` was a Boolean array it could only consist of two elements (instead of two possible values per element): *"In position `0`, I have `1`, which means now at `b[1]` I have `true`. In my bool array I have stored two values"* [Line 10].

In their comprehension of [Line 10], both P7 and P9 disregarded the actual code *syntax*. Instead, they fell back to some *semantic* according to which a `0` represents `false` and a `1` represents `true`. Both participants believed the numeric index positions `0` and `1` to represent their Boolean truth value equivalents.

Decomposition—whereby a complex piece of code is 'split' into its constituent components to simplify its interpretation [21]—is a task with which many novice programmers struggle [15]. In their comprehension of Q3, seven participants found it particularly difficult to decompose the compound index in the expression `b[x[i]]` of [Line 6]. Altogether 29 occurrences of this difficulty were identified.

P10 misinterpreted [Line 6] as 'resetting' *all* values in `b` to `true`, whereas de-facto only the selected values in array `b` are set to `true`: *"`b[x[i]]` set to true* [Line 6]*... yes, no, I am very confused ...*(longer pause)*... `b[i]` ...then the second `for` loop* [Line 5] *sets everything from the integer array to `true`, so, if I am correct, then it resets everything from the first `for` loop* [Line 3] *back to `true`"*.

P6 became so confused with the meaning of the compound index expression that he could not even grasp how the code in [Line 6] was related to the `for` loop in [Line 5]: *"Now I am worried about this `for` loop, the second `for` loop* [Line 5]*, it seems like it has nothing to do with the rest of the statements that come after it... so this second `for` loop is the one that is freaking me out"*. Although P6 had no difficulty to understand any of the other `for` loops in Q3, it seems that his inability to decompose the compound index expression caused so much confusion that he suddenly could not comprehend the basic execution of the `for` loop in [Line 5].

4.2 Programming Logic Difficulties

The following *three* programming logic difficulties were identified from our Q3-related think-aloud transcripts.

Ripple Effect. This effect occurs when the misinterpretation of one statement has a direct impact on the interpretation of statements that follow. This difficulty, which was observed with three participants, typically arises when programmers misinterpret programming logic [20]. Due to P1's failure in interpreting array indices (see above), her interpretation of the statements contained in the third `for` loop completely ignored any changes made to the elements of `b` in the first two `for` loops [Lines 3–6]. She remarked: *"If `b[i]` is `true`* [Line 10]*, I increment `count`* [Line 11]*. So if I increment `count` every time until it is over `5`, then I will have 5"*. Thus she wrongly chose '5' (option **e**) as her answer to Q3.

The difficulties that P6 had in interpreting the second `for` loop [Lines 5–6] (see above: Decomposition) caused him to disregard that loop entirely while interpreting the third `for` loop: *"When looking at this third `for` loop* [Line 8]*, it is the same as the first one* [Line 3] *that says the `bool` array is always equal to `false`. Now in the third one they are saying if the element at position `i` in the Boolean array is equal to `true`* [Line 10]*, then increment `count`* [Line 11]*. But according to this* [Line 4]*, the `b` value is always `false`*".

The utterances by both P1 and P6 indicated that they were not thinking sequentially [5], and therefore failed to follow the algorithmic logic of the source code in question [1]. P9 showed similar behaviour after she realised that she could not interpret any of the `for` loops and the containing statements. In response, she reverted her attention to those statements that she could comprehend and only considered those to arrive at `count=1` as her answer to Q3. Her non-sequential (non-algorithmic) reasoning appears in the following excerpt: *"My first index: I have a `false`* [Line 4]*, and then my second: I have a `true`* [Line 6]*, and then `int count` is equal to `0`* [Line 7] *...it will only increment when I get to this point* [Line 11]*, whereby `count` needs to be `1`*" (option **a**).

The most concerning aspect of the thinking patterns portrayed by these participants is the 'mental block' caused by the statements they could not fully comprehend, and their consequent anxious behaviour (observed by the interviewer). These participants tried to 'escape' the block by entirely ignoring the troublesome statements *as if* those were no longer part of the given code.[2]

Guessing. A common critique of MCQs is that they are solvable through guessing. This is also true of our 12 MCQs (Part 1) for which guessing behavior was previously observed [12,23]. However our format of think-aloud sessions (Phase 2) discouraged guessing, because the participants were repeatedly prompted to explain their reasoning in as much detail as possible. Nonetheless, one participant (P8) attempted guessing when saying *"I just have to go with option **a***" after having traced only a small fraction of the given code. At that stage he was unable to show how he arrived at the chosen answer and had to be prompted by the interviewer to re-explain his reasoning.

Mathematical Expressions. When a line of code contains a mathematical expression, the misinterpretation of an operator can interfer with the comprehension of program logic; for comparison see [31]. One example of such a mistake was observed when P7 failed to grasp the execution of the third `for` loop [Line 8] when the value of `i` increased to 5: *"Yes, `i` becomes `5`... once it runs throughout the loop and becomes `5` then... `b[i]` is going to be `true`... then the `count` also increments"*. He thus treated the $<$ as if it was a $\leq$ operator, which is typically regarded as a 'logical error' in the comprehension of source code.

[2] This observation points to the *psychology* of programming which is a field of research since the late 1960s/early 1970s [44].

4.3 Programming Control Structure Difficulty

The code in Q3 only contained one type of control structure in the form of three `for` repetition structures. As mentioned above (Ripple Effect), the lack of understanding that P6 and P9 portrayed regarding the overall functioning of a `for` loop caused them to eventually ignore the lines of code related to these structures. Another `for` loop misconception was observed when P7 repeatedly executed the loop counter increment statement `(++i)` at the beginning of each loop, thereby setting the initial value of `i` to `1` for each of the three loops. Since repetition structures are one of the concepts that novices find challenging [16], it is not surprising that some participants experienced difficulties in this regard. However, one area of concern is the level of difficulty that these *senior* students experienced in comprehending *basic* `for` repetition structures.

5 Identification of Six SCC Bottlenecks

The results of Phase 2 showed that the participants in this study experienced eight major SCC difficulties related to the concept of arrays, programming logic, and programming control. Following the bottleneck identification guidelines of [29,32], we used our collective experience of more than 25 years of teaching introductory and advanced programming courses, combined with the insights gained from this study as well as relevant literature, to formulate the following six 'usable' SCC bottlenecks.

5.1 Bottleneck 1

Students cannot keep track of variable values while tracing through a piece of code. The above-mentioned think-aloud excerpts contain numerous examples in which students lost track of the changes made to variable values. The students tried to remember the changes to the variable values (instead of writing notes on a provided piece of paper), which put unnecessary strain on their working memories. Their incorrect answers were thus a direct consequence of failing memory or guessing. Lister (et al.) pointed out that, when students document changes to variable values, they are much more likely to produce the correct answer [23]. Most students in our study did not follow a reliable strategy to keep track of such value changes.

5.2 Bottleneck 2

Students cannot comprehend statements containing arrays and basic operations on array elements. The bulk of the identified difficulties can be related to the students' wrong understanding of array concepts: for comparison see [2,13,18,24]. Our students particularly struggled to interpret array indices—especially in combination with other concepts. While one student confused the square [index] brackets with a multiplication operator, others were not able to

determine an array's length. Although most students had little trouble to comprehend the array of numeric values, many of them were lost when having to deal with the Boolean array type.

5.3 Bottleneck 3

Students cannot comprehend the execution of basic `for` repetition structures. Most of the difficulties observed with the `for` loops are due to our students' incorrect comprehension of either the header or the body of the looping structure (as in [13]). While some students failed to recognise when and how a loop terminates (see also [16]), one instance was observed where the loop counter increment statement was executed at the wrong time. Although most of the difficulties observed in comprehension of the body of the looping structure are more specifically related to arrays, referencing the incorrect value of the loop counter variable also caused problems for some students. Most worrying were the two students who entirely gave up interpreting the `for` loops and ignored either the entire structure or the loop header for the remainder of their Q3 interpretation.

5.4 Bottleneck 4

Students do not have adequate strategies to interpret lines of code that look unfamiliar. This bottleneck was observed in cases where students were unable to read, interpret and understand (execute) a specific code statement. Of particular interest here are cases where two or more separate concepts—which a student could comprehend earlier—were combined to form one 'complex' concept. The students were unable to decompose the more complex piece of code into smaller parts in order to simplify the interpretation thereof, (see also [21]). Their most common response to this challenge was to ignore those complex statements or lines of code. Although decomposition is a task that many novice programmers struggle with [15], students may never learn how to deal with complex concepts if they are not taught explicit strategies to resort to in such situations.

5.5 Bottleneck 5

Students view a piece of source code as consisting of separate lines of code, thereby ignoring the significance of each individual line. We typically teach our students that, in order to fully comprehend what a program does, they first need to understand the meaning of each distinct line of code of that program. However, it seems that in following our 'guidelines' some students not only lose sight of how the parts fit together but also of the overall significance of each individual line of code or statement. This behaviour was evident with those students who completely ignored sections of code they could not comprehend, with a complete disregard for the impact this would have on their ability to

determine the correct answer to Q3. For comparison see the *'ignore significance'* bottleneck in [41] about History students' disregard of how individual facts relate to the story they are trying to tell.

5.6 Bottleneck 6

Students cannot reliably think their way through a long chain of reasoning required to comprehend a piece of source code. This bottleneck can be regarded as 'overarching' since it refers to one of the most common and significant SCC difficulties identified in [23]. In our study it is directly related to our *'ripple effect'* that refers to mistakes made when students are unable to think sequentially [5] or fail to follow the source code logic [1]. In our study, we noticed the significant negative impact that inadequate knowledge of semantics and inability to keep track of variable values can have on a student's comprehension of a piece of code. These are all examples of actions that can cause a 'mental block' in students' reasoning ability, which they are unlikely to overcome if they do not have the necessary knowledge and abilities to deal with such difficulties.

Although we present these as six separate bottlenecks, they should be seen as *"interconnected with each other"* [41], since they are all indicators of mental challenges experienced by novice programmers while comprehending source code.

6 Conclusions and Future Work

SCC continues to be a challenge to undergraduate CS students. Understanding the mental processes that students follow while comprehending source code can be crucial in helping students to overcome related challenges. By focusing on Step 1 of the seven-step DtDs framework, this study aimed to expose the major SCC bottlenecks experienced by senior CS students. Thematic analysis of data collected by means of asking questions, observations, and artefact analysis revealed several SCC difficulties specifically related to arrays, programming logic and control structures. The detected difficulties, combined with findings from the literature and our personal experiences, were used to formulate six bottlenecks that are indicative of the typical mental challenges experienced by novice programmers during the comprehension of source code. By focusing on senior students we were able to identify major bottlenecks that point to students' learning difficulties that are currently not adequately addressed in introductory CS courses, and therefore continue to influence the mental processes of final-year undergraduate students.

With this paper we also wanted to raise awareness among instructors regarding the role that a systematic 'decoding' approach can play in exposing the mental processes and bottlenecks unique to the CS discipline. To address the remaining six steps of the DtDs framework [28], future research is needed, firstly, to uncover the mental activities of *expert* programmers to overcome the identified SCC bottlenecks. This knowledge could then be used to devise teaching and

learning strategies that model the mental strategies of the experts. After creating opportunities for students to practice these skills and to receive feedback on their efforts, instructors can assess students' efforts to determine whether they have benefited from the implemented strategies or not. The ultimate goal of this suggested research is to help students to master the mental actions they need to be successful in the CS discipline.

References

1. Alston, P., Walsh, D., Westhead, G.: Uncovering 'threshold concepts' in web development: an instructor perspective. ACM Trans. Comput. Educ. **15**(1), 1–18 (2015)
2. Anyango, J.T., Suleman, H.: Teaching programming in Kenya and South Africa: what is difficult and is it universal? In: Proceedings of the 18th Koli Calling International Conference on Computing Education Research. ACM (2018)
3. Bosse, Y., Gerosa, M.A.: Difficulties of programming learning from the point of view of students and instructors. EEE Lat. Am. Trans. **15**(11), 2191–2199 (2017)
4. du Boulay, B.: Some difficulties of learning to program. J. Educ. Comput. Res. **2**(1), 57–73 (1986)
5. Boustedt, J., et al.: Threshold concepts in computer science: do they exist and are they useful? In: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, pp. 504–508. ACM (2007)
6. Butler, M., Morgan, M.: Learning challenges faced by novice programming students studying high level and low feedback concepts. In: Proceedings Ascilite Singapore, pp. 99–107 (2007)
7. Charters, E.: The use of think-aloud methods in qualitative research: an introduction to think-aloud methods. Brock Educ. **12**(2), 68–82 (2003)
8. Creswell, J.W., Creswell, J.D.: Research Design: Qualitative, Quantitative, and Mixed Methods Approaches. SAGE, Thousand Oaks (2017)
9. Cunningham, K., Blanchard, S., Ericson, B., Guzdial, M.: Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In: Proceedings of the International Conference on Computing Education Research, pp. 164–172. ACM (2017)
10. Deitel, P.J., Deitel, H., Deitel, A.: Visual Basic 2012 — How to Program. Pearson Education, London (2013)
11. Diaz, A., Middendorf, J., Pace, D., Shopkow, L.: The history learning project: a department 'decodes' its students. J. Am. Hist. **94**(4), 1211–1224 (2008)
12. Fitzgerald, S., Simon, B., Thomas, L.: Strategies that students use to trace code: an analysis based in grounded theory. In: Proceedings of the 1st International Workshop on Computing Education Research, pp. 69–80. ACM (2004)
13. Garner, S., Haden, P., Robins, A.: My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In: Australasian Computing Education Conference, pp. 173–180. Australian Computer Society Inc., Newcastle (2005)
14. German, A., Menzel, S., Middendorf, J., Duncan, F.J.: How to decode student bottlenecks to learning in computer science. In: Proceedings of the 45th Technical Symposium on Computer Science Education, p. 733. ACM (2014)
15. Goldman, K., et al.: Identifying important and difficult concepts in introductory computing courses using a delphi process. In: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, pp. 256–260. ACM (2008)

16. Grover, S., Basu, S.: Measuring student learning in introductory block-based programming: examining misconceptions of loops, variables, and boolean logic. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education, pp. 267–272. ACM (2017)
17. Gurevich, Y.: Logic and the challenge of computer science. In: Börger, E. (ed.) Current Trends in Theoretical Computer Science, pp. 1–57. Computer Science Press (1988)
18. Hyland, E., Clynch, G.: Initial experiences gained and initiatives employed in the teaching of Java programming in the Institute of Technology Tallaght. In: Joint Proceedings of the Inaugural Conference on the Principles and Practice of Programming, and 2nd Workshop on Intermediate Representation engineering for Virtual Machines, pp. 101–106. ACM (2002)
19. IUBCITL: Team-Based Learning For Practice and Motivation (2016). https://www.youtube.com/watch?v=1obB-n6JZ8k
20. Kallia, M., Sentance, S.: Computing teachers' perspectives on threshold concepts: functions and procedural abstraction. In: Proceedings of the WIPSCE 12th Workshop on Primary and Secondary Computing Education, pp. 15–24 (2017)
21. Keen, A., Mammen, K.: Program decomposition and complexity in CS1. In: Proceedings of the 46th Technical Symposium on Computer Science Education, pp. 48–53. ACM (2015)
22. Klenke, K.: Qualitative Research in the Study of Leadership, 2nd edn. Emerald Publishing, Bingley (2016)
23. Lister, R., et al.: A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bull. **36**(4), 119–150 (2004)
24. Malik, S.I., Coldwell-Neilson, J.: A model for teaching an introductory programming course using ADRI. Educ. Inf. Technol. **22**(3), 1089–1120 (2017)
25. Marshall, C., Rossman, G.B.: Designing Qualitative Research, 6th edn. SAGE, Thousand Oaks (2016)
26. McCracken, M., et al.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education, pp. 125–180. ACM (2001)
27. Menzel, S.: ISSOTL 2015: Recursion as a Bottleneck Concept (2017). https://www.youtube.com/watch?v=iNvQlm9phEI
28. Middendorf, J., Pace, D.: Decoding the disciplines: a model for helping students learn disciplinary ways of thinking. New Dir. Teach. Learn. **98**, 1–12 (2004)
29. Middendorf, J., Shopkow, L.: Overcoming Student Learning Bottlenecks: Decode Your Disciplinary Critical Thinking. Stylus Publishing/LLC (2018)
30. Milne, I., Rowe, G.: Difficulties in learning and teaching programming: views of students and tutors. Educ. Inf. Technol. **7**(1), 55–66 (2002)
31. Mutanu, L., Machoka, P.: Enhancing computer students' academic performance through explanatory modeling. In: Tait, B., et al. (eds.) SACLA 2019. CCIS, vol. 1136, pp. 227–243 (2020)
32. Pace, D.: The Decoding the Disciplines Paradigm: Seven Steps to Increased Student Learning. Indiana University Press (2017)
33. Patton, M.Q.: Qualitative Research & Evaluation Methods: Integrating Theory and Practice, 4th edn. SAGE, Thousand Oaks (2015)
34. Pinnow, E.: Decoding the disciplines: an approach to scientific thinking. Psychol. Learn. Teach. **15**(1), 94–101 (2016)
35. Plowright, D.: Using Mixed Methods: Frameworks for an Integrated Methodology. SAGE, Thousand Oaks (2011)

36. Qian, Y., Lehman, J.: Students' misconceptions and other difficulties in introductory programming: a literature review. ACM Trans. Comput. Educ. **18**(1), 1–24 (2017)
37. Rouse, M., Phillips, J., Mehaffey, R., Mcgowan, S., Felten, P.: Decoding and disclosure in students-as-partners research: a case study of the political science literature review. Int. J. Stud. Partn. **1**(1), 1–14 (2017)
38. Sanders, K., McCartney, R.: Threshold concepts in computing: past, present, and future. In: Proceedings of the 16th International Conference on Computing Education Research, pp. 91–100. ACM (2016)
39. Shaft, T.M., Vessey, I.: The relevance of application domain knowledge: the case of computer program comprehension. Inf. Syst. Res. **6**(3), 286–299 (1995)
40. Shopkow, L.: How many sources do i need? Hist. Teach. **50**(2), 169–200 (2017)
41. Shopkow, L., Diaz, A., Middendorf, J., Pace, D.: From bottlenecks to epistemology in history: changing the conversation about the teaching of history in colleges and universities. In: Changing the Conversation about Higher Education. Rowman & Littlefield Publishing (2013)
42. Timmermans, J., Barnett, J.: The Role of Identifying and Decoding Bottlenecks in the Redesign of Tax Curriculum. In: Society for Teaching and Learning in Higher Education Conference, Canada (2013)
43. Verpoorten, D., et al.: Decoding the disciplines — a pilot study at the University of Liege (Belgium). In: Proceedings of 2nd EuroSoTL Conference, pp. 263–267 (2017)
44. Weinberg, G.M.: The Psychology of Computer Programming. Van Nostrand Reinhold/Litton Educational Publishing (1971)
45. Whalley, J.L., et al.: An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In: Proceedings of the 8th Australasian Conference on Computer Science Education, pp. 243–252 (2006)
46. Wilkinson, A.: Decoding learning in law: collaborative action towards the reshaping of university teaching and learning. Educ. Media Int. **51**(2), 124–134 (2014)
47. Willes, K.L.: Data cleaning. In: The SAGE Encyclopedia of Communication Research Methods. SAGE (2017)