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

**Full Paper**

**SACLA 2019**



Pakiso J. Khomokhoana [1][0000-0003-3642-1248] and Liezel Nel [2][0000-0002-6739-9285]

Department of Computer Science and Informatics, University of the Free State, Bloemfontein, South Africa
[1]khomo_khoana@yahoo.com, [2]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. Set within the Decoding the Disciplines paradigm, this paper reports on a study aimed at uncovering common SCC bottlenecks that senior CS students experienced. Thematic analysis of the collected data revealed eight common SCC difficulties specifically related to arrays, programming logic and control structures. The identified difficulties, together with findings from existing literature as well as the authors' personal experiences, were then used to formulate six usable SCC bottlenecks. The identified bottlenecks point to student learning difficulties that should be addressed in introductory CS courses. This paper intends to create awareness among CS instructors regarding the role that a systematic decoding approach can play in exposing the mental processes and bottlenecks unique to the CS discipline. Further investigations are needed to uncover the mental tasks that expert programmers follow to overcome the identified bottlenecks so that students can be taught more explicit SCC strategies.

**Keywords:** Undergraduate Programming, Source Code Comprehension, Student 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 [1]. The specific points where students' learning gets interrupted can be referred to as bottlenecks [2, 3]. A bottleneck typically occurs when students are unsure about how to approach a problem and consequently follow inappropriate strategies [1]. In an attempt to assist instructors in addressing student learning

bottlenecks, Middendorf and Pace [3] devised the Decoding the Disciplines (DtDs) paradigm. One of the underlying principles of this paradigm is that each discipline has unique ways of thinking [3]. Those students who fail to master the required ways of thinking are unlikely to succeed in their higher-level studies. Within 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 [1]. Despite the recent uptake in decoding research conducted in other disciplines [4, 5], limited information regarding DtDs research in the Computer Science (CS) discipline is available in the public domain.

However, over the past three decades numerous investigations have been launched to gain better understanding of the various difficulties that computer programming students experience [6, 7]. One such difficulty – which has been researched extensively – relates to the way in which students (also referred to as novice programmers) interpret pieces of source code [8, 9]. This action – commonly referred to as source code comprehension (SCC) – is regarded a vital skill that novice programmers have to master [10].

Most of the previous SCC studies, however, focused on the evaluation of difficulties that students enrolled for introductory programming courses experience [11, 12]. Pace [1] points out that a student's inability to master certain basic concepts may not necessarily lead to his/her failure of an introductory course. However, it is likely that the student's confusion will continue to pile up, 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 could potentially 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 Section 2. This is followed by a discussion of the research design and method in Section 3, and a presentation and interpretation of the results in Section 4. The paper concludes with a presentation of the identified SCC bottlenecks in Section 5, and conclusions and recommendations for future research in Section 6.

## 2 Background

The first step of Middendorf and Pace's [3] seven-step DtDs framework is to identify student learning bottlenecks. The identification of discipline-specific bottlenecks allows instructors to identify specific areas in a module where they need to seriously intervene in order to facilitate maximum learning [1, 13]. In identifying a learning bottleneck, the instructor must ensure that the bottleneck is useful. A useful bottleneck affects the learning of many students; is defined clearly and without jargon; interferes with the major learning in a module; is relatively focused; and does not involve a large

number of very disparate operations [1]. Within the DtDs paradigm, instructors can use various ways to identify bottlenecks.

### 2.1 Bottleneck Identification Approaches

In one of the popular approaches, as suggested by Middendorf and Shopkow [13], instructors themselves identify bottlenecks based on specific student problems they discover during their teaching of a specific module [14]. Instructors can also identify bottlenecks by focusing on a single assignment. In the History discipline, Pace [1] identified a specific difficulty while grading a writing assignment, while Shopkow [4] was alerted to a specific difficulty as a result of questions voiced by her students regarding the specifications of an assignment.

In most of the limited number of decoding studies conducted in the CS discipline to date, researchers have also identified specific bottlenecks based on personal teaching experiences. For his Database Design and Data Retrieval module, Dan Richert [15] identified creating Entity Relationship diagrams, reasoning in MySQL and dualism as the main student learning bottlenecks. At Indiana State University, Menzel [16] used her vast experience in teaching an introductory CS module to identify recursion (a threshold concept in CS [12]) as the main bottleneck that her students experienced. For a follow-up module, her colleague Adrian German [17] focused his decoding study on addressing the challenges his students experienced with debugging.

Bottleneck identification for a specific module can also be facilitated by an outsider (e.g. a pedagogical advisor). In Verpoorten et al.'s [5] study, module-specific bottlenecks were identified by asking seven participants, representing five disciplines (Engineering, Chemistry, History, Social Sciences and Electronics), to each write down a 10-line description of two or three bottlenecks they could think of for modules they were teaching. In an attempt to identify the top bottlenecks experienced by Accounting students in their Taxation modules, Timmermans and Barnett [18] first asked instructors to identify potential bottlenecks. Their eventual selection of the top bottlenecks was based on the responses of 4$^{th}$ year Taxation students who were asked to rate the 40 potential bottlenecks in terms of level of understanding and importance.

When the goal is to identify common bottlenecks within 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 [2, 19] have used individual interviews with instructors to identify common discipline-specific bottlenecks. Wilkinson [20] opted for a peer dialogue strategy where Law instructors collectively established that the reading of case law was the major learning bottleneck that their students experienced. For bottleneck identification in Political Science, Rouse et al. [21] 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 the findings of other research studies.

It is therefore apparent that an instructor's insight often is the main source used for bottleneck identification. However, the role that students can play in bottleneck identification should not be overlooked. 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.

### 2.2 SCC Difficulties

As mentioned in Section 1, numerous previous studies have attempted to uncover the specific difficulties experienced by novice programmers while comprehending source code. Although none of these studies were specifically conducted within the DtDs framework, Middendorf and Shopkow [13] suggest that relevant literature can also be used to identify bottlenecks.

Following an investigation of the programming competency of students enrolled for CS1 and CS2 courses, the 2001 McCracken group [11] concludes that many students still do not know how to program at the end of their introductory programming courses. The McCracken problem was further explored by the BRACElet project, which confirmed students' lack of programming skills as a reality [22]. In an attempt to further understanding of the difficulties experienced by students, the McCracken group [11] refers to the potential role that in-depth analysis of narrative data collected from students can play in creating deeper understanding of these difficulties.

The ITiCSE 2004 working group study [9] was conducted as a follow-up on the McCracken study. They used a set of 12 Multiple Choice Questions (MCQs) to test students' ability on two tasks: firstly, to predict the outcome of executing the given fragments of source code; and secondly, their ability 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 perform both tasks, the latter was found to be the most challenging. The final ITiCSE 2004 working group report concludes 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" [9] (p.132).

The questions that the ITiCSE 2004 working group [9] used focused heavily on the concept of arrays – with arrays featuring in all 12 questions. In a study aimed at improving students' learning experiences, Hyland and Clynch [23] found arrays to be the most challenging topic for first and second year students. In an attempt to record all the difficulties that students experience during practical computer programming sessions, Garner, Haden and Robins [24] found arrays to be featuring among the top three difficulties experienced by students. Other studies [25, 26] have also identified arrays as a challenging concept for novice programmers.

All the ITiCSE 2004 questions [9] included some form of basic control structures such as conditionals (e.g. `if`, `if-else`), loops (e.g. `while`, `for`) or a combination of both. According to Milne and Rowe [27], many novice programmers struggle to comprehend basic control structures. Various studies have reported on the specific difficulties that students experienced while interpreting looping (repetition) structures [23, 26, 28, 29]. Garner et al. [24] mention that most of the difficulties associated with loops originate in students' incorrect comprehension of either the header or body of the looping structure.

Although logic generally is regarded as a Mathematical field, it has grown more relevant to CS especially with regard to its applications [30]. 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 [31]. It is therefore not surprising that students struggle with logical reasoning in solving computer programming related problems [28]. The logical flow of the source code statements is closely related to the control flow of such statements [24]. 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 have the ability to keep track of changes made to variable values [9]. It is therefore especially novices who struggle to follow a program's execution [7, 32] and control flow [24].

As the proponents of the DtDs paradigm argue that bottlenecks directly relate to difficulties hindering the learning of many students [3], these 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 fuzzy: 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? More in-depth knowledge regarding the nature of these difficulties can be invaluable in determining teaching and learning gaps related to SCC.



# 3 Research Methods

## 3.1 Design

Within the realms of a DtDs-based research design, the study described in this paper followed an approach based on Plowright's [33] Frameworks for an Integrated Methodology (FraIM). Within this framework, the 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 from a selected South African university (referred to as 'senior students' in this paper). The empirical part of the 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 uncover specific points or places [3] 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 the 40 students registered for the 3rd year Internet Programming module. The selection of this sample can be described as both purposeful and convenient [34]. The sample was purposeful because the students had already completed four programming modules. However, they could still be regarded as novice

programmers since they did not have any professional programming experience. The sample was also convenient since we had easy access to the participants as the lecturer responsible for the module agreed to make available one of her scheduled class sessions for this research activity.

For the research activity of Phase 1, participants were given a test consisting of the 12 MCQs developed by the ITiCSE 2004 working group [9]. For each of the questions, participants had to work through a short fragment of source code and then either predict the execution outcome of the code fragment or select (from a small set of options) the relevant piece of code needed to complete the given fragment. These 12 MCQs were chosen for two reasons. Firstly, all the questions contained source code fragments that students had to comprehend before they could answer the related question. Secondly, the questions had been tested with a large population of students from several universities in the United States of America and in other countries. Since the original questions were written in Java, we had to convert the code fragments to C# (a programming language familiar to the chosen population).

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

### 3.3 Phase 2 Data Collection

Based on the student performance data collected during Phase 1, a total of 15 participants were invited to take part in Phase 2. These were the participants who provided incorrect answers to all three of the most difficult questions identified in Phase 1. Ten of the 15 invited participants agreed to partake in Phase 2. The research activity in Phase 2 consisted of individual sessions during which each participant had to verbally explain his/her thinking process(es) (through a think-aloud technique [35]) while answering the three most difficult SCC questions identified in Phase 1. This data collection strategy can be regarded as a means of "asking questions".

Time slots of 45 minutes were scheduled for each of the individual sessions. However, participants were informed that they could take as much time as they needed to complete the task. Since none of the participants had prior experience with the required think-aloud technique, this technique was demonstrated to each participant, using an unrelated SCC question. The first author (principal researcher) played the role of the interviewer by asking probing questions when required (i.e. no progress or silence). Where deemed necessary, he also recorded some observations as an additional data collection strategy. The proceedings of each session were audio recorded with permission from the relevant participant.

### 3.4 Phase 2 Data Analysis

To transcribe and analyse the audio recordings made during the individual think-aloud sessions, we used an adapted version of Haregu's [36] Narrative Data Analysis Framework. Upon data transcription, the principal researcher cleansed the data by searching for faults and repairing them [37]. Since the participants had to verbalise their thoughts as part of the think-aloud process, the transcripts contained numerous illogical and repeated statements. He therefore decided to make use of fuzzy validation instead of strict validation (which requires the complete removal of invalid or undesired responses) [37]. With fuzzy validation, the researcher is allowed to correct some data if there is a close match or known answer. After this, the principal researcher familiarised himself with the data [38] by listening and re-listening to the audio records numerous times as well as intensively and repeatedly reading the transcripts. This helped him to decide on a coding plan where the analysis would be guided by the data as it relates to the first research question. At this stage, the 10 validated transcripts were imported into Nvivo 12 Professional for Microsoft Windows, after which codes were developed (by creating several nodes) for each SCC difficulty identified in the data.

In coding, Klenke [39] recommends the use of a unit of analysis. These can be words, sentences or paragraphs. As such, the principal researcher coded the data by highlighting and/or underlining text (from which the SCC difficulties could be extracted) within the realms of the stated units of analysis. He then populated the created codes by moving the necessary text into them. During this process, the names of the codes were continuously revised. Relevant themes and recurrent themes then started emerging. For each theme developed, the Nvivo-generated frequencies of occurrence were used.

## 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 (see Fig. 1). (Note: The code line numbers were added in aid of this discussion). This question was selected since the related think-aloud activity data revealed numerous difficulties that can be directly associated with SCC. This question also tested students' comprehension of arrays and basic control structures – concepts that both have previously been identified as challenging for novice programmers (see Section 2.2). The discussion of the eight most common SCC difficulties identified is grouped into three categories: arrays, programming logic and control structures.

### 4.1 Array Related Difficulties

Analysis of the Question 3 think-aloud data revealed four major array related difficulties experienced by the participants.

**Array index**. An array index refers to a key or value that identifies the position of an element or object stored in an array. Four of the participants had difficulties to interpret

simple array indexes 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 recognising it as the position of the element in the array. One of the other participants (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`*".* From the given examples, it can be deduced that both participants were challenged by the notation [40] of the array index.

**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 from the set of 12 MCQs.

**Length of an array**. The length of an array refers to the maximum number of values that can be stored in a given array. Three of the participants struggled to determine the length of the arrays contained in Question 3. 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`". Similarly, P6 was unable to determine the correct length of the array. He interpreted the Boolean array `b` to have the length of 4 while the correct length was 5: *"So now is 0 less than 4 because our* `b` *value is 4"* [while reading the condition of the `for` loop in Line 3].

**Boolean array.** A Boolean array refers to an array where the elements can only contain `true` or `false` values. Five occurrences of Boolean array difficulties were identified, with P7 being the most challenged (with three identified occurrences). Overall, the identified difficulties ranged from 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 opted to skip 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 later returned to this question, 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].

Similarly, P9 was under the impression that since `b` was a Boolean array it could only contain two values: *"In position `0`, I have `1` which means now at `b[1]` I have `true`. In my bool array I have stored `2` values"* [Line 10]. In their comprehension of Line 10, both P7 and P9 disregarded the actual code syntax. Instead, they reverted back to their basic knowledge about Boolean variables where a `0` represents `false` and a `1` represents `true`. Both participants regarded the index positions of `0` and `1` to represent the Boolean equivalents.

**Decomposition**. Decomposition – where a complicated piece of code is broken down into its constituent components in order to simplify the interpretation thereof [41] – is a task that many novice programmers struggle with [42]. In their comprehension of Question 3, seven of the participants found it particularly difficult to decompose the compound index contained in the expression `b[x[i]]` (see Line 6 in Fig. 1). Overall, 29 occurrences of this difficulty were identified from the Question 3 transcripts.

P10 misinterpreted Line 6 to be resetting all the values contained in the `b` array to `true`, while in actual fact only the selected values in array `b` would be reset to `true`: "*`b[x[i]]` set to `true`* [Line 6] *... yeah no, I am very, very confused actually (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`"*.

Meanwhile, P6 became so confused with the meaning of the compound index expression, that he could not even see 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 comprehend any of the other `for` loops in Question 3, 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 discussion in this sub-section focuses on the three programming logic difficulties identified from the Question 3 think-aloud transcripts.

**The ripple effect**. This effect occurs when the misinterpretation of one statement has a direct impact on the execution of statements that follow it. This difficulty, which was observed with three participants, typically arises when programmers misinterpret programming logic [40]. Due to P1's struggle to interpret the array indexes (see Section 4.1), her interpretation of the statements contained in the third `for` loop completely ignored any changes made to the elements of the `b` array 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"*. She therefore chose "5" (Option E) as her final answer to Question 3, which was incorrect.

The difficulties that P6 had in interpreting the second `for` loop [Lines 5-6] (see Section 4.1 – Decomposition) caused him to overlook that loop completely while he was 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], *that `b` value is always `false`"*.

The behaviour displayed by both P1 and P6 indicated that they were not thinking sequentially [43] and therefore failed to follow the logic of the source code in question [44]. 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 Question 3. Her non-sequential reasoning is evident from 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 three participants is the "mental block" caused by the statements they could not fully comprehend and their consequent anxious behaviour (as observed by the interviewer). These participants tried to resolve the mental block by completely ignoring the troublesome statements as if those were no longer part of the code.

**Guessing**. One of the common critiques of MCQs is that they are answerable through guessing. This is also true of the 12 MCQs used in Part 1 of this study as guessing behaviour was observed by both Fitzgerald, Simon and Thomas [45] and Lister et al. [9], who used the same questions in their studies. The format of the Phase 2 think-aloud sessions discouraged guessing as participants were continuously prompted to explain their reasoning in as much detail as possible. However, one participant (P8) did attempt guessing when he said *"I just have to go with A"* after only tracing through a small

section of the 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 interfere with the comprehension logic. One example of such a mistake was observed when P7 failed to terminate 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 therefore treated the < as if it was a <= operator, which is typically regarded as a logical error in comprehension of source code.

### 4.3 Programming Control Structure Difficulty

The Question 3 code only contained one type of control structure in the form of three `for` repetition structures. As mentioned in the ripple effect discussion (see Section 4.2), 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 that contained 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 [29], 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 SCC Bottlenecks

The results of Phase 2 revealed that the participants in this study (senior CS students) experienced eight major SCC difficulties related to the concept of arrays, programming logic and programming control. In following existing bottleneck identification guidelines [1, 13], we used our collective experience of more than 25 years in teaching introductory and advanced programming modules combined with the new knowledge gained regarding difficulties experienced by our students, as well as relevant literature, to formulate six usable SCC bottlenecks.

**Bottleneck 1: Students are unable to keep track of variable values while tracing through a piece of code.** Throughout the think-aloud excerpts presented in Section 4, there are numerous examples where students lost track of the changes made to variable values, causing them to arrive at an incorrect answer. They all tried to remember the changes to the variable values (instead of making notes on the provided piece of paper), which put unnecessary strain on their working memories. Their incorrect answers were therefore a direct result of failing memory or guessing. Lister et al. [9] point out that when students document changes to variable values they are much more likely to arrive

at the correct answer. Most of the students in our study did not follow a reliable strategy to keep track of such value changes.

**Bottleneck 2: Students are unable to comprehend statements containing arrays and perform basic operations on array elements.** The bulk of the identified difficulties can be related to the students' incorrect understanding of array concepts, thereby supporting findings from previous studies in which arrays were also identified as one of the most challenging concepts for novice programmers [23-26]. Our students particularly struggled to interpret the array indexes – especially when it was integrated with other concepts. While one student confused the square brackets (indicating the array index) with a multiplication operator, others were unable to determine the length of an array. Although most students had little trouble to comprehend the array containing integer (numeric) values, many of them were completely lost when having to deal with the Boolean array.

**Bottleneck 3: Students are unable to comprehend the execution of basic `for` repetition structures.** Most of the difficulties observed with the `for` loops can be traced back to our students' incorrect comprehension of either the header or the body of the looping structure, as Garner et al. [24] also observed. While some students failed to recognise when and how to terminate the loops [29], an instance was also 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 completely gave up on interpreting the `for` loops and opted to ignore either the entire structure or the loop header completely for the remainder of their Question 3 interpretation.

**Bottleneck 4: Students do not possess adequate strategies to help them interpret lines of code they cannot comprehend.** 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 had no trouble to comprehend earlier – were combined to form a single "complex" concept. The students were unable to decompose [41] the more complex piece of code into smaller parts in order to simplify the interpretation thereof. Their most common response to this challenge was to ignore the complex statements or lines of code completely. Although decomposition is a task that many novice programmers struggle with [42], students may never learn how to deal with complex concepts if they are not taught explicit strategies to resort to in such situations.

**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 making up 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 for those students who chose to completely ignore 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 the question. Somewhat similar behaviour is evident in Shopkow et al.'s [19] description of their "ignoring significance" bottleneck – referring to History students' complete disregard for how individual facts relate to the story they are trying to tell.

**Bottleneck 6: Students are unable to reliably work their way through the long chain of reasoning required to comprehend a piece of source code.** This final bottleneck can be regarded as overarching since it refers to one of the most common and significant SCC difficulties originally identified by Lister et al. [9], and which we also observed in our study. It is directly related to our "ripple effect" difficulty that refers to mistakes made when students are unable to think sequentially [43] or fail to follow the source code logic [44]. In this study, we first-hand experienced 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 possess the required 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 [19] 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 uncover the major SCC bottlenecks experienced by senior CS students. Thematic analysis of data collected by means of asking questions, observations and artefact analysis revealed a series of SCC difficulties specifically related to arrays, programming logic and control structures. The uncovered difficulties, combined with findings from existing literature and the personal experiences of the authors, were then used to formulate six bottlenecks that are indicative of the typical mental challenges experienced by novice programmers during the comprehension of source code. By choosing to focus on senior students, we were able to identify major bottlenecks that point to student learning difficulties that are currently not adequately addressed in introductory programming courses, and therefore still influence the mental processes followed by final-year undergraduate students.

Through this paper, we also wanted to create 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. In order to address the remaining six steps of the DtDs framework [3], future research is firstly needed to uncover the mental tasks followed by expert programmers to overcome the six identified SCC bottlenecks. This knowledge can then be used to devise teaching and learning strategies that model the explicit mental strategies that experts follow. After creating opportunities for students to practice these skills and 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 protocol is to help students master the mental actions they need to be successful in the CS discipline.

## References


1. 1 Pace, D.: The Decoding the Disciplines Paradigm: Seven Steps to Increased Student Learning. Indiana University Press, Bloomington (2017)
2. 2 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). doi: 10.2307/25095328
3. 3 Middendorf, J., Pace, D.: Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. New Dir. Teach. Learn. 98, 1–12. Wiley Periodicals, Inc., Hoboken (2004). doi: 10.1002/tl.142
4. 4 Shopkow, L.: How many sources do I need? Hist. Teach. 50(2), 169–200 (2017). http://www.societyforhistoryeducation.org/pdfs/F17_Shopkow.pdf
5. 5 Verpoorten, D., et al.: Decoding the disciplines – A pilot study at the University of Liège (Belgium). In: The 2nd EuroSoTL conference, pp. 263–267. EuroSoTL, Lund (2017)
6. 6 Bosse, Y., Gerosa, M. A.: Difficulties of Programming Learning from the Point of View of Students and Instructors. IEEE Lat. Am. Trans. 15(11), 2191–2199 (2017). doi: 10.1109/TLA.2017.8070426
7. 7 Du Boulay, B.: Some difficulties of learning to program. J. Educ. Comput. Res. 2(1), 57–73 (1986). doi: 10.2190/3LFX-9RRF-67T8-UVK9
8. 8 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 2017 ACM Conference on International Computing Education Research, pp. 164–172. ACM, Tacoma (2017). doi: 10.1145/3105726.3106190
9. 9 Lister, R., et al.: A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bull. 36(4), 119-150 (2004). doi: 10.1145/1041624.1041673
10. 10 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). doi: 10.1287/isre.6.3.286
11. 11 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, Canterbury (2001). doi: 10.1145/572139.572181
12. 12 Sanders, K., Mccartney, R.: Threshold Concepts in Computing : Past , Present , and Future. In: Proceedings of the 16th Koli Calling International Conference on Computing Education Research, pp. 91–100. ACM, Koli (2016)
13. 13 Middendorf, J., Shopkow, L.: Overcoming Student Learning Bottlenecks: Decode Your Disciplinary Critical Thinking. Stylus Publishing, LLC, Sterling (2018)

14. 14 Pinnow, E.: Decoding the Disciplines: An Approach to Scientific Thinking. Psychol. Learn. Teach. 15(1), 94–101 (2016). doi: 10.1177/1475725716637484
15. 15 IUBCITL: Team-Based Learning For Practice and Motivation (2016). https://www.youtube.com/watch?v=1obB-n6JZ8k. Accessed 18 Oct 2018
16. 16 Menzel, S.: ISSOTL 2015 : Recursion as a Bottleneck Concept (2017). https://www.youtube.com/watch?v=iNvQlm9phEI. Accessed 2 Sept 2018
17. 17 German, A., Menzel, S., Middendorf, J., Duncan, F. J.: How to decode student bottlenecks to learning in computer science (abstract only). In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, p. 733. ACM, Atlanta (2014). doi: 10.1145/2538862.2544228
18. 18 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. Society for Teaching and Learning in Higher Education, Cape Breton University, Sydney, Nova Scotia, Canada (2013)
19. 19 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: Thompson, R. (ed.) Changing the Conversation about Higher Education. Rowman & Littlefield Publishers, New York (2013)
20. 20 Wilkinson, A.: Decoding learning in law: Collaborative action towards the reshaping of university teaching and learning. Educ. Media Int. 51(2), 124–134 (2014). doi: 10.1080/09523987.2014.924665
21. 21 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). doi: 10.15173/ijsap.v1i1.3061
22. 22 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 Computing Education, pp. 243–252. Australian Computer Society, Inc., Hobart (2006). doi: 10.1109/TPWRD.2010.2044424
23. 23 Hyland, E., Clynch, G.: Initial experiences gained and initiatives employed in the teaching of Java programming in the Institute of Technology Tallaght. In: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002, pp. 101–106. ACM, Dublin (2002)
24. 24 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)
25. 25 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, Koli (2018). doi: 10.1145/3279720.3279744
26. 26 Malik, S. I., Coldwell-Neilson, J.: A model for teaching an introductory programming course using ADRI. Educ. Inf. Technol. 22(3), 1089–1120 (2017). doi: 10.1007/s10639-016-9474-0
27. 27 Milne, I., Rowe, G.: Difficulties in Learning and Teaching Programming - Views of Students and Tutors. Educ. Inf. Technol. 7(1), 55–66 (2002). doi: 10.1023/A:1015362608943
28. 28 Butler, M., Morgan, M.: Learning challenges faced by novice programming students studying high level and low feedback concepts. In: Proceedings ascilite Singapore 2007, pp. 99–107. Ascilite, Singapore (2007)

29. 29 Grover, S., Basu, S.: Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, pp. 267–272. ACM, Seattle (2017). doi: 10.1145/3017680.3017723
30. 30 Gurevich, Y.: Logic and the Challenge of Computer Science. In: Borger, E., (ed.) Current Trends in Theoretical Computer Science, pp. 1–57. Computer Science Press, Rockville (1988). http://web.eecs.umich.edu/~gurevich/Opera/74.pdf. Accessed 17 Jan 2019
31. 31 Deitel, P. J., Deitel, H., Deitel, A.: Visual Basic 2012 How to Program. Pearson Education, Inc., Hoboken (2013)
32. 32 Qian, Y., Lehman, J.: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. ACM Trans. Comput. Educ. 18(1), 1-24 (2017). doi: 10.1145/3077618
33. 33 Plowright, D.: Using mixed methods: Frameworks for an integrated methodology. Sage Publications, London (2011)
34. 34 Patton, M. Q.: Qualitative research & evaluation methods: Integrating theory and practice, 4th edn. Sage Publications, Thousand Oaks (2015)
35. 35 Charters, E.: The use of think-aloud methods in qualitative research: An Introduction to think-aloud methods. Brock Educ. 12(2), 68–82 (2003). doi: 10.1080/02602938.2010.496532
36. 36 Haregu, T. N.: Qualitative Data Analysis (2009). https://www.slideshare.net/tilahunigatu/qualitative-data-analysis-11895136?next_slideshow=1. Accessed 29 Oct 2018
37. 37 Willes, K. L.: Data Cleaning. In: Allen, M. (ed.) The SAGE Encyclopedia of Communication Research Methods. Sage Publications, Inc., Thousand Oaks (2017)
38. 38 Marshall, C., Rossman, G. B.: Designing Qualitative Research, 6th edn. Sage Publications, Inc.,Thousand Oaks (2016)
39. 39 Klenke, K.: Qualitative Research in the Study of Leadership, 2nd edn. Emerald Group Publishing Limited, Bingley (2016)
40. 40 Kallia, M., Sentance, S.: Computing Teachers' Perspectives on Threshold Concepts: Functions and Procedural Abstraction. In: Proceedings of the 12th Workshop on Primary and Secondary Computing Education, pp. 15–24. WIPSCE, Nijmegen (2017). doi: 10.1145/3137065.3137085
41. 41 Keen, A., Mammen, K.: Program Decomposition and Complexity in CS1. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education, pp. 48–53. ACM, Kansas City (2015). doi: 10.1145/2676723.2677219
42. 42 Goldman, K., et al.: Identifying Important and Difficult Concepts in Introductory Computing Courses using a Delphi Process. In: Proceedings of the 39th SIGSE Technical Symposium on Computer Science Education, pp. 256–260. ACM, Portland (2008). doi: 10.1145/1352135.1352226
43. 43 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, Covington (2007). doi: 10.1145/1227504.1227482
44. 44 Alston, P., Walsh, D., Westhead, G.: Uncovering "Threshold Concepts" in Web Development: An Instructor Perspective. ACM Trans. Comput. Educ. 15(1), 1–18 (2015). doi: 10.1145/2700513
45. 45 Fitzgerald, S., Simon, B., Thomas, L.: Strategies that Students Use to Trace Code: An Analysis Based in Grounded Theory. In: Proceedings of the first international workshop on Computing education research, pp. 69-80. ACM, Seattle (2004). doi: 10.1145/1089786.1089793