

A Framework to Assist Instructors Help Novice Programmers to Better Comprehend Source Code – A Decoding Perspective

Pakiso J. Khomokhoana(✉) and Liezel Nel

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

Abstract. The Decoding the Disciplines paradigm posits that each discipline has unique mental operations – often invisible to instructors due to their own expert blind spots. If the nature of these operations is not made explicit to students, they are likely to develop learning bottlenecks that could prevent them from mastering key disciplinary practices – such as the ability to reliably work through the chain of reasoning necessary for efficient source code comprehension (SCC). This study seeks to assist instructors to help novice programmers reliably think and work their way through a long chain of reasoning to efficiently comprehend a piece of source code. We followed a narrative approach where data was collected through decoding interviews with five expert programming instructors. Several SCC strategies were identified, but only the seven key ones were used to devise a step-by-step framework for efficient SCC. Our findings aim to create awareness among instructors regarding the explicit mental operations required for efficient SCC.

Keywords: Source code comprehension · expert programming instructors · decoding the disciplines · decoding interview · computer science education · bottleneck · CS1

1 Introduction

Source code comprehension (SCC) refers to an ability to read, interpret, and understand existing source code [10, 17, 62]. Despite SCC being a critical disciplinary skill [29, 44, 48], many Computer Science (CS) students are still unable to reliably think and work their way through a long chain of reasoning to efficiently comprehend a piece of source code [20, 26]. This student-learning bottleneck can be attributed to students' fragile knowledge of basic programming concepts [5]. A bottleneck refers to specific places where the learning of many students gets interrupted [31]. This interruption happens when students are not sure how to approach a given problem and as a result, apply improper strategies [38].

The severity of a bottleneck can be further intensified when instructors are unable to accurately portray disciplinary ways of thinking to students [31]. This may occur

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023, LNCS 13956, pp. 677–693, 2023.
https://doi.org/10.1007/978-3-031-36805-9_44

due to instructors' expert blind spots, which typically occur when vital operations have become so natural to disciplinary experts that they tend to omit crucial mental steps when explaining concepts and procedures to others [36]. Decoding the Disciplines (DtDs) is a paradigm that focuses on increasing student learning by bridging the gap between the thinking of novice and expert programmers [32]. The seven-step DtDs paradigm can be used to initially expose the nature of such hidden operations (linked to a specific bottleneck) and to create awareness among instructors regarding steps/operations they typically omit when teaching their students. In subsequent steps of this paradigm, instructors are guided to devise ways of helping students master these operations and hence overcome specific learning bottlenecks [31, 38].

Bottlenecks are typically identified in Step 1 of the DtDs paradigm, while Step 2 focuses on exploring explicit mental steps that disciplinary experts follow to accomplish a task identified as a bottleneck [31, 38]. In executing Step 2, several artefacts and/or techniques (e.g., decoding interviews, rubrics, metaphors/analogies, mind maps, reflective writing, and non-verbal modelling) can be used to uncover the explicit steps followed by experts [31, 32]. However, decoding interviews are cited as the most rigorous and effective technique in this regard [38]. A decoding interview is a special type of interview where disciplinary experts are intellectually guided to reveal explicit steps they follow to get through a predetermined learning bottleneck [30, 38]. As a first step in dealing with a previously defined discipline-specific learning bottleneck (see 2nd sentence of this Section), this study focuses on Step 2 of the DtDs paradigm to answer the following two questions:

- What are the explicit mental strategies that CS experts employ while comprehending source code?
- How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards ways of assisting novice programmers to overcome their "inability to reliably think and work their way through a long chain of reasoning to efficiently comprehend a piece of source code"? (A bottleneck established in [20]).

The rest of this paper is structured as follows: Sect. 2 presents an overview of previously identified SCC strategies relevant to the specific learning bottleneck under investigation. It also provides a discussion comparing strategies used by both novice and expert programmers while comprehending source code. In Sect. 3, detail is provided regarding the selection of the decoding interview participants, interview panel, and validation issues. Section 4 provides the study findings, while Sect. 5 presents a proposed step-by-step framework for efficient SCC. The conclusions, recommendations for future work, and contribution of this study are presented in Sect. 6.

2 Related Work

2.1 Source Code Comprehension Strategies

Literature defines various generic SCC strategies namely, top-down, bottom-up, and variations that either combine these two strategies or incorporate elements thereof [27, 41, 60]. Novice programmers and expert programmers have been shown to favour different strategies, with bottom-down mostly observed with novices [35]. Although the

basic steps involved in each of these generic SCC strategies are well documented, more specific details are needed to truly understand the explicit mental operations required for efficient SCC. Previously, researchers used observations [51] and think-alouds [26] to gather more information on specific strategies employed during SCC.

In evaluating ways by which students answered code-based Multiple-Choice Questions (MCQs), Fitzgerald et al. [11] identified 19 strategies used by novice programmers. These range from reading the question; previewing the code (e.g., by identifying data structures and their initialisation); understanding new concepts (semantic); pattern recognition (e.g., temporally self-referential, outside knowledge and seeking higher levels of meaning from the code); walkthroughs; strategising; thoroughness; starting over; coming back to the question later; posing questions; and doodling. Although this list includes several good SCC strategies, as also identified by other researchers (e.g., [26]), the novices did not always execute these strategies in an optimum way. However, Fitzgerald et al.'s [11] strategies are much more specific than the generic SCC strategies alluded to in the previous paragraph. Although students and experts do not necessarily follow the same SCC strategies, these generic strategies could serve as a starting point to identify more explicit details on the exact mental operations required for effective SCC.

In Fitzgerald et al.'s [11] study, the novices used the reading the question and previewing strategies to acquaint themselves with some elements of the code-based questions they were answering. This type of self-orientation is a strategy used by people to familiarise themselves with the elements of the problem to be tackled [16]. Simon et al. [52] emphasise the importance of reading programs or pieces of code to comprehend them. In this regard, Moore et al. [34] even suggest reading through question specifications or a piece of source code twice. To further enhance comprehension of a task, strategies such as highlighting, underlining, and colouring some words or text can also be used [43, 46]. These important words or pieces of text could be regarded as key for a programmer's comprehension of either the problem description or code in question. However, the use of keywords is one SCC strategy that has not been observed with Fitzgerald et al.'s [11] novices.

Littman et al. [27] observe that programmers use either systematic (line-by-line) source code reading or control/data-flow abstractions to better comprehend the behaviour of a given program or piece of code. For novice programmers to fully comprehend code behaviour, they need to have a global understanding of the piece of source code in question. Global understanding entails gaining an overall understanding of the problem to be solved or the question to be answered before trying to understand the minute details of the task. This understanding already starts to develop in the previewing SCC strategies suggested by Fitzgerald et al. [11]. Comprehension can also be enhanced through pattern recognition where similar or related code elements are organised into categories [22].

Similar to solving many real-world problems, the solving of SCC problems also requires the application of logical reasoning processes [4]. The logical reasoning processes are also a huge challenge to students in comprehending code [19]. In SCC problems, test cases can, for example, be used to check the logic of both basic and advanced conditions [58]. To evaluate these test cases, a programmer typically conducts a walkthrough of the given source code. Walkthroughs are defined as "simply reading the code carefully in the order it would be executed (except for branch points, where all branches

are considered serially), to careful simulation, where the [programmer] attempts to mimic as closely as possible the actions of the [computer/compiler] that executes the code" [18] (p. 12). Other ways in which programmers can apply logical reasoning, is by strategising about how they would solve a given problem if they had to write code from scratch or by asking themselves specific questions (i.e., posing questions) that could help with their comprehension of a given piece of code. The thoroughness strategy, where an answer is re-checked to confirm the correctness thereof, can be applied in answering any type of question. Frederick [12] (p. 35) states that people should have "the ability or disposition to resist reporting the response that first comes to mind" – thereby suggesting that, regardless of the type of question, an answer to a question should always be evaluated to confirm its correctness thereof.

Given the relative simplicity (e.g., in terms of instructions and code fragments used) of the code-based questions used in Fitzgerald et al.'s [11] study, the starting over and coming back to the question later strategies could be regarded as typical novice strategies that one would not necessarily expect to observe with expert programmers.

From this discussion, it is evident that although various SCC strategies have previously been identified, some of these strategies seem to be more novice-specific. For comprehensive literature coverage, a discussion of similarities and differences between novices and experts during SCC could be necessary.

2.2 Comparing Novice and Expert Programmers

In the initial stages of SCC, both experienced and novice programmers follow similar overall strategies, but their strategies differ later on [18]. Experienced programmers use their experience, syntactic knowledge, and knowledge of a problem domain, while novice programmers read source code line-by-line [23, 53]. Experienced programmers focus only on reading source code relating to a particular task at hand, while novice programmers focus on all elements of the code [27, 56]. Experienced programmers further use a semantic approach, while novice programmers are driven by how a program works syntactically rather than what a program does semantically [1, 54]. Experienced programmers get more affected by violations of the rules of discourse in a piece of code than novice programmers [55, 57]. Both experienced and novice programmers pay the least attention to the keywords in the source code's text [9].

Experienced programmers outdo novices in situations where they must recall meaningful source codes. However, both do equally well when they must recall source codes that are not well-designed [47]. Experienced programmers link parts of the source code to the problem domain, which is unusual for novice programmers [41]. It is easier for experienced programmers to realise when they must change or adapt their comprehension strategy – especially as a result of discovering an anomaly in the source code or when the requested task has some inherent special needs [59]. Experts have, however, been observed to revert to a line-by-line strategy in cases where they were not familiar with a programming system [21].

The next section presents the research design and procedure that was followed to identify strategies that expert programming instructors follow during SCC.

3 Research Design and Methods

A narrative research approach based on Plowright's [42] Frameworks for an Integrated Methodology (FraIM) was adhered to in this study. The data source management strategy was a case study. Data were mainly collected through face-to-face decoding interviews [31] (as a means of asking questions), while observations were used as a supplementary strategy. The study population consisted of CS instructors from a selected South African university. From this population, five instructors were purposefully selected [7] based on their experience in teaching programming courses. Two of the participants (P1 and P4) had more than 14 years of experience in this regard, while P2 and P3 had between five and nine years of similar experience. Except for P5, all the other participants worked as industry programmers for at least four years, and they were all, to some extent, still involved in private programming consultancy work. This sample can also be regarded as convenient [40] since the selected participants were close to the researcher and therefore easily reachable. Ethical clearance for this study was obtained from the selected institution and all ethical considerations were adhered to throughout the study investigations.

3.1 Decoding Interviews

Decoding interviews are considered intellectually very demanding [31], where a single interviewer can easily get lost in the details. Pace [38] recommends the involvement of at least two interviewers, as two minds are in a better position to control the interview process. Both interviewers should be able to verbalise their thinking, challenge the interviewees' explanations, and summarise the interviewees' thinking back to them at an abstract level [50]. As members of the same discipline are more likely to share common expert blind spots, Pace [38] also suggests using a second interviewer from outside the discipline in question. Such an interviewer is more likely to notice when mental steps are not well explained. Since we could not find a readily available individual with the relevant decoding-interview experience from outside the CS discipline, we instead selected a non-teaching CS researcher who had some decoding-interview experience as the second interviewer. The decision was also influenced by the context of this research activity. Given the highly discipline-specific nature of SCC, someone from outside the discipline might not necessarily be able to follow the reasoning of the interviewees and could find it difficult to instantly think of appropriate and relevant probing questions to ask.

Separate decoding interviews were conducted with each of the five selected participants. The principal researcher (first author) acted as the main interviewer, while the second interviewer (as described in the above paragraph) played a supporting role. The proceedings of each interview were audio recorded with the permission of the participant. Where relevant, observations were also recorded by the principal researcher. The main purpose of the decoding interviews was to uncover the explicit mental strategies and steps followed by participants during an SCC task. In the first part of the decoding interview, each participant was asked to explain the steps they would follow when requested to predict the output of any piece of source code provided on a piece of paper. The

participants' responses to this general question allowed the interviewers a first glance at some of the basic SCC strategies utilised by expert programmers.

To uncover more explicit details regarding the nature of the shared strategies, the participants were each presented with an SCC task and asked to illustrate how they would implement their SCC strategy (as explained in the first part of the interview) to solve the given problem. The question selected for this activity was sourced from the original set of 12 MCQs used in Lister et al.'s [26] study. The selected question (Question 6 – see Fig. 1) was identified as the second most challenging question in the Lister et al. [26] study. This question was particularly selected as it covers a wider range of concepts (e.g., Boolean variables, for loops, array indices, and the use of a return statement to terminate the for loop) than the most challenging question (Question 12, which mainly focuses on arrays). In answering the selected question, the missing piece of code had to be identified from the five given options. (Note: The correct answer is Option B). The only change made to the question was to convert the original Java code into C# (which was the language mainly used by participants in their teaching). The code line numbers as indicated in Fig. 1 were only added in aid of the discussion in this paper.

3.2 Data Analysis

An adapted version of Creswell and Creswell's [8] narrative Data Analysis Framework guided the transcription of the audio recordings (made during the decoding interviews), as well as the analysis of the resultant narrative data. Considering the open-endedness of the decoding interview proceedings, a fuzzy-validation strategy [39] was employed to clean the data. This strategy allows some corrections to the data if there is a close match or known answer. The resulting transcripts were validated by each participant as part of member checking [25]. Inherently, it is well-accepted that in dealing with narrative inquiries, the researcher is regarded as the instrument [40]. As such, we had to immerse ourselves in the data to be fully familiar with its breadth and depth [3]. This was achieved through several counts of listening and re-listening to the audio recordings, coupled with intensive reading and re-reading of the transcripts.

After immersing ourselves in the data, we decided on a coding plan that would help with the analysis of the data to address the research questions. The five validated transcripts were imported into NVivo 12 software and codes were created based on the strategies identified in the data. Subsequently, words and/or short phrases [45] containing indications of the relevant strategies or steps were extracted (under the guidance of the theoretical guidelines from literature) from the imported transcripts onto the various nodes (forming codes) created on NVivo. As coding gives rise to recurring themes [45], the extraction and movement of the relevant text gave birth to such themes. For each theme, NVivo generated the frequency of occurrence, hence making it easier to put the data back together to make new meaning in relation to fully answering the research questions of the study [24].

3.3 Validation

After completion of the data analysis, an initial step-by-step framework for efficient SCC was compiled. To enhance the validity of the framework, we deemed it necessary

to have this framework evaluated and validated by an informed audience. The validation was deemed necessary to enhance the trustworthiness [49] of this study's findings. In this regard, two separate validation activities were conducted. First, the second decoding interviewer reviewed the initial framework to confirm that the identified strategies and steps were a true representation of the data gathered during the decoding interview; and to check all the statements for clarity and ambiguity. Second, a validation meeting was arranged with five participants (one junior lecturer, a postdoctoral student, and three professors) from the CS department. One of the professors earlier participated in the decoding interviews (as described in Sect. 3.1). The purpose of this meeting was to further check for possible ambiguities in the proposed steps/strategies. To validate the implementability and usefulness of the framework, participants were requested to follow the framework steps (as closely as possible) while answering two SCC questions. After an explanation of the framework, participants worked on solving the first problem under the guidance of the principal researcher. For the second question, each participant independently followed the framework steps to answer the question. This was followed by an open discussion where participants shared their experiences in using the framework to answer the two questions. Some issues regarding the wording of some of the steps came to light and recommendations for possible changes and additions were discussed.

The following method **isSorted** should return **true** if the array is sorted in ascending order. Otherwise, the method should return **false**:

```
1. public static bool isSorted (int[] x)
2. {
3.   //missing source code goes here
4. }
```

Which of the following is the missing source code from the method **isSorted**?

a)
```
5. bool b = true;
6. for (int i = 0; i < x.Length - 1; i++)
7. {
8.    if (x[i] > x[i + 1])
9.          b = false;
10.   else
11.         b = true;
12. }
13. return b;
```

b)
```
14. for (int i = 0; i < x.Length - 1; i++)
15. {
16.   if (x[i] > x[i + 1])
17.         return false;
18. }
19. return true;
```

c)
```
20. bool b = false;
21. for (int i = 0; i < x.Length - 1; i++)
22. {
23.    if (x[i] > x[i + 1])
24.       b = false;
25. }
26. return b;
```

d)
```
27. bool b = false;
28. for (int i = 0; i < x.Length - 1; i++)
29. {
30.   if (x[i] > x[i + 1])
31.         b = true;
32. }
33. return b;
```

e)
```
34. for (int i = 0; i < x.Length - 1;i++)
35. {
36.   if (x[i] > x[i + 1])
37.         return true;
38. }
39. return false;
```

Fig. 1. Source code question used in the decoding interview

Based on the feedback received during this validation meeting, a few minor changes were made to the initial framework. The findings of this study are presented in Sect. 4.

4 Results and Interpretation

Based on the analysis of the decoding-interview transcripts, several SCC strategies were observed with our participants. However, only strategies with four or more occurrences are presented except for the keyword identification strategy (two occurrences). Though not commonly mentioned by the participants, this strategy was frequently observed during the decoding interview when participants were tackling the given task.

4.1 Self-orientation

During self-orientation, programmers typically read the question and perform a lot of code previewing on related aspects as suggested by Fitzgerald et al. [11]. Application of this strategy was identified in all our participants, with a total of 18 occurrences. Participant 4 (P4) employed the strategy the most, with eight occurrences.

As part of his self-orientation, P4 shared very specific details about how he typically starts to comprehend a piece of code and provided reasons for his actions. Linking to the importance of reading the question description and code statement [52], P4 also provided some insight regarding the reading intensity he would employ in the process:

> *"Whenever I read code, I browse through it very quickly, not looking at detail. I try to get the basic idea and then I browse through it again. So, I do not read once from top to bottom and then I am done – never! I read through a piece of code more than once – three, four, five times ... Some of the details will go slow, and some of them will be quick depending on my familiarity with that specific code fragment. ... I will go and look at it globally. Again, I will make sure that my conceptual understanding of what it is supposed to be, is in order."*

From this excerpt, it is apparent that P4 read the source code more than once (i.e., reread – [34]) to ensure that he attained the correct understanding. Expert programmers tend to read code at least twice (even if they understood it at first) just to confirm their original understanding [52]. To confirm the importance of rereading code, P4 was asked if he ever reads a piece of code just once, even if it is very simple. In response, he said: *"No, it depends on the length. If it is two lines of code, yes, then I might read it more than once by looking at it once. I mean, your brain can cognitively observe a thing more than once, while visually looking at it once."* This response could serve as an indication that experts understand that there is some coordination between seeing something and processing it in the brain, as reading is cognitive in nature [37].

4.2 Walkthroughs

Adopting the definition of code walkthrough by Jeffries [18] (see Sect. 2.1), sixteen occurrences of this strategy were identified. P4 employed walkthroughs the most, with seven occurrences.

There were several instances where participants modelled this strategy through their mental actions, as suggested by Hertz and Jump [15]. P2 in particular, said:

> "*I will do a trace table. And I will draw and say, this is where I am tracing the code. I will carry on to another trace table. If I have the code on a piece of paper, I will go and write – in line 1, this is what is happening, and this is my variable. In line 2, this is what is happening. In line 3, we are making a function call to that method. And then I will jump to that method and associate it with lines and write out a picture of what is happening."*

In this instance, P2 was observed physically drawing a trace table and putting in arbitrary labels for the respective input and output values of variables. She was actually trying to make her SCC steps or processes as visible as possible [6] and also modelling what was happening in her mind [15].

Concerning situations where the use of test cases might not be specifically feasible, other participants indicated that they would not use test cases as suggested by Srikant and Aggarwal [58], because it would be time-consuming to consider all cases available in each scenario. In this regard, P2 said: *"Let us say I have an array of 100 elements, I am not going to sit in class and draw 100 things on my trace table, so I will do a few."* However, P2 would be careful in selecting a limited set of test cases that would at least cover the *"worst, average, and best cases"*, thereby accommodating testing boundary or error conditions as specified by Fitzgerald et al. [11].

4.3 Strategic Thinking

Strategic thinking involves the use of high-level and critical thinking as well as logical reasoning in both understanding and solving a problem, thereby approaching this problem from a variety of angles or differing perspectives [14]. Activities performed as part of the strategising and posing questions strategies, as suggested by Fitzgerald et al. [11], require high-level reasoning [26]. These strategies, therefore, challenge programmers to tap into their strategic skills. Fifteen occurrences were identified where this strategy was used. P4 employed the strategy the most, with seven occurrences. It is impossible not to use logical/strategic reasoning processes in problem-solving [4]. P4 used an example of a repetition structure to explain his reasoning while dealing with such a structure*: "If I read code, let us say there is a while loop. The first things I look for are: Are the three elements there? (Do not look at the code, look if those three elements are there.) Is there a condition? Is the condition initialised? Is there a place somewhere in the loop where the conditions will be changed? If those three elements are not there, the while loop is not going to work."* As can be seen from this excerpt, P4's strategic reasoning allowed him to understand that it would be useless to read the code further [27, 56] if the conditions under which such a repetition structure would operate are not met. It is also evident in the excerpt that P4 would be very careful to heavily rely on code functionality than other aspects such as code syntax.

4.4 Revisit Previous Stages

In executing the revisit previous stages strategy, a programmer moves back and forth between different parts of the question (problem specification and/or code). This strategy is typically performed to ensure a complete understanding of concepts and to integrate various aspects contained in the question to be answered and/or the problem to be solved. Five occurrences of this strategy were identified among the participants in this study. P3 employed the strategy the most, with three occurrences.

As an indication that P2 would check previous occurrences of a certain variable if a need arises, she said: "*And you can always refer back if you forget that there was this variable.*" Similarly, P3 said: "*If it is something I cannot fit into my working memory and reliably remember what happened earlier in the program, I have to continually refer back to the previous part of the program just to familiarise myself again.*" It can be deduced from P3's excerpt that this strategy is not applied all the time. Instead, participants (as experts) employ the notion of the capacity of the working memory [33], to say there is no need to revisit previous stages if information needed can be recalled from the working memory. This resonates with Wiedenbeck et al. [61] that expert programmers use little working memory during SCC.

4.5 Data Structure Identification

The identification of data structures entails locating a place where a variable/constant or an object is first encountered (i.e., declared), and identifying the value originally assigned to it (i.e., instantiation or initialisation). The identification occurs mostly during the previewing strategies (e.g., identifying data structures, the initialisation of data structures, and control structures) of Fitzgerald et al. [11]. If a data structure is unfamiliar to a programmer, they could resort to using the *understanding new concept (semantic) strategy*, thereby suggesting that previewing the code and understanding new concepts (e.g., semantic) may be usable in the identification of data structures. Four occurrences of this strategy were identified in the participants of this study. P3 employed the strategy the most, with three occurrences. P2 indicated that, although she would do it faster than students might do it (as a result of experience gathered over many years), she would still identify the data structures involved in a given piece of code. This is a very valid statement since identifying and understanding data structures is a challenge often experienced by novice programmers [28]. In this regard, P2 said*: "I do not have to look line-by-line and explain to myself what is happening. I can follow it much quicker than a student might. So, there might be things that I skip, but I would still look at the variables."* P2's revelation that she would skip some steps could point to an expert blind spot – causing her not to share the same steps she would personally follow while teaching her students. This action could have a negative impact on the SCC's understanding of her students [2].

4.6 Deduction of Meaning from Context

Deduction of meaning from context entails understanding the meaning of certain challenging concepts from reading associated statements or pieces of text. This could form

part of Fitzgerald et al.'s [11] understanding of new concepts (semantic) strategy. To deduce meaning from the problem context, expert programmers follow various courses of action. Three of our participants (though four occurrences) shared what they would do if they came across challenging concepts. P4 employed this strategy the most, with two occurrences. He shared the following in this regard: "*So I carry on, and the context of the global view might clarify that little piece that I do not understand. It happens quite often that if I understand something globally, it will lead me to the details that I do not understand, and it becomes easier. It is all a matter of context. It is easier to understand difficult parts if I have the context to which they belong*." It, therefore, becomes evident that failing to understand how a specific concept is used does not necessarily block the understanding of experts in terms of how that concept works. As established by Fitzgerald et al. [11] and observed with P4, experts do not have a problem proceeding with subsequent steps, because they believe that such steps may give them some idea(s) that could help to improve their understanding of difficult or unclear concepts.

4.7 Keyword Identification

A keyword is a word/concept that provides preliminary ideas on the significance of such words/concepts in the task to be tackled, hence pre-empting a person to remember it throughout the task. As indicated by Powell et al. [43], different ways can be used to identify keywords while comprehending source code. Two occurrences of using keywords were identified in two of the participants. In this regard, P2 said: *"What will catch my attention are the keywords. If I have syntax highlighting, it is a lot easier to identify the keywords."* This suggests that while using an integrated development environment (IDE) such as Microsoft Visual Studio that provides this colour functionality, P2 would take advantage of the built-in syntax highlighting [29, 46] to identify important words in the code.

Another participant, P4, took advantage of the keywords contained in the problem specifications by earmarking them as his main focus. He explicitly indicated that the other words were just there to link all the ideas together: *"The first thing to do is to make sure that I understand the question. You have helped me a bit by boldfacing some words. So, I will read the question, focusing only on the boldfaced words. Then I will read it again, and I will boldface in my mind some other words such as array, method, and sorted. So, these are the words that immediately come to mind. The other words glue everything together."* In this regard, it should be noted that, in the question presented to participants (see Fig. 1), words that referred to specific coding concepts were formatted in a different typeface – merely to distinguish it from the normal sentence text. P4 took advantage of our formatting strategy by using these words to mentally prepare himself for his SCC endeavour and to enhance his understanding of what was required of him in answering the question. In contrast to Crosby and Stelovsky [9], our expert participants focused more on the keywords. These words are scaffolds that help in making links to known information or previously seen/solved problems [13].

5 Framework for Efficient SCC

By using strategies identified in Sect. 4 together with insights gained from literature, observations made of expert participants during the decoding interviews, feedback received from the validation meeting, and the authors' collective experience as programming instructors, a proposed step-by-step framework for efficient SCC (see Table 1) was formulated. As an example, Step 1 was formulated by using the relevant aspects of the five SCC strategies (self-orientation, keyword identification, strategic thinking, and revisiting previous stages; and Step 2 was constituted from the necessary elements of the two strategies namely, self-orientation and revisiting previous stages. As each strategy has various dimensions in its implementation, some strategies have been used in the formulation of more than one step. Typically, strategic thinking has been used in the formulation of seven steps, while self-orientation in six steps. This framework contains 10 key steps linked to each of the relevant mental strategies used by the experts in executing each of these steps (as discussed in Sect. 4). However, within some of the main steps, several sub-steps can be performed. In using this framework, it is recommended that users put a tick mark (✓) against each step/sub-step they use, and a cross mark (✗) against any step/sub-step they do not use. The additional resource(s) mentioned in Step 1 could include official study material, resources from the Internet, or an expert (e.g., instructor, tutor, student assistant). It is also suggested that users of this framework should be encouraged to revisit previous steps whenever they get stuck.

Table 1. Step-by-step framework for answering a source code comprehension question

Strategies Applied	Steps	Description
• Self-orientation • Keyword identification • Strategic thinking • Revisit previous stages	1. Understand the problem	Read through the question statement/requirements **at least twice** (until you understand what you have to do). • **Highlight/mark** important words and/or phrases and make sure you understand their meaning or implication. • If there are any words and/or phrases that you do not understand, consult any additional resource(s)* for clarification. • If I were to write the code to solve the problem, how would I do it?
• Self-orientation • Revisit previous stages	2. Scan through the code (not recognising syntax)	Preview **all** the given code by scanning through it **at least twice** (to get a global overview). • Do not look at detailed syntax.
• Self-orientation • Data structure identification • Strategic thinking • Walkthroughs	3. Scan through the code (recognising syntax)	Scan through the code line-by-line. • Identify all the data structures. • Identify all the control structures (*e.g., Sequence, Iteration, Selection*). • Identify any methods/functions/properties. • Make sure that you understand the syntax and meaning (*e.g., semantics*) of each individual code fragment/statement. • Mark any code syntax and/or code fragments/statements that you do not understand. • Mark code fragments/statements that are similar or repeated.
• Self-orientation	4. Seek clarification	If there is code syntax that you do not understand, consult any additional resource(s) for clarification.
• Deduction of meaning from context	5. Use context	If there are still code fragments/statements that you do not understand, consider the context in which the fragment/statement is used. (**Note:** A more global view of the context in which the code fragment/statement is used might help to clarify your misunderstanding).
• Self-orientation • Strategic thinking • Revisit previous stages	6. Confirm your understanding	Scan through all the code again (**as many times as necessary**) to make sure that you **fully** understand how everything fits together. • Repeat **Step 3**, **Step 4** and/or **Step 5** if necessary. • Draw a diagram to visualise your understanding of the program logic (if applicable).
• Strategic thinking	7. Scan through the code (not recognising syntax)	If the question requires you to select the correct code fragment/statement(s) from multiple options: • Identify the option(s) that look(s) more correct. (Consider these first in **Step 8**). • Identify options that could possibly be incorrect. (Only consider these *possibly incorrect* options if none of the *more correct* options turn out to be a valid/correct answer). • Explain to yourself why you think some option(s) could be more correct than others.
• Self-orientation • Strategic thinking • Walkthroughs	8. Deal with multiple choice questions	Trace through the code by executing (from the top) each line according to the rules of the programming language. • Whenever a new variable/constant/object is created, write down its name and the initial value(s) (if applicable) on a piece of paper. (*Suggestion: Start a trace table*). • Record any changes to the value(s) of the variables/objects on your piece of paper. • Make any applicable drawings, notes or annotations that could help you keep track of or follow the program logic. (**Do not try to keep it all in your head!**)
• Strategic thinking • Revisit previous stages	9.Trace and execute the code	Write down your answer. • If it is not a valid answer, repeat **Step 8** using one of the other answer options.
• Strategic thinking • Walkthroughs • Revisit previous stages • Thoroughness	10. Report your result	Repeat **Step 8** to confirm the correctness of your final answer. • Use your own test case values (if not provided).

6 Conclusions and Future Work

The gap that exists between the ways in which novice and expert programmers comprehend source code continues to be a challenge. A better understanding of those mental operations that have become invisible to instructors (because they perform them automatically), could be valuable in enhancing novice programmer thinking. This can be achieved by uncovering the explicit nature of the mental strategies followed by experts during SCC. By focusing on Step 2 of the DtDs framework, this study utilised decoding interviews to systematically deconstruct mental operations performed by expert programmers while comprehending a piece of source code. Thematic analysis of the data collected during the decoding interviews revealed several strategies that expert programmers typically employ during SCC. However, this paper reports only on the seven key of those strategies. Given the nature of the source code question used as an example and the previous experience in programming subjects, it is worth noting that most of the strategies described by Fitzgerald et al. [11] and in this paper are very imperative-oriented. What also became apparent, is that each expert approaches SCC differently. At any stage during the SCC process, experts' prior knowledge or experience can trigger them to use specific strategies. Experts also find it easy to switch to a completely different strategy based on what they are currently thinking and the information or details they are encountering.

The SCC strategies identified in this study, combined with existing knowledge (from literature and based on the authors' experience), were used to develop a proposed step-by-step framework for efficient SCC. This framework is set to create awareness among instructors regarding the explicit mental operations required for efficient SCC. Knowledge of the nature of these mental operations could help instructors to better understand their expert blind spots. Furthermore, as a practical contribution within the realm of the DtDs philosophy [31], this framework could also serve as a starting point for devising explicit strategies to model these mental operations to students and to help them master each of the identified strategies. It is also believed that the framework has the potential to make a theoretical contribution to the field of CS education as a source of further research on efficient SCC strategies. This framework could also stimulate further research regarding the application and refinement of the framework itself.

The distinct decoding interview approach followed in this study – where multiple experts were observed and questioned (regarding their mental actions) while performing an actual discipline-specific task – could be regarded as an extension of the traditional decoding-interview approach. In disciplines where it is possible to observe actual tasks in real-time, a similar decoding-interview strategy could be used to uncover even more explicit details regarding the mental operations required to overcome discipline-specific student learning bottlenecks. This study also serves as further proof that the DtDs paradigm can be used in the investigation of classroom practices as suggested by Middendorf and Pace [31]. Consequently, such a research approach should hold particular appeal for instructors working in CS education as well as computing education and other related disciplines.

References

1. Adelson, B.: Structure and Strategy in the Semantically-Rich Domains. Harvard University, Cambridge (1983)
2. Ambrose, A.S., Bridges, W.M., DiPietro, M., Lovett, C.M., Norman, K.M.: How learning works: Seven research-based principles for smart teaching. John Wiley & Sons (2010). https://doi.org/10.1002/mop.21454
3. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qual. Res. Psychol. **3**(2), 77–101 (2006). https://doi.org/10.1191/1478088706qp063oa
4. Butler, M., Morgan, M.: Learning challenges faced by novice programming students studying high level and low feedback concepts. In: Atkinson, R., McBeath, C., Swee Kit, A.S., Cheers, C. (Eds.) Proceedings of Ascilite Singapore 2007 ICT: Providing Choices for Learners and Learning, pp. 99–107. Nanyang Technological University (2007)
5. Cheah, C.S.: Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. Contemp. Educ. Technol. **12**(2), 1–14 (2020). https://doi.org/10.30935/cedtech/8247
6. Chou, C.Y., Sun, P.F.: An educational tool for visualizing students' program tracing processes. Comput. Appl. Eng. Educ. **21**(3), 432–438 (2013). https://doi.org/10.1002/cae.20488
7. Cooper, D., Schindler, P.: Business Research Methods, 12th edn. McGraw-Hill Education (2013)
8. Creswell, J.W., Creswell, J.D.: Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 5th edn. Sage (2017)
9. Crosby, M.E., Stelovsky, J.: How do we read algorithms?: A case study. Computer **23**(1), 25–35 (1990). https://doi.org/10.1109/2.48797
10. Fellah, A., Bandi, A., Yousef, M.: Moving towards program comprehension in software development: A case study. In: Proceedings of the 4th International Conference on Computing Methodologies and Communication, ICCMC 2020, pp. 660–665 (2020). https://doi.org/10.1109/ICCMC48092.2020.ICCMC-000122
11. 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 (2005). https://doi.org/10.1145/1089786.1089793
12. Frederick, S.: Cognitive reflection and decision making. J. Econ. Perspect. **19**(4), 25–42 (2005). https://doi.org/10.1257/089533005775196732
13. Gaspar, A., Langevin, S.: Restoring "coding with intention" in introductory programming courses. In: SIGITE'07 - Proceedings of the 2007 ACM Information Technology Education Conference, March 2016, pp. 91–98 (2007). https://doi.org/10.1145/1324302.1324323
14. Grundy, T.: Demystifying strategic thinking: lessons from leading CEOs. Kogan Page Publishers (2014)
15. Hertz, M., Jump, M.: Trace-based teaching in early programming courses. In: Proceedings of the 44th ACM Technical Symposium on Computer Science Education, pp. 561–566 (2013). https://doi.org/10.1145/2445196.2445364
16. Illeris, K.: Learning, identity and self-orientation in youth. Nord. J. Youth Res. **11**(4), 357–373 (2003). https://doi.org/10.4324/9781315620565-6
17. Izu, C., et al.: Fostering program comprehension in novice programmers - learning activities and learning trajectories. In: Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE, December, pp. 27–52 (2019). https://doi.org/10.1145/3344429.3372501
18. Jeffries, R.: A comparison of the debugging behavior of expert and novice programmers. In: Proceedings of AERA Annual Meeting, pp. 1–17 (1982)

19. Karahasanović, A., Thomas, R.C.: Difficulties experienced by students in maintaining object-oriented systems: An empirical study. Proc. Ninth Australas. Conf. Comput. Educ. **66**, 81–87 (2007)
20. Khomokhoana, P.J., Nel, L.: Decoding Source Code Comprehension: Bottlenecks Experienced by Senior Computer Science Students. In: Tait, B., Kroeze, J., Gruner, S. (eds.) SACLA 2019. CCIS, vol. 1136, pp. 17–32. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-35629-3_2
21. Ko, A.J., Uttl, B.: Individual differences in program comprehension strategies in unfamiliar programming systems. In: Proceedings of the 11th IEEE International Workshop on Program Comprehension (2003). https://doi.org/10.1109/WPC.2003.1199201
22. Kpalma, K., Ronsin, J.: An overview of advances of pattern recognition systems in computer vision. In: Obinata, G., Dutta, A. (Eds.) Vision Systems: Segmentation and Pattern Recognition, pp. 357–382. IntechOpen (2007). https://doi.org/10.5772/4960
23. Letovsky, S., Soloway, E.: Delocalized plans and program comprehension. IEEE Softw. **3**(3), 41–49 (1986). https://doi.org/10.1109/MS.1986.233414
24. Lewins, A., Silver, C.: Using Software in Qualitative Research: A Step-by-Step Guide. Sage Publications (2007)
25. Lincoln, Y.S., Guba, E.G.: Naturalistic inquiry. Sage Publications (1985)
26. Lister, R., et al.: A multi-national study of reading and tracing skills in novice programmers. In: SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), vol. 36, no. 4 (2004). https://doi.org/10.1145/1041624.1041673
27. Littman, D.C., Pinto, J., Letovsky, S., Soloway, E.: Mental models and software maintenance. J. Syst. Softw. **7**(4), 341–355 (1987). https://doi.org/10.1016/0164-1212(87)90033-1
28. Litvinov, S., Mingazov, M., Myachikov, V., Ivanov, V., Palamarchuk, Y., Sozonov, P., Succi, G.: A tool for visualizing the execution of programs and stack traces especially suited for novice programmers. In: Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 235–240. https://doi.org/10.5220/0006336902350240
29. Medvidova, M., Porubän, J.: Program comprehension and quality experiments in programming education. In: OpenAccess Series in Informatics, vol. 102, no. 14, pp. 14:1–14:0. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany (2022). https://doi.org/10.4230/OASIcs.ICPEC.2022.14
30. Middendorf, J., Baer, A.: Bottlenecks of information literacy. In: Gibson, C., Mader, S. (Eds.), Building Teaching and Learning Communities: Creating Shared Meaning and Purpose, pp. 51–68. ACRL Publications (2019)
31. Middendorf, J.K., Pace, D.: Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. New Dir. Teach. Learn. **2004**(98), 1–12 (2004). https://doi.org/10.1002/tl.142
32. Middendorf, J., Shopkow, L.: Overcoming Student Learning Bottlenecks: Decode Your Disciplinary Critical Thinking. Stylus Publishing, LLC (2018)
33. Miller, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychol. Rev. **101**(2), 343–352 (1956). https://doi.org/10.1037/h0043158
34. Moore, D., Zabrucky, K., Commander, N.E.: Metacomprehension and comprehension performance in younger and older adults. Educ. Gerontol. **23**(5), 467–475 (1997). https://doi.org/10.1080/0360126970230506
35. Mosemann, R., Wiedenbeck, S.: Navigation and comprehension of programs by novice programmers. In: Proceedings of the 9th International Workshop on Program Comprehension, pp. 79–88 (2001). https://doi.org/10.1109/WPC.2001.921716

36. Nathan, M.J., Petrosino, A.: Expert blind spot among preservice teachers. Am. Educ. Res. J. **40**(4), 905–928 (2003). https://doi.org/10.3102/00028312040004905
37. O'Brien, M.P., Buckley, J.: Inference-based and expectation-based processing in program comprehension. In: Proceedings of the 9th International Workshop on Program Comprehension (2001). https://doi.org/10.1109/WPC.2001.921715
38. Pace, D.: The Decoding the Disciplines Paradigm: Seven Steps to Increased Student Learning. Indiana University Press (2017)
39. Parcell, E.S., Rafferty, K.A.: Interviews, recording and transcribing. In: Allen, M. (Ed.), The SAGE Encyclopedia of Communication Research Methods. Sage Publications, Inc. (2017). https://doi.org/10.4135/9781483381411.n275
40. Patton, M.Q.: Qualitative Research & Evaluation Methods: Integrating Theory and Practice, 4th edn. Sage Publications (2015)
41. Pennington, N.: Comprehension strategies in programming. In: Olson, G.M., Sheppard, S., Soloway, E. (eds.) Empirical Studies of Programmers: Second Workshop, pp. 100–113. Ablex Publishing Corporation (1987)
42. Plowright, D.: Using Mixed Methods: Frameworks for an Integrated Methodology. Sage Publications (2011)
43. Powell, N., Moore, D., Gray, J., Finlay, J., Reaney, J.: Dyslexia and learning computer programming. ACM SIGCSE Bull. **36**(3), 242 (2004). https://doi.org/10.1145/1026487.1008072
44. Rus, V., Akhuseyinoglu, K., Chapagain, J., Tamang, L., & Brusilovsky, P.: Prompting for free self-explanations promotes better code comprehension. In: CEUR Workshop Proceedings, p. 3051 (2021)
45. Saldaña, J.: The Coding Manual for Qualitative Researchers. Sage Publications (2016)
46. Sarkar, A.: The impact of syntax colouring on program comprehension. In: Proceedings of the 26th Annual Conference of the Psychology of Programming Interest Group, pp. 49–58 (2015)
47. Schmidt, A.L.: Effects of experience and comprehension on reading time. Int. J. Man-Mach. Stud. 399–409 (1986)
48. Schröer, M., Koschke, R.: Recording, visualising and understanding developer programming behaviour. In: Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, pp. 561–566 (2021). https://doi.org/10.1109/SANER50967.2021.00066
49. Schwandt, T.A., Lincoln, Y.S., Guba, E.G.: Judging interpretations: But is it rigorous? Trustworthiness and authenticity in naturalistic evaluation. N. Dir. Eval. **2007**(114), 11–25 (2007). https://doi.org/10.1002/ev.223
50. Shopkow, L., Diaz, A., Middendorf, J., Pace, D., Díaz, A., Middendorf, J., Pace, D.: The History learning project "decodes" a discipline: The union of teaching and epistemology. In: McKinney, K. (Ed.) Scholarship of Teaching and Learning in and Across the Disciplines. Indiana University Press (2013)
51. Siegmund, J., Kástner, C., Apel, S., Brechmann, A., Saake, G.: Experience from measuring program comprehension - Toward a general framework. In: Kowalewski, S., Rumpe, B. (Eds.) Software Engineering 2013 - Fachtagung des GI-Fachbereichs Softwaretechnik. GI-Edition - Lecture Notes in Informatics (LNI), vol. P-213, pp. 239–257. Gesellschaft für Informatik e.V (2013)
52. Simon, B., Lopez, M., Sutton, K., Clear, T.: Surely we must learn to read before we learn to write! In: Proceedings of the Conferences in Research and Practice in Information Technology Series, pp. 165–170 (2009)
53. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. Commun. ACM **29**(9), 850–858 (1986). https://doi.org/10.1145/6592.6594

54. Soloway, E., Adelson, B., Ehrlich, K.: Knowledge and processes in the comprehension of computer programs. In: Chi, M.T.H., Glasser, R., Farr, M.J. (eds.) The Nature of Expertise, pp. 129–152. Lawrence Erlbaum Associates (1988)
55. Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. IEEE Trans. Softw. Eng. **SE-10**(5), 595–609 (1984)
56. Soloway, E., Lampert, R., Letovsky, S., Littman, D., Pinto, J.: Designing documentation to compensate for delocalized plans. Commun. ACM **31**(11), 1259–1267 (1988). https://doi.org/10.1145/50087.50088
57. Soloway, E., Lochhead, J., Clement, J.: Does computer programming enhance problem solving ability? Some positive evidence on Algebra word problems. In: Sediel, R.J., Anderson, R.E., Hunter, B. (Eds.) Computer Literacy, pp. 171–201. Academic Press, Inc. (1982). https://doi.org/10.1016/b978-0-12-634960-3.50023-3
58. Srikant, S., Aggarwal, V.: A system to grade computer programming skills using machine learning. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1887–1896 (2014). https://doi.org/10.1145/2623330.2623377
59. Storey, M.A.D., Wong, K., Müller, H.A.: How do program understanding tools affect how programmers understand programs? Sci. Comput. Program. **36**(2), 183–207 (2000). https://doi.org/10.1016/S0167-6423(99)00036-2
60. Von Mayrhauser, A., Vans, A.M.: Industrial experience with an integrated code comprehension model. Softw. Eng. J. **10**(5), 171–182 (1995). https://doi.org/10.1049/sej.1995.0023
61. Wiedenbeck, S., Fix, V., Scholtz, J.: Characteristics of the mental representations of novice and expert programmers: an empirical study. Int. J. Man Mach. Stud. **39**(5), 793–812 (1993). https://doi.org/10.1006/imms.1993.1084
62. Xie, B., Nelson, G.L., Ko, A.J.: An explicit strategy to scaffold novice program tracing. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education, pp. 344–349 (2018). https://doi.org/10.1145/3159450.3159527