Source code comprehension: Decoding the cognitive challenges of novice programmers

by

Pakiso Joseph Khomokhoana

Thesis submitted in fulfilment of the requirements for the degree

Philosophiae Doctor in Computer Information Systems

(PhD Computer Information Systems)

Three-article option

in

The Faculty of Natural and Agricultural Sciences

Department of Computer Science and Informatics

UNIVERSITY OF THE FREE STATE
UNIVERSITEIT VAN DIE VRYSTAAT
YUNIVESITHI YA FREISTATA

Bloemfontein - South Africa

January 2020

Promoter: Prof L Nel

Declaration

I, **Pakiso Joseph Khomokhoana**, hereby declare that the thesis titled '*Source code comprehension: Decoding the cognitive challenges of novice programmers*' is the result of my own independent investigation and that all the sources I have used or quoted have been indicated and acknowledged by means of complete references. I further declare that the work is submitted for the first time at this university/faculty towards the ***Philosophiae Doctor degree in Computer Information Systems*** and that it has never been submitted to any other university/faculty for the purpose of obtaining a degree. I also cede copyright of this product in favour of the University of the Free State.

12 February 2020

……. …….

Signature **Date**

Acknowledgments

“A journey of a thousand miles begins with a single step”

Lao Tzu

Had my promoter not genuinely and intellectually guided me to move step by step on this research journey, would it have been possible for me to complete this thesis? Definitely a big “NO”. So what? She deserves my sincere appreciation, heartfelt gratitude, and unlimited thanks. I will forever be grateful to Prof L Nel – I could not have imagined a better adviser, counsellor, mentor and promoter for my PhD. She was always resolutely available to provide insightful comments and/or ideas; necessary resources; constructive criticism; some words of encouragement during dark days; and any other relevant information for this project to materialise. She used to say, “don’t work yourself too hard”, and in response I would say, “I will try not to”. What I actually wanted to say was, “you don’t know that you are giving me motivation to work harder”. At times she would say, “I never thought you would produce this kind of work”, and I would simply respond by saying, “Thank you”. All the support she provided, coupled with her verbal statements, as well as the ‘warm, but hot’ meetings we had, were some of the key pillars that carried me through this research project.

I am also very grateful to the following:

- The Department of Computer Science and Informatics at the University of the Free State (UFS), not only for an all-embracing welcome and interactions, but also for all the resources available to me while I conducted this research study.
- The UFS Postgraduate School for a series of workshops that were eye-opening and informative.
- The UFS for a tuition-fee bursary allocated to me for each year of the three years I spent at the university.
- All my family members (Joalane, Shaun, Gavin, and parents) – this thesis is dedicated to you all.
- Mrs Elize Gouws for language editing the manuscript; and Mrs S Opperman for language editing an article which forms one chapter of this thesis.

- Third-year students who agreed to take part in a questionnaire survey; instructors who agreed to take part as interviewees in a decoding interview; and the selected students from a third-year class who agreed to take part in the think-aloud interviews. Everyone was of great assistance in collecting data for this research project. Their contributions were immense and cannot go unnoticed, because a research project like this can never exist without data.
- All colleagues and fellow postgraduate students who assisted me with the pilot studies and decoding interviews.
- Prof AC Wilkinson for the critical review and constructive feedback on the three articles forming three chapters of this thesis.
- **Our Almighty Father** for his everlasting and enduring love and for having been good to me throughout this journey (1 Chronicles 16:34).

Table of Contents

List of Figures

List of Tables

Summary

After four decades of research investigations, source code comprehension (SCC) continues to be challenging to undergraduate Computer Science (CS) students. CS instructors, on the other hand, do not generally have any problems to comprehend source code. The Decoding the Disciplines (DtDs) philosophy is based on the premise that each discipline has its own unique set of mental operations. In many cases, these operations have become invisible to instructors, as they tend to perform them automatically based on years of experience. If the nature of these operations is not made explicit to students, it is likely that they will develop learning 'bottlenecks' which could prevent them from mastering key disciplinary practices (such as SCC). Better understanding of the nature of the cognitive processes and related strategies employed by experts during SCC could ultimately be utilised to expose these 'hidden' mental steps.

The overall aim of this study was to explore how a systematic decoding approach can be used to uncover cognitive strategies for efficient SCC by novice programmers. The research findings are presented in the format of three interrelated articles:

Article 1 reports on a study aimed at uncovering common SCC bottlenecks experienced by senior CS students. 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 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 article 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.

Article 2 describes a study that employed decoding interviews, followed by thematic data analysis, to uncover a variety of explicit cognitive processes and related strategies utilised by a select group of experienced programming instructors during a

SCC task. The insights gained were then used to propose a set of mental scaffolding techniques for efficient SCC. It is foreseen that programming instructors could use these techniques as an SCC teaching aid to convey expert ways of thinking more explicitly to their students. Insight into the general cognitive strategies utilised by expert programmers is also an important step towards further exploration of the more detailed step-by-step procedures followed by experts during SCC.

One of the key bottlenecks identified in the CS discipline, relates to students' inability to reliably work their way through the long chain of reasoning necessary to comprehend source code. In an attempt to narrow the existing gap between expert and novice thinking in this regard, Article 3 describes a study in which decoding interviews with five expert programmers (who were also experienced programming instructors) were utilised to systematically deconstruct the explicit mental techniques and reasoning strategies necessary for efficient SCC. Thematic analysis of the mental operations performed by these experts during an SCC activity, led to the identification of 11 key strategies. Knowledge of these strategies as well as the related explicit mental operations were then used to devise a step-by-step framework for efficient SCC. The main purpose of this framework is to create awareness among CS instructors regarding the explicit mental operations required for efficient SCC, and to serve as a source of further research and refinement. Moreover, within the realm of the DtDs philosophy, this framework can 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.

Keywords: Source code comprehension, decoding the disciplines, decoding interview, student-learning bottlenecks, cognitive processes, cognitive strategies, undergraduate programming, Computer Science Education, novice programmers, expert programmers.

Chapter 1 – Introduction

1.1 Background to the study

In the global world of Computer Science (CS), it is well documented that learning to program poses a challenge to many students. As such, several efforts have been undertaken to assist entry-level CS students overcome programming-related challenges. Most of these challenges are rooted in students' inability to effectively and efficiently read, comprehend, and modify source code (Lister et al., 2004; McCracken et al., 2001). This is evidenced by the struggle students encounter when they have to modify source code that they did not write themselves (Mishra & Mohanty, 2012; Singh, Pollock, Snipes & Kraft, 2016; Cimitile, Tortorella & Munro, 1994). Several authors (Perscheid, 2011; Soh, Khomh, Gueheneuc & Antoniol, 2013; Standish, 1984; Tiarks, 2011; Von Mayrhauser, Vans & Howe, 1997) are in agreement that students (as programmers) devote most of their time to the process of reading and understanding source code in order to modify it. This process is commonly referred to as source code comprehension (SCC).

Source code comprehension is widely recognised as central to programming (Bednarik & Tukiainen, 2006; Shaft & Vessey, 1995). It is also regarded as a precondition for any type of modification to occur in a computer program (Alam & Padenga, 2010). In computer programming courses, instructors must address an assortment of programming aspects that could help enhance students' ability to understand source code. These aspects may be the various small challenges of computer programming that, if overlooked, may ultimately hamper the SCC ability of students. Therefore, inherent challenges experienced by instructors in teaching computer programming and difficulties encountered by students in the learning of computer programming are considered next.

1.1.1 Challenges in teaching computer programming

Although teaching is a complex activity, courses in various disciplines are normally taught by instructors who have not received formal training in pedagogy, but who are

experts in the courses they teach. Consequently, these instructors tend to follow methods and strategies that were used on them when they were students (Ambrose, Bridges, DiPietro, Lovett & Norman, 2010). The teaching of computer programming is not an exception to this practice. Hence, computer programming instructors are faced with challenges, including the following:

- Devising instructional strategies that would adequately reach all students (Lahtinen, Ala-Mutka & Järvinen, 2005) due to factors such as high enrolment rates and diversity in students' prior knowledge.
- Retaining and graduating most of the enrolled students, due to the fact that learning to read and write source code is generally considered hard (Eranki & Moudgalya, 2016).
- Using effective pedagogical strategies and methods that will help students to learn programming maximally (Oroma, Wanga & Ngumbuke, 2012; Sentance & Csizmadia, 2016).

If teaching computer programming poses challenges, it can be inferred that programming students also have to deal with discipline-specific challenges.

1.1.2 Challenges in learning computer programming

Of all the students enrolled in computer programming courses, the entry-level students are normally the most challenged (Kinnunen, 2009). Literature (Busjahn & Schulte, 2013; Fisler, 2014; Lee & Ko, 2015; Pope, 2016) commonly refers to entry-level programming students as 'CS1/CS2 students'. One fundamental reason that could be attributed to the fact that CS1/CS2 students are the most challenged, is that they must first learn to 'speak' a new (programming) language. In addition, they also have to face the following challenges:

- Thinking analytically and reasoning logically in solving computer programming problems (Butler & Morgan, 2007; Ismail, Ngah & Umar, 2010).
- Decomposing a problem description into sub-problems, implementing these sub-problems, and putting the pieces together into a complete solution (Lister et al., 2004).

- Translating a manually solved problem into an equivalent computer program (Soloway, Ehrlich & Black, 1983).
- Making a transition from an understanding of separate program statements to the tasks that are to be achieved by groups of statements (Liffick & Aiken, 1996).
- Dividing program functionality into procedures (Piteira & Costa, 2013).
- Understanding programming concepts to be applied in solving problems or in developing computer programs (Lister et al., 2004; Sentance & Csizmadia, 2016).
- Mapping what is in the code or program back into the original software specifications or requirements (*concept assignment problem*) (Biggerstaff, Mitbander & Webster, 1993).

All of the stated challenges could have a negative impact on the SCC abilities of students. If students are unable to fully comprehend and master source code, their software maintenance abilities may be hampered in future. To identify specific challenges with SCC, several techniques have been suggested and used. These include showing source code to students and giving them a task to solve in a controlled environment to determine their level of source code understanding (Siegmund, Kástner, Apel, Brechmann & Saake, 2013); and using think-aloud techniques or protocols (Anderson, Bachman, Perkins & Cohen, 1991).

By applying the aforementioned techniques, differences between strategies used by experienced and novice programmers to understand source code have been identified. These include the fact that experienced programmers pay attention to meaningful areas of the source code and complex statements, while novice programmers visually concentrate on the comments and comparisons (Busjahn, Schulte & Busjahn, 2011; Crosby & Stelovsky, 1990; Von Mayrhauser & Vans, 1995b). The experienced programmers also need little working memory when solving SCC-related problems, because they readily identify the procedural nature of the source code – which is not the case with novice programmers (Wiedenbeck, Fix & Scholtz, 1993).

In close examination of such strategies, deficiencies inherent in novice programmers are exposed. To help them overcome these challenges, a myriad of strategies and/or techniques have been suggested and used. These include using programming plans (stereotype source code fragments that represent known action sequences) (Davies, 1990; Gilmore & Green, 1988; Green & Navarro, 1995; Rist, 1986; Soloway & Ehrlich, 1984); developing tools with search capabilities (Singer, Lethbridge, Vinson & Anquetil, 1997); syntax highlighting (Sarkar, 2015); cognitive load reduction (Sweller, 1988; Sweller, Van Merrienboer & Paas, 1998); pair-programming (Braught, Wahls & Eby, 2011; Cronje, 2013); bottom-up comprehension strategy (Basili & Mills, 1982; Shneiderman, 1976; Shneiderman & Mayer, 1979); and top-down comprehension strategy (Brooks, 1999).

In addition to the above-mentioned strategies, another angle that could be considered to help CS students (as novice programmers) understand source code better, is the cognitive perspective. Reasons for considering this perspective are twofold: First, SCC is regarded as a highly cognitive task (Praveen, 2016). Second, for students to better comprehend source code, they need to acquire a mental model of the structure and function of the source code (Bednarik & Tukiainen, 2006). The mental model refers to students' understanding of the source code during the comprehension process (Letovsky, 1987).

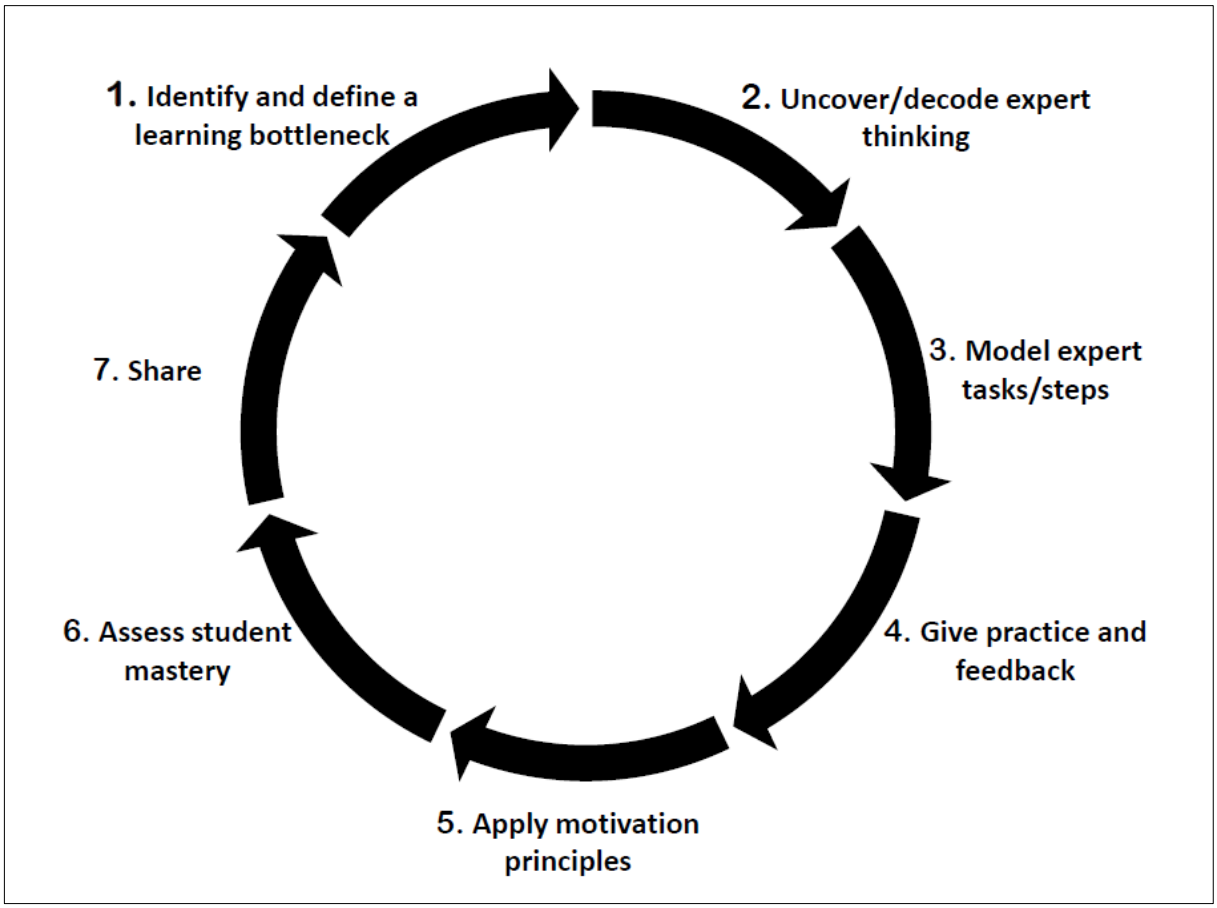
Using the cognitive perspective, several studies have attempted to provide insights regarding the strategies used by programmers of various expertise levels, including novice programmers, to comprehend source code (Burkhardt, Détienne & Wiedenbeck, 2002; Ko & Uttl, 2003; Littman, Pinto, Letovsky & Soloway, 1987; Letovsky, 1987; Soloway, Lampert, Letovsky, Littman & Pinto, 1988; Von Mayrhauser & Vans, 1996). These investigations have mostly been based on verbal protocols (*capturing the thought-processing*). Similar investigations based on non-verbal protocols used eye-movement tracking (Bednarik & Tukiainen, 2006; Crosby & Stelovsky, 1990) and neuro-imaging methods (Siegmund et al., 2014). Some other interventions were to carry out simulations of the source code that is being maintained (Soloway, 1986).

Most of the previous research studies that considered the mental processes involved in understanding source code, employed cognition models as their theoretical lenses. Considerable research (Basili & Mills, 1982; Brooks, 1983; Letovsky, 1987; Littman et al., 1987; Shneiderman & Mayer, 1979; Soloway, Adelson & Ehrlich, 1988) on developing these models has been conducted from the late 1970s throughout the 1980s. Von Mayrhauser and Vans (1993; 1995b) even developed an integrated code comprehension model which is based on many of these cognition models.

However, an approach from a different theoretical lens could be considered. Decoding the Disciplines (DtDs) is a framework that could be usable due to its multidisciplinarity and pedagogical nature. This seven-step framework was devised by Joan Middendorf and David Pace (Middendorf & Pace, 2004). Within this framework, the challenges experienced by students are normally referred to as bottlenecks. These are defined as specific points where the learning of a significant number of students gets interrupted (Diaz, Middendorf, Pace & Shopkow, 2008; Middendorf & Pace, 2004). Bottlenecks usually come to the fore when students do not have the knowledge of how to deal with a situation or problem, and hence resort to unsuitable strategies (Pace, 2017a).

DtDs presents an all-embracing framework within which these bottlenecks can be addressed. One of the fundamental principles of this framework is that each discipline has its own unique ways of thinking (Middendorf & Pace, 2004). Students who are unable to master the required ways of thinking are unlikely to succeed in their higher-level studies. Within the DtDs framework, instructors are therefore encouraged to identify discipline-specific learning bottlenecks that could prevent students from mastering the basic disciplinary ways of thinking (Step 1). After identifying the bottlenecks, the crucial mental operations required to overcome such bottlenecks are uncovered with the assistance of disciplinary experts (Step 2). These operations are then modelled explicitly to students (Step 3). After this, instructors create opportunities for students to practise these operations or skills and get feedback on their mastery of the skills (Step 4). In the process, motivational strategies or principles are applied to assist students in effectively learning the imparted skills (Step 5). Eventually, an assessment is made of how well the undertaken efforts help students to master the intended learning content (Step 6). As part of the final step (Step 7), instructors are

encouraged to share (formally or informally) their experiences from this process (Middendorf & Pace, 2004; Pace, 2017a). The seven distinct steps of the DtDs framework, as described above, are presented in Figure 1.1. Despite the recent uptake in decoding research conducted in other disciplines (Shopkow, 2017; Verpoorten et al., 2017), limited information regarding DtDs research in the CS discipline is available in the public domain.



[Source: Middendorf & Pace, 2004, p. 3)

Figure 1.1 – Seven steps of the DtDs framework

1.2 Problem statement

Despite numerous efforts undertaken since the early 1980s (Siegmund, 2016) to assist students in improving their SCC skills and generally performing well in programming courses, many CS students continue to struggle with SCC. This is evidenced by the findings of several studies (Lister et al., 2004; Mccartney, Boustedt, Eckerdal, Sanders & Zander, 2013; McCracken et al., 2001; Utting et al., 2013; Whalley et al., 2006). Most of these studies have reported students' struggles when

they have to read, interpret, and/or comprehend given pieces of code. The continuation of the struggle can be attributed to the fact that, of all the initiatives undertaken, there is no consensus among researchers, educational developers, and instructors on how to address this issue best. This happens irrespective of the seemingly better strategies used by programming experts themselves.

To some extent, most CS instructors can be regarded as experts in their discipline. Despite their 'expert' skills, these instructors often struggle to explain source code and its underlying concepts to their students (as novice programmers) in such a way that these novices understand it in the same way they (the instructors) do. The problem emanates from the constantly confirmed hypothesis termed 'expert blind spot' (Grossman, 1990; Nathan & Petrosino, 2003; Shulman, 1986). This hypothesis was developed from the works of Nathan and his colleagues (Nathan & Koedinger, 2000; Nathan, Koedinger & Alibali, 2001). It states that instructors:

> *"with advanced subject-matter knowledge of a scholarly discipline tend to use the powerful organising principles, formalisms and methods of analysis that serve as the foundation of that discipline as guiding principles for the students' conceptual development and instruction, rather than being guided by knowledge of the learning needs and developmental profiles of novices"* (Nathan & Petrosino, 2003, p. 906).

Therefore, it can be deduced from this hypothesis that an expert blind spot refers to vital operations that have become so natural to the experts that they omit crucial steps when explaining concepts and procedures to others.

Hence, these expert blind spots, coupled with ways in which instructors teach source code comprehension, could lead to students developing mental blocks when it comes to SCC. It may, therefore, be essential to pay further attention to the cognitive perspective of this problem. Coupling this perspective with the fundamental elements of students' thinking and doing to facilitate effective learning and understanding, as suggested by Middendorf and Pace (2004), it is essential that novice programmers are:

- Made aware of the intrinsic cognitive processes or steps that experts follow while comprehending source code. This can be used as a comprehension strategy to obtain new knowledge, as suggested by Von Mayrhauser and Vans (1995a; 1995b);
- Engaged in practising the models (being motivated in the process); and
- Provided with effective feedback in order to see how they can better comprehend source code.

It is, for example, strongly believed that students must think and do for learning to happen (Herbert, as cited in Ambrose et al., 2010, p. 1). Students are more likely to remember what they do than what they are being told to do. Accordingly, students should be engaged in the modelling and practising of the models in order for them to learn. On feedback issues, Brookhart (2008) alludes to various factors such as timing, amount (*content*) of feedback, mode, and audience as fundamental in providing feedback to students.

If CS1/CS2 instructors are able to put appropriate pedagogical interventions (*in the form of a pedagogical process*) in place and effectively tap into students` cognitive blocks, this could help students to systematically overcome the identified blocks. As part of this process, students should be helped to improve and/or refine their mental actions in understanding source code in a classroom setting (*naturally occurring context*) (Lewin, 1951; Sagor, 2000; Stringer, 2014).

1.3 Aim and research questions

Based on the problem statement (as described in Section 1.2), this study is set out to explore how a systematic decoding approach can be used to uncover cognitive strategies for efficient SCC by novice programmers. In order to address this aim, the research study attempted to answer the following main research questions:

RQ1: What are the SCC challenges experienced by novice programmers?

RQ2: How can a systematic decoding approach be used to devise cognitive strategies that could be used to address these challenges?

For the purpose of answering the aforementioned two main research questions, the following nine subsidiary research questions were formulated:

- **Subsidiary research questions – (guiding the literature review)**

  SRQ1: What are the strategies that programmers (novices and experts) follow during the SCC process?

  SRQ2: What are the challenges that influence the development of novice programmers' SCC skills?

  SRQ3: How do cognitive and metacognitive practices influence SCC?

- **Subsidiary research questions – (directing the empirical investigations)**

  SRQ4 (a): What are the major SCC difficulties experienced by senior CS students?

  SRQ4 (b): How can knowledge of these difficulties be used to identify SCC bottlenecks that should ideally be addressed in introductory programming courses?

  SRQ5 (a): What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC?

  SRQ5 (b): What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students?

  SRQ6 (a): What are the explicit mental strategies (techniques and reasoning) that CS experts employ while comprehending source code?

  SRQ6 (b): How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards narrowing the gap between expert and novice thinking with regard to efficient SCC?

1.4 Research design and methodology

The research design of this study was based on the seven-step DtDs framework. Within this framework, an integrated-methods research approach based on

Plowright's (2011) Frameworks for an Integrated Methodology (FraIM) was adhered to. The study consisted of three phases to distinguish between the different sources of data (cases). Phase 1 was aimed at identifying specific senior CS students who were having difficulties in comprehending short pieces of source code. Phase 2 was aimed at uncovering specific points or places where senior students were experiencing SCC difficulties, with the ultimate goal of identifying common and useful SCC bottlenecks. Phase 3 was aimed at uncovering the explicit nature of steps and strategies that programming experts would follow in order to accomplish the tasks associated with one of the student-learning bottlenecks identified in Phase 2. The specific details of how each of these three research phases unfolded, are provided in Chapter 3 (see Section 3.4).

1.5 Research Contexts

As part of the FraIM, Plowright (2011) suggests that there are various contexts that could impact the choice of topic in any research study. In the following sub-sections, some background information is provided regarding four specific contexts that are of relevance to this study.

1.5.1 Professional context

The researcher is a full-time lecturer at an institution of higher learning in Lesotho. He started teaching in 2014 after working in the Department of ICT Services of the same institution since 2003. He teaches Information Systems-related courses, including introduction to programming, website development, and information systems in a business environment. The choice of the research topic was influenced by both the researcher's own experiences as a CS student and his experiences in teaching programming courses to a diverse group of students. As an undergraduate student, the researcher perceived programming to be a difficult subject. However, upon moving to a South African institution of higher learning to pursue his postgraduate studies (Honours), he was required to take additional undergraduate programming modules to come on par with the other Honours students. While studying these modules, he was engulfed in a student-centred and welcoming environment. The teaching aids used in these modules had emotional connection with the students, and most of the examples shared were meaningful and made sense to students. As a result, the

researcher performed well in these undergraduate programming modules. This led him to change his perception about programming being difficult. While teaching programming, the researcher observed a lot of students struggling (e.g. syntax, semantics, conceptualisation, code explanation, debugging, and tracing). This happened irrespective of the fact that the same strategies were used as those used during his Honours studies.

1.5.2 Organisational context

This research study was conducted at a selected South African higher education institution. The novice programmers used as student participants in this research study were senior CS students. These students were all enrolled for a three-year Bachelor of Science degree majoring in Information Technology. As part of this degree, students must complete a number of CS modules, together with modules from at least one other specialisation field (Business Management, Chemistry, Mathematical Statistics, Mathematics or Physics). In their first two study years, these students take CS modules that focus on building foundational knowledge regarding programming in C# (introductory and advanced), web development, computer hardware, databases, human-computer interaction, and software design principles. In the third year, these students must complete modules in advanced databases and computer networks, as well as two other modules (Internet programming and Software Engineering) where they have the opportunity to combine knowledge gained from previously studied CS modules.

1.5.3 National context

In the past few years, higher education institutions in South Africa have been faced with challenges such as burgeoning numbers of students enrolling in various programmes (i.e. massification in higher education) (Council on Higher Education, 2016; Jansen, 2003). These students come from various settings in terms of ethnicity, professional and personal background, socio-economic status, language, and sexual orientation. It is therefore evident that these students are not all academically equally prepared for the higher learning environment, which is characterised by lots of pressure for adaptation, independence, and performance. Irrespective of the aforementioned challenges, the institutions are pressurised to increase student throughput (Council on Higher Education, 2016).

1.5.4 Theoretical context

Source code comprehension has been identified as one of the main difficulties that novice programmers continue to experience (Cunningham, Blanchard, Ericson & Guzdial, 2017; Lister et al., 2004). Computer Science instructors (as experts in the field) are able to comprehend source code. However, they struggle to help novice programmers understand source code in the same way they do. As indicated in the discussion of the problem statement, expert blind spots (Grossman, 1990; Nathan & Petrosino, 2003; Shulman, 1986), coupled with ways in which instructors teach source code, can lead to students developing mental blocks when it comes to SCC. As such, there is a need to identify the specific SCC challenges experienced by novice programmers in an educational context. Since CS instructors (as experts) do not typically have problems to comprehend source code, there is a need to uncover (or decode) the explicit mental operations (techniques and reasoning strategies) they follow during SCC. Knowledge of these explicit cognitive strategies and/or steps could then be used to identify specific strategies for efficient SCC that instructors can use in teaching students to comprehend source code more efficiently.

1.6 Scope of research

In this study, the DtDs framework was adapted to create an enabling environment to conduct the empirical investigation and to ultimately answer the two main research questions of the study. DtDs adaptations are supported by Middendorf and Pace (2004), who indicate that the DtDs' steps are neither 'mechanical' nor 'deterministic'. The framework is deemed suitable because it is pedagogical in nature (King, Linkon & Middendorf, 2013). Furthermore, the framework helps to answer a series of questions that instructors can ask themselves as they try to understand how their students think and learn in their specific disciplines (Middendorf & Pace, 2004). Based on this scope as well as the theoretical framework (as outlined in Section 1.5.4), Figure 1.2 provides a conceptual framework for the research study that also shows the link with the empirical part of the investigation. Given the amount of work involved, rigour applied in doing the work, and large amounts of data collected while following the DtDs framework, this research study only focused on Step 1 and Step 2 of the framework.

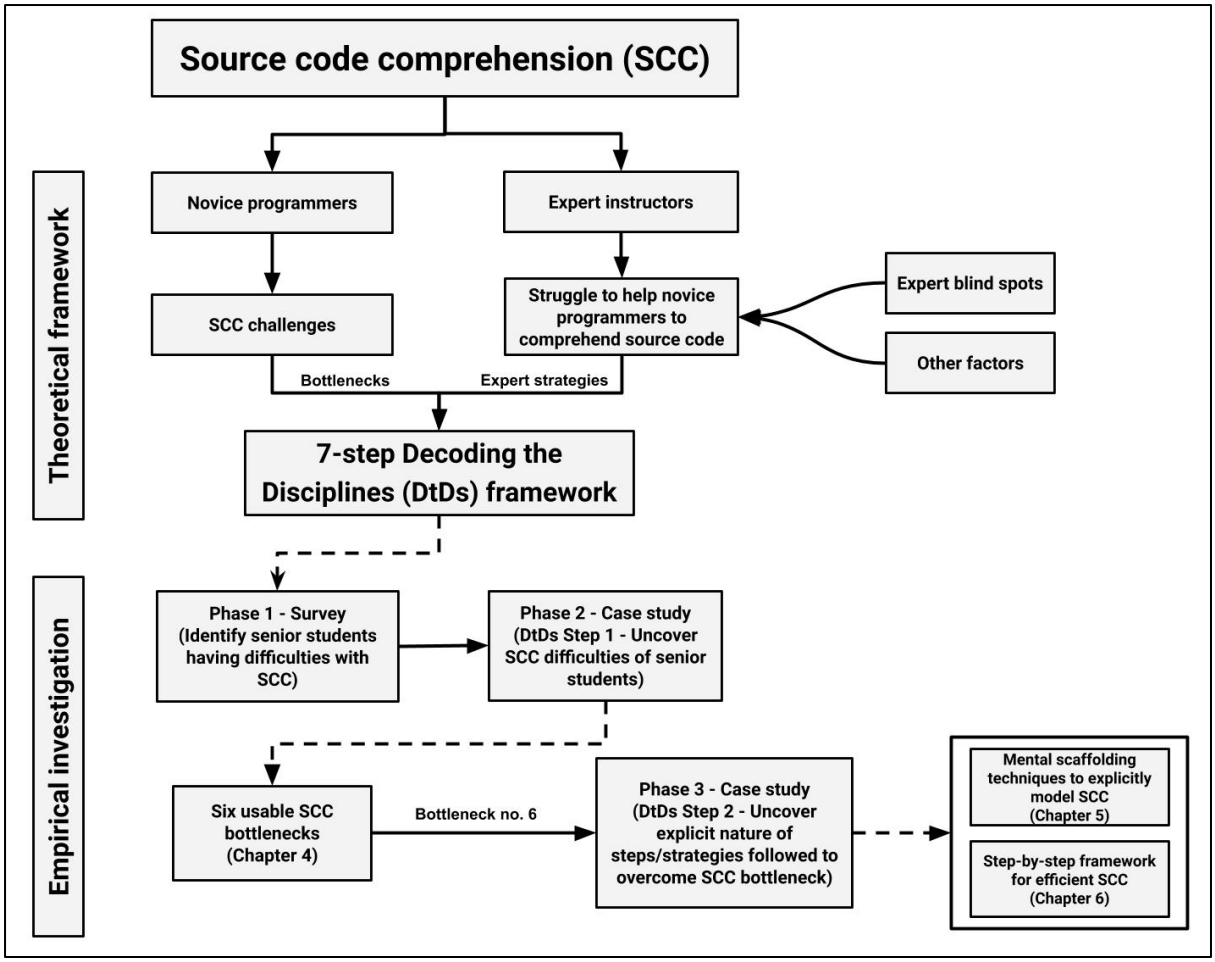


Figure 1.2 – Conceptual framework for this study

For the empirical investigation, Phase 1 and Phase 2 of the study were conducted as part of DtDs framework Step 1, while Phase 3 was conducted as part of DtDs framework Step 2. The six usable SCC bottlenecks identified as part of Phase 2 are reported in Chapter 4. Phase 3 only focused on addressing one of the six identified bottlenecks. The main outcomes of Phase 3 were a set of mental scaffolding techniques for the explicit modelling of SCC (as reported in Chapter 5) and the proposed step-by-step framework for efficient SCC (discussed in Chapter 6). It is believed that the implementation of these techniques, and the execution of this framework, could be instrumental in assisting instructors to help novice programmers comprehend source code the same way they do, hence addressing the cognitive challenges that students experience. The entire investigation was conducted within the field of Computer Science Education. The central issue was to use a systematic decoding approach to devise a range of cognitive strategies that could be used to address the specific SCC challenges experienced by novice programmers.

1.7 Presentation of the thesis

This thesis report consists of seven chapters.

In this chapter, Chapter 1, a brief introduction of the research study discussed in the thesis is provided. The discussion presents preliminary insights from the literature on which the study was grounded. This literature indicates the theoretical direction taken by the study.

Chapter 2 presents a detailed review of related contemporary literature. The discussions in this chapter specifically focus on factors that could influence the SCC ability of programmers, strategies followed by programmers during the SCC process, and the influence that cognitive practices can have on SCC.

Chapter 3 provides a discussion of the research design and methodology of this study, as well as the theoretical underpinnings of the theories used for the selected research design and methods. This chapter also provides a detailed discussion of how the research study unfolded in the process of finding answers to the stated research questions, together with the subsidiary research questions as explained in the introductory chapter. Issues related to trustworthiness and ethical considerations are also addressed in this chapter.

The research findings of the study are presented in the format of three articles included as Chapters 4, 5 and 6 of this report. As per university regulations for the ‘thesis by articles’ format, the main criterion for each article is that it must either be a ‘published article’ or a ‘publishable manuscript’. As such, each article represents a standalone document without any cross-references to other parts of the report. Each article is also formatted according to the guidelines of the specific publication for which it was prepared.

Chapter 4 presents Article 1, titled: *Decoding source code comprehension: Bottlenecks experienced by senior Computer Science students*. Set within the DtDs paradigm, this paper reports on an investigation aimed at identifying the major SCC difficulties experienced by senior CS students. The identified difficulties, together with

information from other sources, were used to formulate six usable SCC bottlenecks. These bottlenecks point to student-learning difficulties that should ideally already be addressed in introductory CS courses.

Chapter 5 presents Article 2, titled: *Decoding the explicit cognitive strategies of expert instructors: Mental scaffolding techniques for efficient source code comprehension*. Set within the DtDs paradigm, this paper reports on an investigation aimed at identifying the cognitive processes and related cognitive strategies that expert programmers follow during the SCC process. The knowledge of the identified strategies was used to formulate a set of mental scaffolding techniques for efficient SCC. Programming instructors could use these techniques as an SCC teaching aid to convey expert ways of thinking more explicitly to their students.

Chapter 6 presents Article 3, titled *Narrowing the gap between expert and novice thinking: A step-by-step framework for efficient source code comprehension*. Set within the DtDs paradigm, this paper reports on an investigation aimed at identifying the explicit mental operations (techniques and reasoning strategies) that expert programmers employ while comprehending source code. Insights into these strategies were used in the development of a framework for efficient SCC. This framework is aimed at creating awareness among CS instructors regarding the explicit mental operations required for efficient SCC. It 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.

Chapter 7 outlines the conclusions of this study relating to the main and subsidiary research questions. This includes a discussion of how the research questions were answered, the presentation of the main findings and the contributions of the study, its limitations and recommendations for future research.

Other documents related to this study and the various research activities are included as appendices at the end of this thesis.

Chapter 2 – Theoretical Background

2.1 Introduction

Given the format of this thesis, each of the three articles (as presented in Chapters 4, 5 and 6) includes a section that considers relevant literature. While guarding against unnecessary duplication, it was deemed necessary to also provide a wider conceptual and theoretical basis upon which the remainder of this thesis builds. This chapter therefore presents an overview of three key concepts. First, the general strategies that can be used to comprehend source code are examined. In the course of this examination, the different strategies used by novices and experts are compared. The second section considers three challenges that could influence the development of a novice programmer's SCC skills. Lastly, in the light of the teaching and learning focus of this study, the third section considers relevant cognitive and metacognitive practices and examines the relation between these practices and SCC.

2.2 Source code comprehension strategies

Source code comprehension refers to the process of reading, interpreting, and understanding pieces of source code that make up an entire computer program (Busjahn & Schulte, 2013; Lister et al., 2004; Lister, Simon, Thompson, Whalley & Prasad, 2006; Maalej, Tiarks, Roehm & Koschke, 2014). Numerous attempts have been made to describe and classify the general strategies used by programmers to comprehend pieces of source code (Fitzgerald, Simon & Thomas, 2005; Lister et al., 2004; Xie, Nelson & Ko, 2018). The underlying philosophy of the DtDs paradigm is that each discipline has its unique ways of thinking that instructors should teach their students from early on (Middendorf & Pace, 2004). This also applies to the discipline-specific skill of SCC. In the absence of more explicit knowledge regarding the exact mental processes followed by programmers to efficiently comprehend pieces of source code, it would therefore be impossible to accurately model these ways of thinking to students (see Step 3 of the DtDs framework as presented in Figure 1.1). Knowledge of the general SCC strategies used by programmers (both novices and experts) can, however, serve as a starting point in uncovering the SCC learning

bottlenecks experienced by students (as part of Step 1 of the DtDs framework) as well as the explicit mental processes required for efficient SCC (in Step 2).

With regard to general SCC strategies, traditional taxonomy refers to ‘bottom-up’ and ‘top-down’ as well as various combinations of these strategies (Brooks, 1983; O’Brien, 2003; Pennington, 1987a; Shneiderman, 1980; Von Mayrhauser & Vans, 1995b). With a bottom-up strategy, a programmer approaches the comprehension process by first considering the lower-level structures, then the intermediate structures, and finally the higher-level structures of the source code (Pennington, 1987a; Shneiderman, 1980). When following this approach, a programmer first reads and understands the individual lines of source code and information relating to procedure. Second, the lines of code are grouped into parts that have meaning (chunking). Lastly, these chunks are grouped to form an understanding of how the source code functions (Pennington, 1987b; Shneiderman & Mayer, 1979). The top-down strategy can be regarded as an inverse of the bottom-up strategy, where the programmer starts with the higher-level structures and then works towards the lower-level structures (Brooks, 1983). This means that the programmer first develops hypotheses about the source code being studied. Beacons are then used to evaluate (verify) and refine the initial hypotheses while interacting with the source code (Basili & Mills, 1982; Détienne, 1990; Soloway, Ehrlich & Bonar, 1982). Beacons are defined as knowledge of the source text structure from which a programmer can identify common source code features that act as a signpost that there is an occurrence of certain structures or operations (Brooks, 1983). For both of these strategies, the mentioned steps are repeated as and when necessary until the programmer is able to either partially or fully comprehend the source code under examination (Détienne, 1990; O’Brien, 2003).

Although these models share some common elements, the main difference, however, is that the bottom-up strategy is suitable for situations where programmers are unfamiliar with the domain (O’Brien, 2003), while the top-down strategy requires programmers to utilise domain knowledge to develop their initial hypotheses about the code (Brooks, 1983). It is also highly unlikely that a programmer will exclusively rely on only one of these strategies (O’Brien, 2003). Instead, Von Mayrhauser and Vans (1997) suggest that programmers rather use one of these as their predominant strategy (a subconscious decision based on their level of domain knowledge) and then

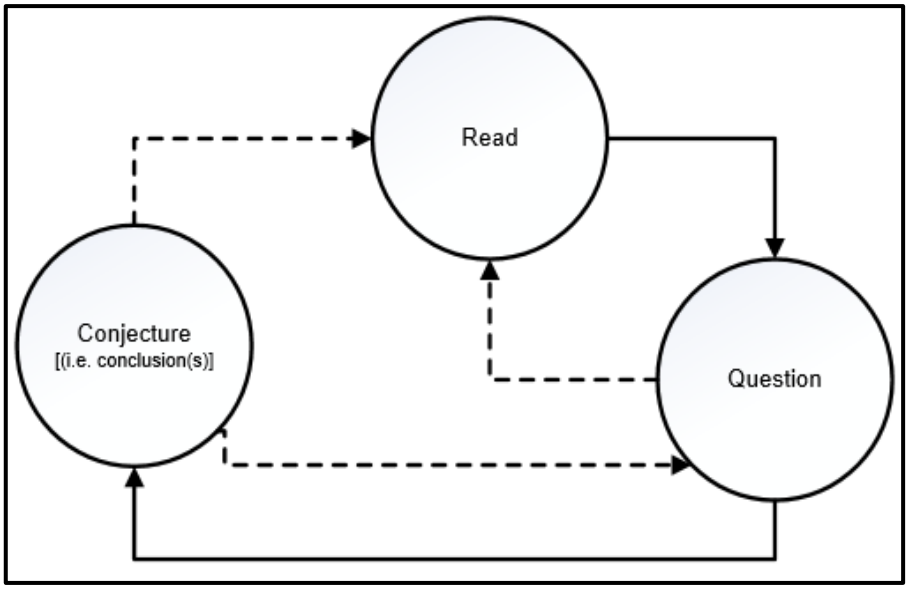
follow an opportunistic approach (Letovsky, 1987) where they switch with ease between strategies as more information becomes available. When a programmer switches between strategies, it might also include elements that are not necessarily part of either the bottom-up or top-down approaches. A number of researchers have attempted to name and describe these 'opportunistic' strategies used by programmers to comprehend source code.

When following a *knowledge-based comprehension strategy* (Letovsky, 1987), programmers use their experience and expertise, including syntactic knowledge and existing and/or newly acquired knowledge, about a problem domain during the comprehension process. Depending on circumstances, the programmer may apply either bottom-up or top-down reasoning. This strategy is considered more applicable and useful for experienced programmers than for novices (Letovsky, 1987; Stan Letovsky & Soloway, 1986; Storey, Fracchia & Muller, 1999).

With a *systematic comprehension strategy* (also known as the *control flow-based strategy*), a programmer reads the source code text in detail and traces through the control flow and data flow. The objective is to gain a global understanding of the source code in order to successfully complete the given SCC task (Littman et al., 1987). As programmers read the source code, they consult associated documentation and perform the necessary simulations. These simulations are strategies that programmers use to uncover the unwanted causal interactions between the various components of the source code that is being examined (Soloway, 1986). The interactions are produced by the dynamic aspects of the source code. An advantage of this strategy is that correct augmentations to the source code are highly likely to be made, because the causal relationships contained in the delocalised plans are identified and studied in adequate detail. Letovsky and Soloway (1986) define delocalised plans as programming plans whose parts are located in non-contiguous parts of the source code. Although it may be realistic to systematically work through short programs, this is not feasible with large programs (Soloway et al, 1988).

Another approach is the *micro comprehension strategy* (Letovsky, 1987), where a programmer uses inquiry episodes. These are activities or groups of activities in which a programmer follows the comprehension search cycle model depicted in Figure 2.1.

Programmers read the given source code and then develop questions that (when answered) will help to enhance their understanding of the source code. With the information obtained from reading the source code and developing questions, programmers make small conclusions about their understanding of the source code. During question development, the programmer can go back to read the source code again. Even when making small conclusions about source code understanding, the programmer is allowed to revisit the development stage of the questions. If they are satisfied with their understanding, programmers stop at the conjecture stage, hence the small conclusions become the final conclusion. Otherwise, they repeat the process and the small conclusions already made will be revised accordingly. The whole process is grounded in the delocalised plans that exist within the pieces of source code in question (Letovsky, 1987; Storey et al., 1999).



[Source: Adapted from Letovsky (1987, p. 327)]

Figure 2.1 – Comprehension Search Cycle Model

With the *as-needed comprehension strategy*, programmers use their experience to identify and only focus on parts of the source code that they think are relevant to the current SCC task (Adelson & Soloway, 1985; Littman et al., 1987; Sillito, De Volder, Fisher & Murphy, 2005). This strategy is also known as an *isolation strategy* (Nanja & Cook, 1987) or *opportunistic relevance strategy* (Koenemann & Robertson, 1991). One advantage of this strategy is that if a programmer identifies appropriate parts of the source code intrinsically relevant to the given comprehension task, it may reduce

the time needed to complete the task. Filtering out source code locations irrelevant to what the programmer wants to achieve will also save time. This strategy is, however, more prone to errors because causal interactions within the source code are not studied in sufficient detail (Soloway et al., 1988).

When following an *integrated comprehension strategy*, a programmer develops code comprehension by switching between the three main strategy categories (bottom-up, top-down and opportunistic) as and when the need arises during the comprehension process (Von Mayrhauser & Vans, 1993; 1995b). This strategy is different from Letovsky's (1987) reasoning in that if the top-down reasoning is used and the programmer wants to change to the bottom-up reasoning, the top-down journey must either be completed or the reasoning must be completely discarded. The same is true when the programmer starts with bottom-up reasoning (Storey et al., 1999).

2.2.1 General reflection on the nature of SCC strategies

Even if one is aware of the processes involved in all of the above-mentioned SCC strategies, it is impossible to predict which comprehension strategy or combination of strategies a programmer would use in a given SCC-related task. Source code comprehension is considered hard and time consuming (Maalej, Tiarks, Roehm & Koschke, 2014). As such, even professional programmers avoid deep understanding of the source code as long as they can achieve their comprehension goals without having to comprehend everything intensely (Maalej et al., 2014). Some authors (Brandt, Guo, Lewenstein, Dontcheva & Klemmer, 2009; LaToza, Garlan, Herbsleb & Myers, 2007) indicate that using the minimum effort possible to maximise outcome is applicable to various strategies that programmers use to comprehend source code. Applying minimum effort with the objective to get the maximum outcome possible is the philosophy underlying Carroll's (2003) minimalist theory.

Source code comprehension is also cognitive in nature (Praveen, 2016) and therefore requires a lot of mental effort (Maalej et al., 2014). This implies that it is never easy to predict or know what a person is thinking, unless they share their thoughts. In the specific context of Maalej et al.'s (2014) study, programmers were found to comprehend source code by asking questions and answering them, as well as developing hypotheses and testing them. Their findings were consistent with the

results of several other authors (Brooks, 1983; Ko & Myers, 2004; Letovsky & Soloway, 1986; Von Mayrhauser, Vans & Lang, 1998).

Understanding the intricacies of an individual's SCC process is further complicated by all the additional tools and practices that programmers have at their disposal to support or facilitate their chosen SCC strategy. Given the cognitive nature of the SCC process (Praveen, 2016), programmers have been shown to use various artefacts to reflect their mental models and record knowledge. Maalej et al. (2014) found that programmers use notes, while Lister et al. (2004) found them to be using doodles and walkthroughs. Doodles are drawings, calculations, and annotations that programmers create as they work through a given piece of code in order to ultimately establish what the output would be if executed (Lister et al., 2004). 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*" (Jeffries, 1982, p. 12). Additionally, programmers have a tendency to utilise the source code itself rather than associated documentation. Maalej et al. (2014) found this tendency to be consistent with the findings of LaToza et al. (2007), who realised the importance of people gaining knowledge from the actual reading of the code compared to reading the text that explains what the code does. Another potential reason for this tendency is that documentation is rarely available. In instances where the documentation is available, it is time consuming to use it to figure out how the code works (Maalej et al., 2014).

An individual's choice of SCC strategy can also be influenced by personal preferences and circumstances. An SCC study by Maalej et al. (2014) reveals that, in practising code comprehension, programmers base themselves on the context, and have a tendency to follow pragmatic comprehension strategies. This means that programmers deal with comprehension in a realistic way that makes sense to them, and they are mostly guided by practical considerations instead of theory (Holmes & Walker, 2012). It has also been shown that programmers do not necessarily want to comprehend source code; instead, they just want to complete their tasks (Kim, Bergman, Lau & Notkin, 2004; Maalej et al., 2014; Singer et al., 1997). This may dictate that programmers ignore the SCC strategies developed by researchers and

practitioners, unless they are subjected to conditions that compel them to utilise such strategies.

Maalej et al. (2014) also established that there is a gap in the perception of SCC between programmers (in practice) and researchers. One main reason attributed to this gap is that researchers come up with strategies that may be too abstract, complicated, or not relevant for application in the software industry (Singer, 2013). Consequently, such strategies may be even less relevant in an educational context. Given that programmer experience has also been identified as having a significant impact on the choice of an SCC strategy (Maalej et al., 2014; Singer et al., 1997), it is necessary to consider how the SCC strategies used by novice and expert programmers differ. The different ways in which novices and experts think about and perform discipline-specific tasks (such as SCC) is one of the main reasons why students tend to develop mental blocks in their learning (Middendorf & Pace, 2004). This is the kind of problem that can be addressed through application of the DtDs framework.

2.2.2 Novice versus expert comprehension strategies

In the past 40 years, numerous studies have been conducted to compare the general SCC strategies used by novice and experienced programmers. Soloway and Spohrer (2013) point out that it takes approximately 10 years for a novice programmer to move on the continuum from a novice to becoming an expert. In general, the results of these studies indicate that novices tend to use bottom-up-based comprehension strategies, while experienced programmers are more likely to use strategies that favour a top-down approach. The identified major differences as well as some discovered similarities between the approaches used by experienced and novice programmers to read, interpret, and understand source code can be summarised as follows:

- In the initial stages of SCC, both experienced and novice programmers follow similar overall strategies, but their strategies differ later on (Jeffries, 1982; Nanja & Cook, 1987; Gugerty & Olson, 1986; Widowski & Eyferth, 1986).
- Experienced programmers use their experience, syntactic knowledge, and knowledge of a problem domain (*knowledge-based strategy*), while novice programmers read source code line-by-line (Letovsky & Soloway, 1986).

- Experienced programmers focus only on reading source code relating to a particular task at hand (*as-needed-based strategy*), while novice programmers focus on all elements of the source code (Littman et al., 1987; Soloway et al., 1988).
- Experienced programmers use a semantic approach (*reliance on functionality*), while novice programmers are driven by how a program works syntactically rather than what a program does semantically (*semantic versus syntactic approach*) (Adelson, 1981; 1983).
- Experienced programmers are more affected by violations in the rules of discourse in a piece of source code than novice programmers (Soloway & Ehrlich, 1984; Soloway, Lochhead & Clement, 1982).
- Both experienced and novice programmers pay least attention to the keywords in the source code's text (Crosby & Stelovsky, 1990).
- Experienced programmers do better than novices in situations where they have to recall meaningful source codes. However, both do equally well where they must recall source codes that are not well designed (Adelson, 1984; McKeithen, Reitman, Rueter & Hirtle, 1981; Schmidt, 1986).
- Experienced programmers link parts of the source code to the problem domain (*cross-referencing strategy*), which is unusual for novice programmers (Pennington, 1987b).
- Experienced programmers do not study source code line-by-line like novices (Letovsky, 1987). They search instead for key lines (*beacons*) (Brooks, 1999).
- It is easier for experienced programmers to realise when they have to change or adapt their comprehension strategy – especially as the result of discovering an anomaly in the source code or when the requested task has some inherent special needs (Storey, Wong & Müller, 2000).
- Experienced programmers resort to using other skills *(*e.g. *simulations of the source code* to make its dynamic properties explicit) when their higher-order skills do not help them in understanding the source code. This is not typical with novice programmers (Soloway, 1986).

- Experienced programmers tend to use a source code reading strategy that follows the order in which the source code would be executed (Jeffries, Turner, Polson & Atwood, 1981; Mosemann & Wiedenbeck, 2001; Nanja & Cook, 1987). Novice programmers, on the other hand, tend to read code line-by-line as if they are following a cookbook recipe (Saha & Ray, 2015). Experts have, however, been observed to revert to a line-by-line strategy in cases where they were not familiar with a programming system (Ko & Uttl, 2003).
- Experienced programmers have developed the ability to identify the most effective and appropriate strategy to follow for a given comprehension task (Storey et al., 2000). Novices tend to use a guessing or trial-and-error strategy to ultimately arrive at an acceptable comprehension strategy (Nanja & Cook, 1987).
- Experienced programmers form mental models in terms of abstractions, while novice programmers' models are formed in terms of source code statements or sequentially (Corritore & Wiedenbeck, 1991; LaToza et al., 2007; Wiedenbeck, Ramalingam, Sarasamma & Corritore, 1999).
- Experienced programmers make little use of their working memory during SCC because they are able to readily identify the procedural nature of the source code (Wiedenbeck et al., 1993). Novice programmers need more mental attention (Wiedenbeck, 1985).

From the above comparisons, it can be deduced that the knowledge of programming experts is more organised than that of novice programmers. This knowledge is activated when programmers engage their thinking during the comprehension process (Teasley, 1993). By engaging their thinking, programmers start to build the mental representations or models of the source code text being examined. The resulting mental representations built, are likely to differ for novices and experts. However, even within the novice category, there are those that Corritore and Wiedenbeck (1991) refer to as 'high comprehenders'. These are novice programmers who display the level of thinking and use strategies which are typical of experienced programmers. The mental models developed by these comprehenders are also similar to those associated with expert programmers. This implies that it is possible for a novice programmer to

traverse much faster on the continuum from non-expert programmer to expert programmer than the 10-year timeframe suggested by Soloway and Spohrer (2013).

Furthermore, in the process of SCC, experienced programmers read the source code in question and locate a place(s) where comprehension needs to happen (Maalej et al., 2014). In order to discover these 'places', programmers need to have experience of some sort. For example, they ought to have some knowledge of the lower and higher syntactic structures, as well as at least the lower semantic structures (but ideally knowledge of both structures) of the programming language (Adelson, 1981; 1983). For this reason, experienced programmers are considered to pay attention to meaningful areas of the source code and to complex statements (*functional characteristics*), while novice programmers tend to visually concentrate on the comments and comparisons (*superficial features*) (Crosby & Stelovsky, 1990; Von Mayrhauser & Vans, 1995b). More specific details regarding the actual SCC strategies followed by novice programmers are presented as part of Article 1 (see Chapter 3). The intricate details of the strategies and detailed steps required for efficient SCC (as executed by experts) are covered as part of Article 3 (see Chapter 5).

While level of experience can have a big influence on a programmer's SCC competency (Singer et al., 1997), it is also necessary to consider other challenges that could potentially influence the development of a novice programmer's SCC skills. Moreover, knowledge of such additional challenges could be of value in the process of uncovering students' learning bottlenecks as part of Step 1 of the DtDs framework.

2.3 Challenges impacting the development of SCC skills

Due to the massification of higher education (Council on Higher Education, 2016; Phillips, 2019; Jansen, 2003), CS departments have to deal with large groups of students coming from diverse backgrounds. Since most of these students have limited or no programming experience (Kirkpatrick & Mayfield, 2017), many of them find it particularly difficult to master the key disciplinary skills of SCC (Cunningham et al., 2017; Shaft & Vessey, 1995). Over the past three decades, numerous studies have attempted to uncover the specific challenges experienced by novice programmers in comprehending source code (Bosse & Gerosa, 2017; Cunningham et al., 2017; Du

Boulay, 1986; Lister et al., 2004). Some of the more discipline-specific challenges (or difficulties) identified by these and other authors are covered as part of Article 1. There are, however, also other challenges that could influence the development of a novice programmer's SCC skills. The discussion in this section examines three such challenges: Lack of prior knowledge, lack of problem-solving skills, and lack of strong mental models. In the course of this examination, strategies are also considered that can be used by instructors to help students overcome these challenges.

2.3.1 Lack of prior knowledge

The term *prior knowledge* refers to what a student "already knows about a topic before learning more about it" (Veerasamy, D'Souza, Lindén & Laakso, 2018, p. 228). This knowledge is seen as "a much more important determinant of comprehension than was earlier thought" (Malarz, 1998). When students are unable to engage their prior knowledge to connect it to new understandings, it will hamper the creation of new knowledge (Bransford, Brown & Cocking, 2000). Programming students therefore need relevant prior knowledge in order to understand the concepts of the discipline and to perform well in programming courses (Alturki, 2016). For example, if students struggle to handle a mouse, type with one finger, and/or do not know how to save a document, they find it particularly difficult to learn basic computer literacy and programming skills at the same time (Oroma et al., 2012). Considerable research has been conducted that links prior knowledge (or the lack thereof) to programming performance (Allert, 2004; Patil & Goje, 2009; Pillay & Jugoo, 2005; Wilson, 2002). Veerasamy, D'Souza, Lindén and Laakso (2018) specifically investigated the role played by prior programming knowledge in lecture attendance and performance in the subsequent final programming examination during an introductory programming module. They found that students with prior programming knowledge performed significantly better than those without it. The students' lack of prior knowledge is something that is largely beyond the instructor's control (Kuhn, 2014). However, the impact of prior programming knowledge on student performance has been shown to gradually fade during the course period (Iv, Jagodzinski, Hao, Liu & Gupta, 2019) as students become more accustomed to the environment.

2.3.2 Lack of problem-solving skills

Programming is a process that is characterised by problem solving (Faux, 2001; Hazzan, Lapidot & Ragonis, 2011). As such, programming students at all levels of study should be formally equipped with skills to solve problems (Hazzan et al., 2011). Failure to do so will result in (1) failure of programming learning (O' Kelly et al., 2004); or (2) students using problem-solving strategies of their own that may be inadequate (i.e. lengthy or not helping students to arrive at the solution) for the problem-solving tasks in question (Oroma et al., 2012).

A process to follow when solving a problem starts with outlining the problem specifications and ends with the outline of a solution. This implies that programmers are challenged in the continuum from the specifications to the solution (Hazzan et al., 2011). However, programmers may also be required to move from the solution back to the original requirements (Biggerstaff et al., 1993). This further implies that programmers experience challenges in the continuum from specifications to solution and vice versa. When scrutinised, the following challenges associated with programmers are judged to fall within the specified continuum:

- Planning and designing computer programs (Astrachan & Rodger, 1998; Butler & Morgan, 2007; Mhashi & Alakeel, 2013);
- Developing algorithms from problem specifications (Sarpong, Arthur & Amoako, 2013);
- Thinking analytically and reasoning logically in solving computer programming problems (Butler & Morgan, 2007; Ismail, Ngah & Umar, 2010);
- Decomposing a problem description into sub-problems, implementing these sub-problems, and putting the pieces together into a complete solution (Lister et al., 2004, p. 119);
- Translating a manually solved problem into an equivalent computer program (Soloway et al., 1983);
- Making a transition from an understanding of separate program statements to the tasks that are to be achieved by groups of statements (Liffick & Aiken, 1996);

- Finding bugs in students' own written computer programs (Piteira & Costa, 2013, p. 76);
- Dividing program functionality into procedures (Piteira & Costa, 2013, p. 76);
- Combining syntax and semantics of individual program statements into a valid program (Wills, Deremer, Mccauley & Null, 1999);
- Understanding programming concepts to be applied in solving problems or to develop computer programs (Lister et al., 2004, p. 120; Sentance & Csizmadia, 2016, p. 480); and
- Mapping what is in the code or program back into the original software specifications or requirements (*concept assignment problem*) (Biggerstaff et al., 1993).

These challenges provide ample evidence of the intensity of the impact that a lack of problem-solving ability could have on programmers' capability to comprehend source code. This is in agreement with several authors (Chi, Bassok, Lewis, Reimann & Glaser, 1989; Reed, Miller & Braught, 2000; Sweller, 1988) on the importance of problem-solving abilities or skills in CS courses such as programming. In training students, strategies introduced to them should not be limited to a programming paradigm and/or language. Instead, the strategies should also be implementable in or applicable to other environments.

Regarding problem solving in introductory computer programming courses, McCracken et al. (2001, p. 126) proposed a framework that could be helpful to students. The framework comprises five successive steps: (1) abstract the problem from its description; (2) generate sub-problems; (3) transform sub-problems into sub-solutions; (4) re-compose the sub-solutions into a working program; and (5) evaluate and iterate. However, the framework is not sufficient because there are other skills that students may lack in the run-up to problem-solving (Lister et al., 2004). Hence, the ITiCSE 2004 working group (Lister et al., 2004) hypothesised that students might have problems reading even simple code, which has been confirmed by their multinational empirical investigation and other studies (e.g. Fitzgerald et al., 2005; Xie

et al., 2018). In addition to problem-solving abilities, this therefore implies that code-reading skills can also play a role in influencing the comprehension of source code.

2.3.3 Lack of strong mental models

Alturki (2016) loosely defines mental models as the internal "representation of concepts and ideas related to a given area, such as programming, in one's mind" (pp. 177-178). In the context of this study, mental models refer to a programmer's understanding of source code during the comprehension process (Letovsky, 1987). Throughout this process, a programmer reasons, explains, and makes some predictions about the behaviour of the source code in question (Cañas, Bajo & Gonzalvo, 1994; Norman, 1983). However, it is possible that the reasoning put forward by students and the explanations they provide, as well as the predictions they make, may be incorrect. In this way, they form non-viable mental models. Ma, Ferguson, Roper and Wood (2007) define non-viable mental models as those models that result in an invalid understanding of programming concepts. Instances of non-viable mental models are evidenced in some studies (Biggs & Collis, 1982; Lopez, Whalley, Robbins & Lister, 2008; The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society, 2013). Several studies (Corritore & Wiedenbeck, 1991; Nanja & Cook, 1987; Pennington, 1987b; Littmann, Pinto, Letovsky & Soloway, 1986; Soloway & Ehrlich, 1984; Wiedenbeck et al., 1999; Wiedenbeck, LaBelle & Kain, 2004) have indicated that mental models play a vital role in the ability of programmers to comprehend source code. More evidence of students' fragile knowledge identified through the mental models and/or representations they form while comprehending source code is discussed as part of Article 1 (see Chapter 4).

In their study, Corritore and Wiedenbeck (1991) conducted two experiments. In the first experiment, they examined the mental representations of the source code text formed by novices. In this examination, they used short fragments of the source code. Their results indicated that novice programmers form detailed and concrete mental representations of the source code text. In the second experiment, they used the results of the first experiment. They selected novice programmers classified as upper and lower comprehenders and tested them on a longer computer program. Their results indicated that novice programmers use detailed mental models of the source

code text and that they seldom make reference to the real world. On the other hand, high comprehenders (e.g. advanced novices in programming) use more abstract concepts in their mental models of the source code text and their abstractions have a lot of references to the real world.

Having considered three high-level challenges that could affect the development of SCC skills, more specific SCC challenges or bottlenecks are identified in Article 1 (see Chapter 4). Since SCC is regarded a skill that requires efficient application of a series of complex cognitive processes (Orlov, Bednarik & Orlova, 2016; Praveen, 2016), it is important for students to be aware of their 'thinking' as well as how they 'think about how they think' when doing things (e.g. performing SCC-related tasks). As such, the next section discusses both cognitive and metacognitive practices that can influence the comprehension of source code.

2.4 Cognitive practices

Cognition is defined as internal or mental processes that enable human beings to gain knowledge from their surroundings and to retain it (Cognifit, 2019; Preece, Rogers & Sharp, 2015; Schlinger, 1995). In any teaching-and-learning-related study, it is important not only to understand how knowledge is gained, but also how individuals are able to locate, identify, and accurately retrieve this information for future use.

2.4.1 Knowledge acquisition and retention

In an educational environment, students are generally expected to be able to remember what they have learned so that they can pass tests and examinations, as well as (ultimately) apply knowledge gained in lower- to advanced or upper-level classes (Barkley, 2010). This is in line with the learning edge momentum philosophy (Robins, 2010), namely that a student's ability to understand new programming concepts is linked to the preceding related concept(s) that the student has learned. Careful implementation of this philosophy therefore suggests that it may not be possible for students to progress to higher levels of study in programming while the concepts that ought to have been studied in the lower levels of study are still lacking (Alturki, 2016). Consequently, these students who did not experience the learning

edge momentum that upper-level students had, are not likely to overcome the threshold concept (Meyer & Land, 2003) problem.

Both short- and long-term memory play a vital role in remembering information received and applying acquired knowledge. According to Barkley (2010), "short-term memory occurs when the brain works with new information until it decides if and where to store it more permanently" (p. 22). If such information is to be retained over long intervals, it is stored in the long-term memory (Waugh & Norman, 1965). Newly acquired information or knowledge can typically be stored for about 24 hours in the short-term memory and can be recalled during that time. After 24 hours, such information or knowledge is consolidated into long-term memory storage, and is available for recall over a longer period of time (e.g. forever or for several decades) (Barkley, 2010; Robin, 2002). The problem, however, is that the neural networks – those brain structures that facilitate the storage of knowledge in long-term memory for future retention – gradually wear off when that knowledge is not used in relevant future work or applied in life situations (Ratey, 2001). This implies that people's (including students') brains must be continuously tapped into so that the existing connections between new and already known information can be strengthened (Moon, 2004; Sousa, 2006; Wlodkowski & Ginsberg, 2010).

According to Barkley (2010), knowledge or information does not just move from the short- to long-term memory (a process known as consolidation). There are three specific attributes that must be achieved during instruction to help ensure that this movement occurs:

- *Emotional connection* – There is a likelihood that information can be stored in the long-term memory if students are able to connect emotionally to such information. To achieve this, instructors can instigate the relevant human dimension in the learning process by using teaching aids that cause the learning content to have some impact on students' lives.
- *Sense* – The information or knowledge should make sense to students and be relevant to what they already know. To achieve this, instructors can organise learning into units characterised by themes and integrate the relevant analogies or metaphors into the teaching and learning process.

- *Meaning* – Students must find a reason to justify why the knowledge they gain during the teaching and learning process must be remembered. To achieve this, instructors can ask students to "connect what they are learning to their past, to what is going on presently in the world around them, or to the professional or civic responsibilities they may have in the future" (p. 101).

With respect to uncovering expert ways of thinking as suggested by Step 2 of the seven-step DtDs framework, it is vital to also delve deeper into the programmers' 'thinking about how to think' (metacognition) while comprehending source code.

2.4.2 Metacognition

Metacognition is defined as "one's knowledge concerning one's own cognitive processes and products or anything related to them" (Flavell, 1976, p. 232). Hennessey (1999) later defined it as:

> *"Awareness of one's own thinking, awareness of the content of one's conceptions, an active monitoring of one's cognitive processes, an attempt to regulate one's cognitive processes in relationship to further learning, and an application of a set of heuristics as an effective device for helping people organize their methods of attack on problems in general"* (p. 3).

Therefore, using some elements of these definitions, metacognition in the context of this study can be defined as the ability of a person to be cognisant of processes in their mind (mental or cognitive processes) and to be in a position to exert the necessary control (regulation) over such processes for the best possible learning. [More details on the cognitive process aspects are covered in Article 2 (see Chapter 5)].

Metacognition is a critical element in the learning environment (Ambrose et al., 2010; Frith, 2012) that is often taken for granted by stakeholders. Students with good metacognitive skills are typically classified as high academic achievers (Greeno, Collins & Resnick, 1996; Lovett, 2008). In the CS discipline, metacognitive skills have been found to play an essential role in solving computer programming problems (Parham, Gugerty & Stevenson, 2010). Well-performing CS students have also been found to use more metacognitive strategies than their lower-performing peers (Bergin,

Reilly & Traynor, 2005; Shaft, 1995). More details on the metacognitive strategies applied by expert programmers during SCC are covered as part of Article 2 (see Chapter 5).

2.4.2.1 Metacognitive promotion strategies

Although fostering metacognitive practices among students is not an easy task, there are reports of positive results in this regard. Most of the desired outcomes are characterised by strategies that include the planning, monitoring, and regulating of mental processes when doing something or performing a task (Akturk & Sahin, 2011; Lai, 2011). Deducing from the activities of Step 3 in the seven-step DtDs framework as suggested by Middendorf and Pace (2004), modelling can also be used as a strategy to promote metacognition.

Planning

The planning phase is characterised by setting goals, scanning through a task, asking yourself several questions before starting with and while working on the task, and analysing the tasks or steps that are involved or should be involved in tackling the problem at hand. These planning activities trigger the prior knowledge of programmers (Bergin et al., 2005; Pintrich, 1999). In the triggering process, programmers are able to compare and contrast the new information they are seeing/receiving or getting with already known information. In the process they learn, and are hence able to confidently decide on the appropriate approaches to use for the task in question (Barkley, 2010).

Monitoring

The monitoring phase is characterised by programmers paying attention to their mental processes as they read or work through a given task. In the process, they also use some techniques to test their comprehension (Bergin et al., 2005; Simons & Bolhuis, 2004). The techniques can include thinking aloud (Whalley & Kasto, 2014); underlining some lines of code or keywords (Powell, Moore, Gray, Finlay & Reaney, 2004); drawing some diagrams (Lister et al., 2004); making analogies or metaphors (Pace, 2017a); asking yourself questions about the various aspects of the task at hand (Eisenführ, Weber & Langer, 2010; Herrmann, 2017; Uzonwanne, 2016); and making notes or summaries to ensure comprehension of the content (Van Gorp & Grissom,

2001). According to Pintrich (1999), employing these techniques exposes programmers to situations where they might have breakdowns in attention or comprehension. He believes that the use of regulation strategies or activities (regulation phase) can repair the associated breakdowns.

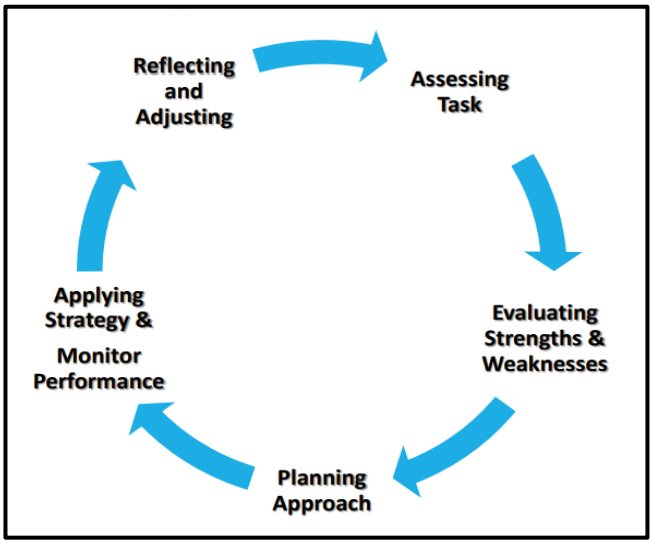
Regulation

The regulation phase is characterised by programmers continuously modifying activities based on what they found during the planning and monitoring phases, as well as what they experience as they continue working on the activity (Bergin et al., 2005; Simons & Bolhuis, 2004). As an example, programmers may reread the given text or scenario in order to confirm whether they really understand it or not. They may also slowly read or reread the scenario if they think they do not understand it or if there are some aspects that are difficult to understand. Additionally, programmers may skip some aspects that they find challenging to understand with the hope that information contained in the subsequent text may provide more insight (Moore, Zabrucky & Commander, 1997; Pintrich, 1999).

Modelling

With the modelling strategy, the instructors explicitly show the students the steps that they (the instructors) themselves would follow to tackle a given assignment or problem-solving task. Following these steps, instructors walk through various stages of their metacognitive processes, and normally demonstrate their thinking physically on the board and in front of the students as they go about solving a problem. A thinking-aloud technique is deliberately used in this type of demonstration. This technique requires the person using it to verbalise the steps or procedures followed to perform a certain task. To further strengthen the modelling strategy, the verbalised thought processes can be performed concurrently with asking the relevant questions in the course of the demonstration (Ellis, Denton & Bond, 2014; Kistner et al., 2010). Nilson (2013) suggests that, other than modelling by the instructor or his/her assistants, the task can also be assigned to students. Overall, modelling is normally achieved by following the fundamental metacognitive processes suggested by Ambrose et al. (2010) (also see Figure 2.2). In these processes, students:

- “Assess the [learning] task at hand, taking into consideration the task’s goals and constraints;
- Evaluate their own knowledge and skills, identifying strengths and weaknesses;
- Plan their approach in a way that accounts for the current situation;
- Apply various strategies to enact their plan, monitoring their progress along the way; and
- Reflect on the degree to which their current approach is working so that they can adjust and restart the cycle as needed” (pp. 192-193).



[Source – Adapted from Ambrose et al. 2010, p. 193)]

Figure 2.2 – Metacognitive Process Cycle

According to Ambrose et al. (2010), the modelling process does not stop immediately after going through the metacognitive cycle. Instead, they recommend that instructors give students a problem to solve or an activity to undertake. To help students develop metacognitive skills in the given assignment, instructors are expected to provide them with the leading questions that they should ask themselves throughout the steps. Examples of such questions could be: What problem am I trying to solve? Where do I start? How do I start? What are the next steps after starting? What do I need to solve this problem? What strategies do I need to solve the problem? How will I decide on the best strategy? Is there an alternative approach?

The aforementioned questions form part of a scaffolding strategy which has been identified in several studies (e.g. Bickhard, 2013; Feyzi-Behnagh et al., 2014; Fitzgerald et al., 2005) to play a critical role in facilitating student learning. Ambrose et al. (2010) define scaffolding as the "process by which instructors provide students with cognitive supports early in their learning, and then gradually remove them as students develop greater mastery and sophistication" (p. 146). Lev Vygotsky is one of the early thinkers of metacognition who strongly influenced the concept of scaffolding (Robins, 2010). In the scaffolding strategy, instructors use various teaching tools, guides or techniques (also known as scaffolds) that help students develop comprehension beyond their ability or to successfully perform unfamiliar tasks (Deejring, 2015; Raymond, 2017; Vygotsky & Cole, 1978; Wood, Bruner & Ross, 1976). This comprehension happens by way of instructors asking students to work on distinct phases of the given task in isolation, before being asked to assimilate the phases. Instructors can also provide a framework for tasks that require considerable or full student autonomy, and later relegate most of the tasks back to the students (Ambrose et al., 2010).

Considerable research has indicated that the use of scaffolding techniques helps students to improve their performance (Azevedo, Cromley, Winters, Moos & Greene, 2005; Molenaar, Van Boxtel & Sleegers, 2011; Roehler & Cantlon, 1997). Consequently, it is vital to integrate scaffolding techniques [more details can be seen in Article 2 (see Chapter 5)] into all teaching and learning efforts, including the teaching of SCC skills. The objective is to see how the identified cognitive and metacognitive practices can help students to better comprehend source code if they "reflect on their ideas" (Davis, 2000, p. 819). The biggest problem in executing the modelling and scaffolding techniques proposed here, is that instructors are not necessarily always aware of the exact mental processes they follow in comprehending source code – mostly due to their 'expert blind spots' (Grossman, 1990; Nathan & Petrosino, 2003; Shulman, 1986). One of the main purposes of Step 2 in the DtDs framework is to uncover the explicit nature of these blind spots (Middendorf & Pace, 2004). In this regard, both Article 2 and Article 3 consider cognitive aspects useful in overcoming problems related to blind spots.

2.5 Summary

This chapter presented the possible strategies that can be followed in comprehending source code. From the discussion of these strategies, it surfaced that there are no 'hard and fast' rules to adhere strictly to these strategies. Instead, some elements of each strategy or a combination of these strategies can be used. Realising this, it became necessary therefore to consider how novices and experts differ in applying these strategies. This was achieved by comparing the strategies that they (novices and experts) use in comprehending source code. As a precursor to Step 1 of the seven-step DtDs framework, the challenges related to the development of SCC skills were also considered. Moreover, both cognitive and metacognitive aspects that may influence the process of comprehending source code were considered.

Informed by the content of this chapter, the next chapter presents a detailed description of the research design and methods for the study, as well as detailed procedures followed throughout the investigation.

Chapter 3 – Research Design and Methodology

3.1 Introduction

In this chapter, details regarding the research design and research methodology of the study are outlined. The chapter commences with a discussion of the theoretical underpinnings of the selected design and methodology. Thereafter, a detailed discussion is provided on how the empirical part of this study unfolded in the process of finding answers to the stated research questions (as set out in the introductory chapter). For each of the three phases of this study, aspects related to the aim; chosen data source management strategy; population and sampling procedures; data collection strategies and data analysis methods are discussed. Next, measures taken to ensure the trustworthiness of the study findings are outlined. The chapter concludes with a description of various ethical considerations adhered to in this study.

3.2 Research design

The research design of this study was based on the seven-step DtDs framework (see Figure 1.1). In applying this framework, Step 1 guides instructors to identify students' discipline-specific bottlenecks in the learning process. A bottleneck is defined as points or places in a given course where the learning of many students is interrupted such that they are not able to advance their thinking and discovery (Middendorf & Pace, 2004). After identifying the bottlenecks, the crucial mental operations required to overcome such bottlenecks are uncovered through the assistance of disciplinary experts (Step 2). These operations are then modelled explicitly to students (Step 3). After this, instructors create opportunities for students to practise these operations or skills and get feedback on their mastery of the skills (Step 4). In the process, motivational strategies or principles are applied to assist students in effectively learning the imparted skills (Step 5). Eventually, an assessment is made of how well the undertaken efforts help students to master the intended learning content (Step 6). As part of the final step (Step 7), instructors are encouraged to share (formally or informally) their experiences from this process (Middendorf & Pace, 2004; Pace, 2017a).

The basic idea underpinning the DtDs theory is that ways of thinking and doing are specific (and often unique) to each academic discipline (Middendorf & Pace, 2004). Boman et al. (2017, p. 13) claim that instructors, “operating as experts in their disciplines, hold tacit knowledge and implicit ways of thinking that are not accessible to novices in the discipline”. In this case, the concept of ‘expert blind spots’ surfaces. Expert blind spots occur when instructors (as experts) unintentionally omit skills, steps or information that are essential for novice students to effectively learn and perform learning activities (Ambrose et al., 2010, p. 112). This implies that instructors in their various disciplines may not always be aware of the steps they are trying to reflect in their thinking, or they may sometimes take some steps for granted when explaining a new concept. Computer Science instructors, particularly those teaching programming courses, may not be an exception to this practice, especially considering the distinct and inherent challenging thought processes involved in comprehending source code.

Over the past three decades, numerous research studies have highlighted the importance of shaping teaching to match the specific (and unique) conditions of each academic discipline (Middendorf & Pace, 2004). One important aspect of the differences between disciplines was elucidated by Tobias (1992-1993). She observed that even intelligent and hard-working instructors and students alike experience problems when they move from courses in their own speciality to other courses in other disciplines. It follows, therefore, that students have made their own preconceived assumptions about their academic disciplines. According to Sengupta, Bhattacharya and Sengupta (2012), these assumptions hinder students’ learning. Therefore, these preconceived assumptions may need to be dealt with for students to understand content (academic and/or non-academic) beyond the boundaries of their own discipline.

DtDs was developed by David Pace and Joan Middendorf at Indiana University in the United States of America. Their pioneering work, conducted in the discipline of History Education, was published in a special issue of *New Directions for Teaching and Learning* in 2004 (Middendorf & Pace, 2004). DtDs started as a program to help instructors teaching large classes. Over time, it developed into a model to help students get through places where they get stuck (also known as bottlenecks) in thinking distinct to a particular disciplinary environment. DtDs has recently grown into

a theory of pedagogy – the core here is for experts to break down complicated thinking in a specific discipline into steps that novices can understand and easily follow (King et al., 2013).

By 2013, at least 17 published studies in various disciplines such as history, marketing, humanities, and biology had already employed the DtDs theory (Tingerthal, 2013, pp. 50-51). This trend has continued over the past few years, with more studies employing the DtDs theory "coming out all the time" (Middendorf, personal communication, 1 September 2017). According to Indiana University (2019), in 2017 alone, at least 32 research works using DtDs were published in various disciplines (e.g. Belanger, 2017; Pace, 2017b; Shopkow, 2017; Timmermans & Meyer, 2019; Tucker, 2017). Originally, the dominating studies were in the History discipline (Shopkow, Diaz, Middendorf & Pace, 2012). However, the trend seems to be taking another direction as DtDs is gaining popularity worldwide and is used in other disciplines. Further, Miller-Young and Boman (2017, p. 9) point out that there are several teams in at least ten countries exploring DtDs.

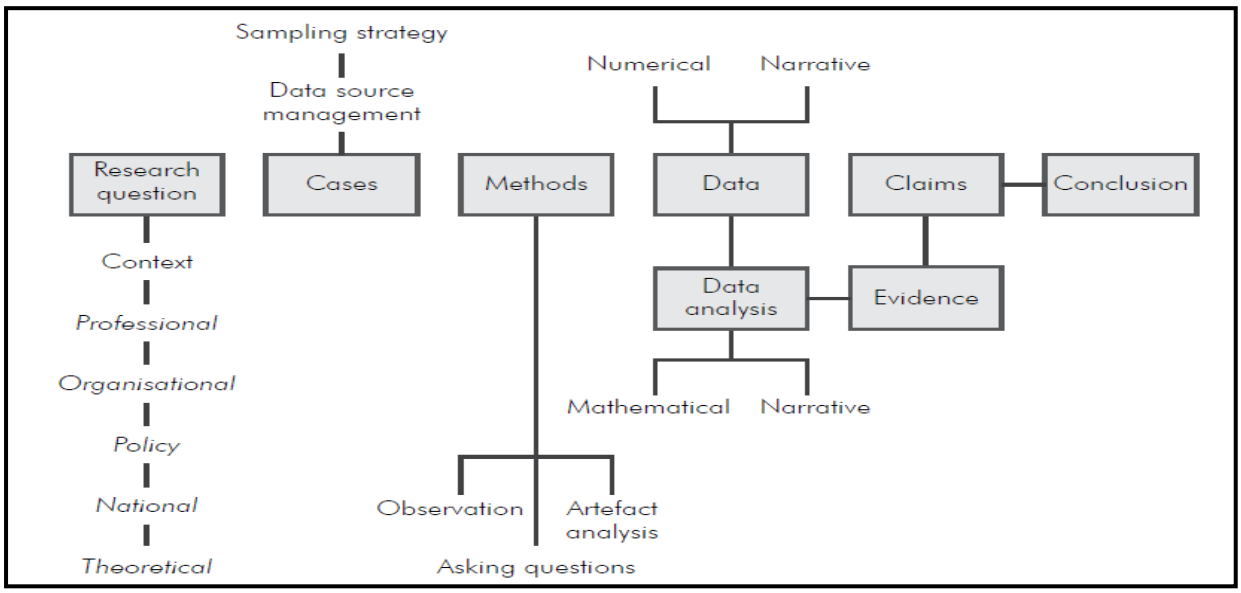
Although not much work is available in the public domain employing DtDs in the CS discipline, Menzel and colleagues have used the DtDs model to help students in Computer Science overcome bottlenecks associated with understanding the complex concepts of recursion (Menzel, 2015); debugging; and logical reasoning in algorithmic complexity and proving computer program correctness (*Discrete Mathematics course*) (German, Menzel, Middendorf & Duncan, 2014). Furthermore, Dan Richert used DtDs to overcome three problems encountered by students in a Database Design and Data Retrieval course. These were (1) understanding ER diagrams; (2) reasoning in MySQL; and (3) dualism (there is no one right answer on MySQL statement) (IUBCITL, 2016). Thurner, Zehetmeier, Hammer and Böttcher (2017) also applied the DtDs model, where they attempted to analyse the initial competences of CS freshmen students and relate them to study requirements. It can, therefore, be deduced that although DtDs originated in the United States of America, it is spreading as a theory and methodology presently employed in enhancing disciplinary instruction in many different disciplines and in many parts of the world. As such, it can be regarded as highly applicable to a study in CS with its particular ways of thinking and doing; and to a study in which the researcher specifically attempts to understand cognitive

processes and challenges of novice students as they go through the process of source code comprehension.

In this study, the DtDs framework was adapted to create an environment conducive to conducting the empirical investigation and ultimately answering the research questions of the study. DtDs adaptations are supported by Middendorf and Pace (2004, p. 4), who indicate that the DtDs steps are neither 'mechanical' nor 'deterministic'. The framework is deemed a suitable research design because it is pedagogical in nature (King et al., 2013). Furthermore, the framework helps to answer a series of questions that instructors can ask themselves as they try to understand how students think and learn in their specific disciplines (Middendorf & Pace, 2004, p. 1). Details on how the adaptations of the DtDs framework were done in this study are provided in the subsequent sections.

3.3 Research methodology

Within the realm of the DtDs-based research design, this study followed a research approach based on Plowright's (2011) Frameworks for an Integrated Methodology (FraIM) (see Figure 3.1). The following sub-sections provide details regarding the characteristics of this framework, issues surrounding data collection in studies using the framework, and justification for using the framework in this study.



(Source : Plowright, 2011, p. 9)

Figure 3.1 – The FraIM

3.3.1 Characteristics of FraIM

The FraIM indicates how the researcher can traverse from formulating a research question throughout all research stages to making conclusions about the study. This integrated methodology presents a very fitting way around the controversial issues of qualitative and quantitative research in other mixed methods' designs (Hellawell, 2016). As such, FraIM is characterised by the following (Plowright, 2011; 2016b):

- No philosophical position needs to be taken before commencing with the study. Such position can, however, be taken as the study evolves or even with the interpretation of results.
- The philosophy used to comprehend and conceptualise the research process should not determine the methodology applied in research, but the reverse is true.
- The *Q words* (quantitative and qualitative) should not be used in any part of the study (i.e. conceptualisation, design, methodology, axiology, and reporting). Instead, the *N words* (numeric and narrative) are recommended for use. Consequently, no distinction is made between quantitative and qualitative methods in any FraIM-based research study.
- A combination of different types of data source management strategies (*experiments, surveys and case studies*) can be used in one research study.
- FraIM-based research studies use *cases* or data sources which are managed through these three data source management strategies.
- The cases can be organised based on three factors:
    - The number of cases in the research;
    - The degree of control the researcher has over cases allocated to various groups; and
    - Ecological validity – the degree of how natural the location and placement of the research is.
- Exact and precise counting is not used to decide whether the data source management strategy is a survey, experiment, or a case study. Instead, the professional judgement of the researcher is used (i.e. the amount of in-depth

information that can be gathered from the selected cases and the degree of generalisability that can be made about the inferences from the data collection are used as the basis for the judgement).

- After researchers decide on the approach to data source management, they make sampling decisions which determine the cases or participants (i.e. number, richness, and nature or type of cases) to be included in a research study.
- The researcher is encouraged to be more responsive, flexible, and have an open-minded attitude in the process of answering the study's research question(s) or finding a solution to a problem.
- In FraIM-based research studies, it is easy to integrate both structured and relatively less structured approaches to data collection methods.
- All data collection methods are treated as being equally acceptable and their use is essentially determined by the research question(s) and what the research is set out to achieve.
- No preference is given to either numerical or narrative data over another.
- The previous phases of the investigation can be revisited as and when necessary.
- Different research approaches can be used and integrated at all stages of the research process.
- The term 'unstructured' is not used in research conducted within FraIM, especially when it comes to data collection (i.e. interviews). Instead, the structure is viewed on the continuum from low degree to high degree of structure.
- Decisions must be taken at each stage of the research process. These decisions help the researcher to understand the phenomena under investigation.

3.3.2 Data collection in FraIM

The FraIM distinguishes between three broad types of data generation and collection methods: observations, asking questions, and artefact analyses. In any research study

employing this methodology, any one data collection method or a combination of these methods can be used (Plowright, 2016a, p. 251). It is also important to note that any type of data (*numerical or narrative*) can be collected using any type of data collection method (*observations, asking questions or artefact analyses*) and any data source management strategy (*surveys, experiments or case studies*). According to FraIM, each type of data collection differs in terms of (1) structure (*highly structured or little structure determined by using open or closed questions*); and (2) level of mediation (*how close in time and space the researcher is to the phenomenon under study*).

For example, in the case of 'asking questions', structured questionnaires and/or open interviews can be used. Resultant data can thus be numerical or narrative in nature (*two main categories of data in research employing FraIM*). The difference in the level of mediation in this example can be described as follows: researchers are distant to the unit of analysis in structured questionnaires, while they are in the vicinity of the phenomenon under study in open interviews. Guided by the preliminary findings as the research study unfolds, it is argued that the study employing FraIM should allow for numerical data to be converted into narrative data and vice versa (Plowright, 2011).

3.3.3 Justification for using FraIM

Based on the given characteristics of FraIM and issues surrounding data collection, it can be deduced that 'mixed methods using FraIM' is what Plowright (2016b, p. 21) refers to as 'an integrated methodology'. This methodology goes beyond the *mainstream mixed methods research paradigm* as discussed by several authors (Bryman, 2006; Creswell, 2014; Creswell & Plano Clark, 2011; Hesse-Biber, 2010; Teddlie & Tashakkori, 2009). According to Plowright (2016a, p. 243), the mainstream mixed-methods paradigm is 'traditional', because researchers who use it still adhere to the thorny issues of quantitative and qualitative research. As an example, Morse (2003, p. 190) describes mixed methods as a paradigm involving the use of several quantitative and qualitative techniques in a research study driven by quantitative and qualitative theories.

Plowright's (2016b) integrated methodology can, therefore, be regarded as an emerging paradigm, and as such, as a novel way of research thinking in the 21st

century and beyond. As a result, FraIM was regarded as highly relevant for this study due to the following reasons:

- In an endeavour to overcome the SCC challenges of novice students, there was a need to move back and forth using various strategies and data collection methods that were not yet positively known at the beginning of the study.
- Multiple data collections and analyses were undertaken throughout the empirical investigation process. This helped the researcher to obtain thick and rich information that helped him to effectively address the stated research questions of the study.
- The number of cases used in this study was not yet positively known at the beginning.
- Part of the data collection for this study was drawn from test/examination or assignment scripts (*artefact analysis*).
- Some decisions were informed by the events and outcomes of the investigation [i.e. final conclusion of the 'useful' bottleneck(s) that formed the focus of this study].
- Data collection structure and the researcher's level of mediation (i.e. low, medium or high) to the unit of analysis were not yet positively known at the beginning.
- The degree of control (i.e. high, medium or low) that the researcher had over cases involved in this study was not yet positively known at the beginning.
- The degree of structure associated with data collection methods in the study was not yet positively known at the beginning, hence the type of coding (pre/closed or post/open) used during data analysis was not predictable.

Having detailed the design and methodology chosen for this study, issues related to purpose and procedure, coupled with the selection of data source management strategies; population and sampling; data collection methods; and data analysis techniques, as well as the nature of the resultant data in the three phases of the study, are described next. According to Plowright (2011), the data source management and sampling decisions are two levels at which research cases occur. Data source

management happens when the researcher decides on the approaches to use in managing the sources of data. On the basis of FraIM, approaches to data source management can be organised based on three factors, namely the number of cases in the research; the degree of control the researcher has over cases allocated to various groups; and ecological validity (the degree of how natural the location and placement of the research is).

3.4 Details of empirical study

In the following sub-sections, the three research phases that made up the empirical part of this study are discussed.

3.4.1 Phase 1

3.4.1.1 Aim

As a start to Step 1 of the DtDs framework, the aim of Phase 1 was to identify specific senior CS students who were having difficulty comprehending short pieces of source code.

3.4.1.2 Data source management

The data source management approach in Phase 1 was a survey. This approach was selected because the researcher wanted to collect a wide breadth of information from a relatively large number of participants (or cases). These participants were also drawn from naturally existing groups, with little disruption to their ongoing activities. Furthermore, we (the researcher and the lecturer of the selected module) had authority (i.e. control) over those students (as study participants). The survey was more structured in the sense that students had to answer specific questions in a given venue and within a specific time slot (Plowright, 2011; 2016b).

3.4.1.3 Population and sampling

The population for the Phase 1 research activity was senior CS students from a selected South African higher education institution. The sample for Phase 1 consisted of the 40 students registered for the 3rd year Internet Programming module. The sample was purposeful (Cooper & Schindler, 2013) 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 (Patton, 2015), since the researcher had easy access to the participants as the lecturer responsible for the module agreed to open up one of her scheduled class sessions for this research activity.

3.4.1.4 Data collection method

The data collection method for Phase 1 was 'asking questions' (through a questionnaire). This method was characterised by a medium level of mediation, because it was possible not to know – during the first encounter – some actual interpretations for some answers provided by the participants. This means that there had to be extended engagement with these answers for full understanding. Using this method, the degree of structure was high due to the following reasons: First, the researcher had a high level of control in the sense that the questionnaire had specific questions that participants had to answer. Second, the possible responses to the questions were predetermined. Third, the questions included in the questionnaire were pre-structured, as it was based on an existing set of questions. This data resulted from students' answers to 12 multiple-choice SCC questions (Plowright, 2011; 2016b). The data consisted of two sub-categories, namely artefacts and performance data. Artefacts indicated the real writings (e.g. sketches, drawings, text) that students made in answering the questions. According to Lister et al. (2004), these writings are known as doodles. Performance data indicated the overall scores of students on the test.

3.4.1.5 Procedure

Students were given a questionnaire containing the 12 multiple choice questions (MCQs) (see Appendix A) developed and used by *the ITiCSE 2004 working group* for a multi-national study (Lister et al., 2004). The students answered these questions under test/examination conditions. They worked through the 12 short fragments of source code and then either had to predict the outcome of executing such fragments or select a piece of source code (from a small set of options) that would correctly complete a given near-complete code snippet. There were two reasons for using this specific set of MCQs: First, all the questions contained source code fragments that students had to read, interpret, comprehend and ultimately answer related questions. Second, the questions had been tested with a large population of students in several universities in both the United States of America and other countries. The questions,

as used by the *ITiCSE 2004 working group,* were originally written in Java, but were all converted to the C# programming language. This was the one language that all the study participants were familiar with.

3.4.1.6 Data analysis

Analysis of the numeric data collected through the questionnaire was fairly straightforward. The researcher first graded all the submitted scripts, after which he captured the marks on an MS Excel spreadsheet. Since one mark was awarded for each of the 12 questions, a '1' or '0' was captured for each question. This was used to identify students who answered specific questions wrong. For each of the questions, the mean and standard deviations were calculated. These helped the researcher to identify questions that were the most challenging to students. The aggregate performance of students who answered the questionnaire, as well as the ranking of the questions compared to the ITiCSE 2004 group (Lister et al., 2004) can be seen in Appendix B. Consequently, the three most difficult questions (Q3, Q6 and Q8) were selected for use in Phase 2. The results of the research activities in Phase 1 were also used to identify appropriate participants for Phase 2. A description of the Phase 1 activities is therefore included as part of Article 1 (see Chapter 4).

3.4.2 Phase 2

3.4.2.1 Aim

As a continuation of Step 1 of the DtDs framework, the aim of Phase 2 was to uncover specific points or places (Middendorf & Pace, 2004) where senior students were experiencing SCC difficulties, with the goal of identifying common and useful SCC bottlenecks. Phase 2 was therefore set up to answer the following two research questions:

> ***RQ4 (a)****: What are the major SCC difficulties experienced by senior CS students?*
>
> ***RQ4 (b)****: How can knowledge of these difficulties be used to identify SCC bottlenecks that should ideally be addressed in introductory programming courses?*

3.4.2.2 Data source management

The data source management approach in Phase 2 was a case study. This approach was selected because of three reasons: First, the researcher wanted to collect in-depth information from a smaller number of participants. Second, participants would be disrupted minimally in order to spend some time with the researcher when they were free from their other academic responsibilities. Third, the researcher wanted to study only a few participants in a conducive environment, but where he had some degree of control in terms of probing the participants where he deemed it necessary in order to obtain rich and thick descriptions (Plowright, 2011; 2016b).

3.4.2.3 Population and sampling

The population for the Phase 2 research activity was third-year students registered for the 3rd year Internet Programming module at the selected institution. The sample consisted of the 15 students selected on the basis of a specific criterion. These were the students from Phase 1 who incorrectly answered all the questions selected for use in Phase 2. The sample could therefore be regarded as purposeful (Cooper & Schindler, 2013). The sample was also convenient (Patton, 2015), because the students were studying at the selected institution and the researcher could have sessions with them any time they were free.

3.4.2.4 Data collection methods

The data collection methods for Phase 2 were ‘asking questions’ (through think-aloud interviewing) and ‘making observations’. The asking-questions method was characterised by a higher level of mediation because the researcher was close to all the events, sources of data, as well as the actual data involved in the research activity for this phase. The researcher had a high level of control for the same reasons mentioned in Phase 1 above. The making-observations method was characterised by a relatively high level of mediation, because the researcher was also asking probing questions where necessary while making observations. The degree of structure with the observations was high due to the reasons already alluded to (Plowright, 2011; 2016b).

The questions used for the research activity in Phase 2 were identified based on the results of the Phase 1 research activity (see Section 3.4.1.6). Three forms of narrative

data were collected as part of the Phase 2 research activity. First, narrative data resulted from the transcription of the audio recordings of the individual sessions the researcher had with the participants. Second, doodle data resulted from the real writings (e.g. sketches, drawings, text) that students created in answering the questions. Third, observations made throughout the session were written down as notes (narrative) to later supplement the discussion of the study findings. In order to fine-tune the research activities of this phase, a pilot was conducted.

3.4.2.5 Piloting of Phase 2

As pilot studies play a crucial role in the research process, such a study was conducted for Phase 2. Since pilot studies are nearly always conducted with a smaller number of participants (Van Teijlingen & Hundley, 2002; Van Teijlingen, Rennie, Hundley & Graham, 2001; Vogel & Draper-Rodi, 2017) , only one participant was involved in the Phase 2 pilot. This participant was a postgraduate CS student. The pilot study was conducted to fine-tune the structure of the individual sessions and to finalise a generic list of possible probing questions to be used in the event that participants got stuck or remained silent for too long.

The pilot participant took 16 minutes and 30 seconds to answer all three questions (selected and used in this Phase) in a think-aloud manner. Given the advanced experience level of the pilot participant, the researcher decided to schedule at least 45 minutes for each of the individual research participant sessions. In response to a question posed by the researcher (as the interviewer), the pilot participant indicated that she was familiar with the think-aloud technique, hence it was not demonstrated to her. Although the pilot participant did not have any particular difficulties in executing the technique, the researcher decided that it would be better to explicitly demonstrate the technique to all participants in the real Phase 2 study so that they would know exactly what was expected of them.

3.4.2.6 Procedure

To identify the Phase 2 participants, the researcher chose all students who incorrectly answered all three questions (Q3, Q6 and Q8 – see Figure 3.2, Figure 3.3, and Figure 3.4) selected for use in this phase. Fifteen students were found to belong to this category. These participants were invited through a formal invitation letter (see

Appendix C) to take part in the research activity in this phase. However, only 10 of the invited participants selected to participate in the interview sessions. These participants were given a questionnaire containing the three selected MCQs. For each of these questions, participants had to work through the short fragments of source code and then predict the outcome of executing such fragments.

Consider the following source code fragment:

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

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

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

int count = 0;

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

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

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

(Source: Lister et al., 2004, p. 141)

Figure 3.2 – Question 3

Participants were also asked to use the think-aloud technique while they worked out the question answers. Think-aloud is a technique whereby participants are instructed to speak out loud any thoughts that come to their minds while performing the task at hand. This technique allows the researcher some insight into how the participant reached the solution. Using the technique, the researcher is also able to better understand the participant's mental steps (i.e. processing of working memory) (Charters, 2003; Van Someren, Barnard & Sandberg, 1994). The protocol that guided the proceedings for the think-aloud sessions is included in Appendix D. The think-aloud technique was demonstrated to all participants before they started working on

the tasks in question. For this demonstration, an SCC question (see Figure 3.5) sourced from the study of Sheard et al. (2015, p. 146) was used.

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

```
public static bool isSorted (int[] x)
{
//missing source code goes here
}
```

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

a)
```
bool b = true;
for (int i = 0; i < x.Length - 1; i++)
{
 if (x[i] > x[i + 1])
    b = false;
 else
    b = true;
}
return b;
```

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

c)
```
bool b = false;
for (int i = 0; i < x.Length - 1; i++)
{
     if (x[i] > x[i + 1])
     b = false;
}
return b;
```

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

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

(Source: Lister et al., 2004, p. 142)

Figure 3.3 – Question 6

If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array **x** that contains the following six numbers:

```
  4 5 6 2 1 3
```

There are 10 inversions in that array, as:

```
x[0]=4 > x[3]=2
x[0]=4 > x[4]=1
x[0]=4 > x[5]=3
x[1]=5 > x[3]=2
x[1]=5 > x[4]=1
x[1]=5 > x[5]=3
x[2]=6 > x[3]=2
x[2]=6 > x[4]=1
x[2]=6 > x[5]=3
x[3]=2 > x[4]=1
```

The skeleton source code below is intended to count the number of inversions in an array **x**:

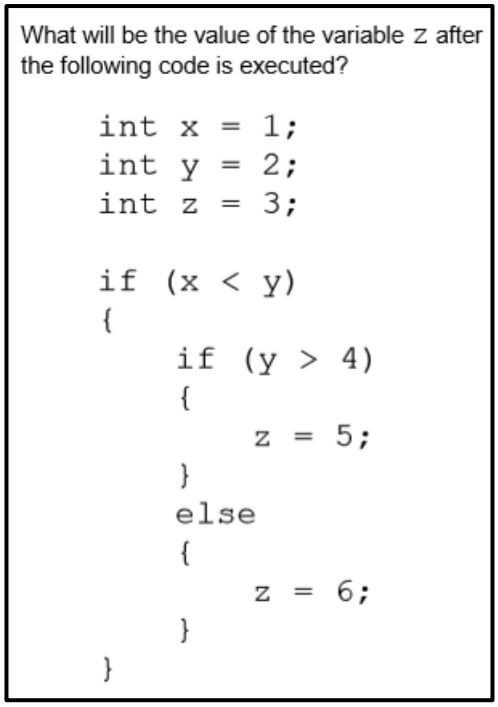
```
int inversionCount = 0;

for (int i = 0; i < x.Length - 1; i++)
{
      for xxxxxx
      {
            if (x[i] > x[j])
            ++inversionCount;
      }
}
```

When the above source code finishes, the variable **inversionCount** is intended to contain the number of inversions in array **x**. Therefore, the **xxxxxx** in the above source code should be replaced by:

a) `(int j = 0; j < x.Length; j++)`
b) `(int j = 0; j < x.Length - 1; j++)`
c) `(int j = i + 1; j < x.Length; j++)`
d) `(int j = i + 1; j < x.Length - 1; j++)`

(Source: Lister et al., 2004, p. 143)

Figure 3.4 – Question 8



(Source: Sheard et al., 2015, p. 146)

Figure 3.5 – A think-aloud technique demonstrating question

3.4.2.7 Data analysis

The narrative data collected during the individual think-aloud sessions was analysed thematically based on an adapted version of Creswell and Creswell's (2017) Narrative Data Analysis Framework (NDAF) (see Table 3.1). The discussions in the following sub-sections describe specific activities performed during the execution of this analysis framework.

Data preparation and organisation

Data preparation started when the think-aloud interview recordings were moved from the audio recorder to a computer for storage. For robust backup purposes, each recording was saved at this stage on a laptop, desktop, external DVD/CRW drive, Google Drive and a copy was given to the supervisor. For each recording, the researcher also recorded the date, time, and details of each participant as suggested by Marshall and Rossman (2016) and Step 1 of the NDAF (Creswell & Creswell, 2017). These audio recordings were then transcribed. All 10 audio recordings from the think-aloud interview sessions were transcribed by the researcher. The transcription process followed guidelines developed by the Minnesota Historical Society (2001).

They suggest using brackets (especially square brackets [ ]) to supply information not available on the recording but which is necessary for clarity; using proper spelling for slurred words ('gonna' is 'going'); removing false starts; removing stumbles; using ellipsis points (…) when a statement is not finished; eliminating some crutch words; and writing numbers and words and vice versa.

Table 3.1 – Narrative Data Analysis Framework

Step	Description	Activities
1	Prepare and organise the data	• Data transcription. • Data translation (if necessary). • Data cleansing. • Data labelling (i.e. structuring and familiarising).
2	Identify a coding plan	• Read, read, read, … (e.g. read and reread the transcripts). • Decide whether analysis should be guided by research questions (explanatory) or the data (exploratory) or both (mixed).
3	Sort the data into a coding plan	• Code the data. • Modify the coding plan (if necessary). • Enter the data (if Computer-Aided Narrative Data Analysis software is used).
4	Use the coding plan in descriptive analysis	• Put a range of responses or statements under the created nodes (as determined by the coding plan identified in Step 2). • Identify recurrent themes.

(Source: Adapted from Creswell & Creswell, 2017)

Transcription process

Literature (Bailey, 2008; Hart, 2015; Powers, 2005) indicates that transcribing audio tapes or recordings is time consuming. The researcher made three passes of the think-aloud recordings. First, he listened to each recording immediately after the session to make absolutely sure that the entire session was properly recorded. Second, he started typing all the words that were said on the voice recorder verbatim on a Word document. During this process, he had to pause, move back, and move forward in order to capture all that was said on the recorder. Third, when the entire recording or transcript was completed, he listened and re-listened to the whole recording while reading the transcript to confirm whether all the words were properly captured. During this process, he was able to form a better understanding of some sentences, and hence make corrections to words which were vaguely heard in the second transcription pass.

Cleansing of transcripts

After verbatim transcription of the audio recordings, the data was cleansed. Data cleansing is defined as the process aimed at enhancing the quality of data by searching for faults in the data (performing diagnosis) in order to repair it (by either correcting or deleting these faults) (Chu, Ilyas, Krishnan & Wang, 2016; Parcell & Rafferty, 2017; Van den Broeck, Cunningham, Eeckels & Herbst, 2005). Background noise, incomprehensible words, hanging words that distort meaning, unfamiliar terminology, blurred communication, mistyped words or typos, inappropriate punctuations, silences, overlapped speech and sounds, accents and dialects, and incomplete statements are all regarded as elements that can cause inconsistencies within transcribed data (Easton, McComish & Greenberg, 2000; Hinds, Vogel & Clarke-Steffen, 1997; ten Have, 2011). Since the participants had to verbalise their thoughts as part of the think-aloud process, the transcripts also contained numerous illogical and repeated statements. The researcher therefore decided to make use of fuzzy-validation instead of strict validation (which requires the complete removal of invalid or undesired responses) (Parcell & Rafferty, 2017). With fuzzy-validation, the researcher is allowed to correct some data if there is a close match or known answer. Parcell and Rafferty (2017) specifically mention “detecting and modifying, replacing or deleting incomplete, incorrect, improperly formatted, duplicated or irrelevant records” (p. 337) in their description of the fuzzy-validation process.

The decision to use fuzzy-validation was based on the following two reasons:

1. It was possible for both the interviewer(s) and interviewees to construct incomplete or illogical statements or sentences, as they had to do a lot of thinking and engagement throughout the interview process.
2. A lot of repetition in the uttered sentences or statements occurred due to the candid nature of the questions that were asked throughout the interview. This is especially because the probing questions were triggered by the responses that interviewees provided (i.e. the questions were not predetermined).

During the fuzzy-validation process, some modifications were made. Typical examples are writing abbreviations (e.g. isn’t, I’ll, I’m, and they’ve) in full; removal of verbal tics

(e.g. um, eh, and uh); representation of pauses with three dots (…); and removal of repetitions (Arksey & Knight, 1999; Gibbs, 2018; Minnesota Historical Society, 2001).

Coding plan identification

The researcher immersed himself in the data (Liamputtong, 2009; Marshall & Rossman, 2016; Holton III & Swanson, 2005; Thorne, 2000; Ulin, Robinson & Tolley, 2005) by listening and re-listening to the audio recordings numerous times, as well as intensively reading and re-reading the transcripts (see Step 2 in Table 3.2). He wanted to be completely familiar with the data (depth and breadth) before beginning the coding process (Braun & Clarke, 2006). Upon familiarising himself with the data, he decided on a coding plan where the analysis would be guided by the data as it relates to the first research question (see RQ3 (a) in Section 3.4.2.1). Codes were therefore created for every source code comprehension difficulty identified in the data.

Data coding

Data coding is defined as a method used to organise the data to help the researcher to be clearer about the underlying messages portrayed by the data and its salient features (Smith & Davies, 2010). As suggested by Saldaña (2013), the researcher performed data coding by highlighting and/or underlining sections/passages (i.e. words/keywords, sentences, paragraphs) from which difficulties with source code comprehension could be extracted. During this process, the researcher found no need to modify the coding plan. NVivo 12 Professional (for Microsoft Windows) was used for analysis of the 10 validated transcripts. At this stage, the transcripts were uploaded to NVivo (under Data) and the researcher then developed codes by creating several nodes (each equivalent to a class of difficulties). The names of the codes consisted of single words or simple phrases. Furthermore, the names of these nodes were continuously revised by combining some and/or renaming them. During this stage, it was also necessary to read and re-read the transcripts over and over again.

Descriptive analysis

During this stage, words, statements, and paragraphs (single and multiple) highlighted and/or underlined during coding, were extracted from the transcripts and moved to created nodes. This again required numerous re-reading of the transcripts in order not to miss any important meanings or details. Braun and Clarke (2006) define a theme

as something that captures or pinpoints some important information about the data set in relation to the research question. As such, some themes started emerging from the process of extracting and moving the relevant text. Continuing this process led to the emergence of recurrent themes. For each theme, frequencies of occurrence and transcripts from which these themes were extracted, were also visible on NVivo.

The results of the Phase 2 research activities are reported as part of Article 1 (see Chapter 4). The main outcome of Phase 2 was six usable SCC bottlenecks experienced by senior CS students.

3.4.3 Phase 3

3.4.3.1 Aim

In response to the six bottlenecks identified in Phase 2 (as part of Step 1 of the DtDs framework), Phase 3 focused on Step 2 of the DtDs framework. The main aim of Phase 3 was to uncover the explicit nature of steps and strategies that programming experts would follow in order to accomplish the tasks associated with one of the student-learning bottlenecks identified in Phase 2 and reported in Article 1 (*Bottleneck 6: Students are unable to reliably think their way through a long chain of reasoning required to comprehend a piece of source code*). Phase 3 was therefore set up to answer the following four research questions:

> ***RQ5 (a):*** *What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC?*
>
> ***RQ5 (b):*** *What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students?*
>
> ***RQ6 (a):*** *What are the explicit mental strategies (techniques and reasoning) that CS experts employ while comprehending source code?*
>
> ***RQ6 (b):*** *How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards narrowing the gap between expert and novice thinking with regard to efficient SCC?*

3.4.3.2 Data source management

The data source management approach in Phase 3 was a case study. This approach was selected because the researcher wanted to collect in-depth information from a small number of participants. Participants would be disrupted minimally to spend some time with the researcher when they were free from their other academic and research activities. The researcher also wanted to study a few participants in a conducive environment, but where he had some degree of control in terms of probing the participants where necessary in order to obtain as rich and thick descriptions as possible (Plowright, 2011; 2016b).

3.4.3.3 Population and sampling

The population for Phase 3 consisted of CS instructors who had experience in teaching programming on first-year level and, ideally, had at least some industry programming experience. These instructors were selected from a South African higher education institution. A sample of five instructors was selected from this population. This sample was purposeful (Cooper & Schindler, 2013) because the instructors were involved in the teaching of programming modules, and had at least three years of experience in teaching programming at first-year level (CS1 and/or CS2 modules). Two of these participants (P1 and P4) had more than 14 years of experience in the subject area, 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. For ease of reference, these participants will be referred to as the 'expert programmers' in the rest of this discussion. The sample could also be regarded as convenient (Patton, 2015), since the participants were in the proximity of the researcher, hence he could easily (physically, electronically or otherwise) reach them.

3.4.3.4 Data collection methods

The data collection methods for Phase 3 were 'asking questions' (through decoding interviewing) and 'making observations'. The asking-questions method was characterised by a higher level of mediation and a higher degree of structure because of the reasons alluded to in data collection methods for both Phases 1 and 2. Similar to observations made in Phase 2, the degree of structure with Phase 3 observations provided the researcher with a relatively high level of mediation and degree of

structure (Plowright, 2011; 2016b). Based on the selected data collection methods, two sets of narrative data resulted from the Phase 3 research activity: Transcriptions of the audio recordings made during each of the decoding-interview sessions, and a written record of the observations and notes (for future reference) made by the researcher.

3.4.3.5 Piloting of Phase 3

Using the same reasoning as provided in the discussion of the pilot study in Phase 2 (see Section 3.4.2.5), a pilot study was conducted with one participant who was a lecturer for one of the CS programming modules at the selected institution. The specific objective of the pilot was to assess the feasibility of all the logistics made in preparation of the real sessions. The sub-objective was to determine whether the probing questions would trigger any emotional responses from participants, as is typical of decoding interviews (MacMillan et al., 2016). One of the concerns in this regard was that DtDs interviews – where the interviewee is not the one who comes up with the bottleneck(s) – may be challenging, because according to the proponents of the DtDs framework (Middendorf & Pace, 2004), bottlenecks typically originate from the interviewee. However, by the time of the interviews, bottleneck(s) for this study had already been identified and refined (Lahm & Kaduk, 2016). The pilot participant completed the entire interview session in 1:32:32 minutes. In addition to the first two questions, she only managed to complete about half of the third question, however. Consequently, the researcher decided to retain only one question [Q6 – the most challenging question from the previous phases of this study (see Figure 3.3)]. The researcher further decided to schedule at least 60 minutes for each of the upcoming sessions.

3.4.3.6 Procedure

Five eligible participants (‘expert programmers’) were invited through verbal communication to take part in the individual decoding interviews for Phase 3. Justification for choosing these participants was already provided earlier (see Section 3.4.3.3). All the invited participants (100%) were able to show up for the interviews. In these interviews, the researcher played the role of the principal researcher, while a non-teaching CS researcher who had some decoding-interview experience, acted as the second interviewer. (Note: A more detailed explanation regarding the selection of

the second interviewer is provided as part of the methods discussion of Article 2. This individual had no other direct connection to the study). Each of the participants was first taken through the interview protocol (see Appendix E). After that, they were each asked the following question:

> *Suppose you are presented with a piece of source code on a piece of paper and asked to read/work through it to predict its output. Can you explain to us how you would go about doing that?*

In response, the participants proceeded to explain the process they would typically go through when having to comprehend any given piece of source code. Whenever the interviewers felt that the participant was not clearly verbalising all their mental operations, one of them would intervene with a probing question. After about 30 minutes, a specific SCC question (Q6 printed on a piece of paper) was presented to the participant and the following question was asked:

> *Assuming you are given the following question, how would you go about answering it?*

In response, the participants therefore had to verbally illustrate the general SCC process that they had previously explained. For each participant, the probing questions were triggered by their response to the above questions and everything else they said and did while attempting the given question. Although only 60 minutes were scheduled for each interview session, participants were told that they could take longer than that time duration, but no longer than 90 minutes. Participant 2 recorded the shortest time (01:05:09 minutes), while Participant 1 recorded the longest time (01:21:54 minutes). The rest of the sessions were completed as follows: Participant 3 recorded 01:10:20 minutes; Participant 4 recorded 01:08:44 minutes; and Participant 5 recorded 01:11:38 minutes.

3.4.3.7 Data analysis

The narrative data collected during the decoding-interview sessions was analysed thematically based on the adapted version of Creswell and Creswell's (2017) NDAF. The exact same procedure as outlined in the Phase 2 data analysis discussion (see Section 3.4.2.7) was followed. However, different data was extracted in order to address the Phase 3 research questions (see Section 3.4.2.1). After completion of the

transcription process, the five validated transcripts were imported into NVivo 12 for further analysis. The data was then coded by highlighting and/or underlining sections/passages (e.g. words/keywords, sentences, paragraphs) (Saldaña, 2013) for each cognitive process recognised in the data (Article 2), and the mental SCC strategies identified in the data (Article 3). The developed codes were then populated by moving the necessary text into them. Consequently, some themes started to emerge which revealed important information about the data set in relation to the research questions (Braun & Clarke, 2006). Continuing to populate the codes led to the emergence of recurrent themes. Finally, NVivo 12 was used to generate frequencies of occurrence for each of the developed themes. Using these frequencies and some of the excerpts from the transcripts, the data were put back together to create new meaning (Lewins & Silver, 2007). In reporting the results of Phase 3, RQ5 (a) and RQ5 (b) are covered as part of Article 2, while Article 3 addresses RQ6 (a) and RQ6 (b).

3.5 Trustworthiness

For readers to develop confidence and/or trust in any research study, a researcher must continuously take measures to safeguard the accuracy, consistency, and legitimacy of the research findings throughout design, data collection, analysis, interpretation, and reporting (Haworth & Conrad, 1997). The prominent approach used in evaluating investigations which are narrative in nature, consists of five key criteria, namely credibility, transferability, dependability, confirmability, and integrity (Guba, 1981; Schwandt, Lincoln & Guba, 2007; Wallendorf & Belk, 1989).

3.5.1 Credibility

The main issue in credibility is to establish whether the research findings can be seen as a true reflection of the information obtained from the participants' original data when viewed from the perspective of the participants involved in the research (Trochim, 2006). To ensure that the study was credible, the researcher made sure to file these documents (hard and soft copies of the test scripts from Phase 1 research activity; transcripts from the think-aloud interviews; and audio recordings of the interview sessions) for future cross-checking (Guba & Lincoln, 1982). These were also shared with the supervisor. All copies will be destroyed within five years after the completion

of this study. In further ensuring credibility of this study, the researcher used triangulation – multiple sources of data (senior students and expert programmers); methods of data generation and collection (observations, asking questions, and artefact analyses); and data collection instruments (questionnaire and interviews).

3.5.2 Transferability

To ensure transferability, a researcher continuously makes a series of judgements as to whether the study findings can be generalised to other contexts or settings where different participants are used (Trochim, 2006). Thick descriptions and the use of purposive sampling are cited as strategies that can be used to facilitate transferability (Bitsch, 2005). Using the decoding and think-aloud interviews, the researcher collected adequately detailed descriptions of the data in context. This data was reported with sufficient detail and precision, allowing readers to judge for themselves whether or not the findings are transferrable to other contexts. In order to maximise the acquisition of rich information and data from few participants, the researcher used purposive sampling for both the senior students and expert programmers.

3.5.3 Dependability

In dependability, the main issue is whether the research findings would be similar if the study was to be repeated in similar contexts and with the same participants (Lincoln & Guba, 1985). Bitsch (2005) shares the view that even if the findings may vary, such variations should be reasonable and easy to justify. To ensure that the study met these conditions with respect to the data collected through decoding interviews, the transcripts were sent to the five participants who took part in the interviews, after the experts' narrative data (i.e. transcripts) was cleaned. They were asked to indicate whether the content of the transcripts was a true reflection of what they shared during the interviews. Using this member-checking technique (Lincoln & Guba, 1985), all the participants approved the transcripts either conditionally or unconditionally. The conditional approval was due to some words that could not be deciphered from the audio recording. Consequently, these sections of the interview were excluded from the transcript.

Furthermore, in analysing all narrative data collected for this study, the researcher used a specific and direct approach (see Section 3.4.2.7) where most of the codes

were established from the themes identified in the literature and/or were guided by the research questions. This facilitated staying within the predetermined boundaries during the data coding process. Additionally, in analysis and reporting, the researcher was very careful with the quotes he used; he completely avoided using participants' words out of context and/or editing them with the objective to suit the arguments that he might have wanted to put forward. The researcher also employed the code-recode strategy as suggested by Chilisa and Preece (2005). By doing so, he coded the data more than once; the benefit was that he was able to ultimately formulate robust code names. In like manner, he employed the peer-debriefing strategy (also known as 'reality check') (Saldaña, 2013), where he debriefed with his supervisor on a regular basis, especially during the data collection, analysis, and reporting of the results. She (the supervisor) was very critical of each and every aspect of the data reported, interpretations made, and analysis presented.

Moreover, upon formulation of a step-by-step framework (consisting of 10 main steps) for efficient source code comprehension (see Article 3), the second decoding interviewer took part in checking the framework (i.e. part of validation); he indicated that the steps were well written and could be taught to students. To further evaluate and validate the proposed framework, the researcher arranged a validation meeting with five CS instructors (expert programmers and others). One of these participants took part in the original decoding interviews. Upon explaining the steps to the participants, they had an opportunity to solve two selected questions [Question 1 and Question 9 from the original set of 12 MCQs (Lister et al., 2004)] by applying the formulated steps. Question 1 was selected because it was found to be the easiest question from the previous phases of this study and was mainly used to familiarise the validation participants with the proposed steps. Question 9 was selected because it was not as straightforward as Question 1 – it involved some relatively significant requirement descriptions, and it had a lot of code lines to be interpreted; even the options that had to be chosen were actually lines of code (e.g. not necessarily ultimate values of variables). In solving the given SCC problems, validation participants were asked to put a check mark against each of the 10 steps and their sub-steps (e.g. to put a tick [☑] against a step/sub-step they used, and a cross [☒] against a step/sub-step they did not use) (see Article 3). Throughout the validation meeting, which took

the form of an open discussion, the participants provided constructive feedback that helped the researcher consolidate the proposed steps and finalise the framework.

3.5.4 Confirmability

In confirmability, a researcher continuously makes sure that the data and interpretations of research findings are derived from or grounded in the data, and hence can be confirmed or corroborated by other researchers (Lincoln & Guba, 1985). To ensure confirmability in this study, the researcher piloted the research activities of both Phases 2 and 3. He also interpreted and reported the study findings in such a way as to avoid bias at all costs. Furthermore, he had no personal inclination (i.e. was as neutral as possible) in the analysis of data and reporting of findings. To further enhance the confirmability of the results of this study, the analysis of data was continuously reviewed by the research supervisor. Moreover, for cross-checking (Guba & Lincoln, 1982), all the research-related records were kept throughout the research process and are available upon request.

3.5.5 Integrity

The focal issue of integrity is to ensure that the data interpretations made and recorded, do not in any respect contain elements of lies, evasions, misinformation or misinterpretations by participants (Wallendorf & Belk, 1989). Considering the nature of the interviews (both decoding and think-aloud), it was not possible for participants to tell lies, because it was all about solving given problems. The only questions that participants had to answer were to explain why they were doing certain actions or how they arrived at definite decisions – there was therefore a low possibility that participants could lie. Furthermore, there were no aspects related to social and/or cultural understanding or legal implications that could make participants uneasy to open up in the discussions. Additionally, participants provided information that the researcher never doubted or felt that it might not be correct, hence his scepticism measure of integrity was not exercised. Integrity was further safeguarded by not revealing the identity of participants in the reporting of the study findings; instead, pseudonyms were used for identification. Moreover, the interviewing approach used in this study was rigorously prepared and tested – specific protocols existed and were piloted before the real studies.

3.6 Ethical considerations

This study was guided by the research ethics code of the University of the Free State. Ethical clearance (see Appendix F) was obtained before any form of data collection commenced. Reflections of ethical issues in the three phases of this study are described as follows:

- **Phase 1:** The introduction of the questionnaire contained a statement to explain the purpose thereof. Participants were also informed of the approximate time needed to complete the questionnaire. These participants were further informed that data resulting from the questionnaire would only be used for research purposes. Likewise, the participants were assured that confidentiality and anonymity regarding information provided in the questionnaire would be respected to the maximum extent possible. Additionally, the participants were informed that their participation was fully voluntary and that they could withdraw at any time if they felt they no longer wanted to participate. Moreover, they were informed that completing the questionnaire or failing to complete it would not have any impact on any of the CS modules for which they were enrolled (see Appendix A).
- **Phase 2 and Phase 3:** Before data collection commenced, each participant was provided with a participant information sheet (PIS) in which details regarding the purpose of the study; why the research activity was conducted; what was required of each participant in the activity; as well as the potential benefits and risks, were explained. PIS for senior students (as participants in Phase 2) (see Appendix G) was different from the one for the expert programmers (as participants in Phase 2) (see Appendix H). After agreeing with all the information explained in the participant information sheets, each participant signed the consent form (see Appendix I) and carried on with the research activities as guided by the researcher.

Furthermore, the ethical confidentiality and privacy of participants' rights was protected to the maximum extent possible. In any part of the reporting in this thesis and related publications, pseudonyms have been used for anonymity instead of using the real names of the participants. Long before taking part in any research activity in this study, all the participants were informed that their participation was voluntary and

that they were perfectly free to withdraw from the research activities at any time without any form of penalty whatsoever. Participants who did not show up for some of the research activities were not followed up, because the researcher assumed that they did not want to participate in the study. Moreover, the researcher expressed his willingness to share the summary of the study findings with participants once the study was completed.

3.7 Summary

Within the realm of the DtDs-based research design, this study followed an integrated approach based on Plowright's (2011) FraIM. Within this framework, the focus was on collecting narrative and/or numeric data by means of observations, asking questions, and/or artefact analysis. In this chapter, the rationale for the selected research design and research methods, as well as the various strategies used and decisions made to answer all the research questions of this study, has been provided. The chapter also provided detailed descriptions of issues relating to purpose and procedure, coupled with the selection of data source management strategies; population and sampling; data collection methods; and data analysis techniques, as well as the nature of the resultant data from the three phases of the study. A discussion on how issues relating to ethics and trustworthiness were addressed in this study, concludes the chapter.

In the next three chapters, the three research articles that were prepared for this thesis, are presented. Article 1 covers the research activities of Phases 1 and 2. The data set that transpired from Phase 3 was used to inform the discussions in both Article 2 and Article 3. Table 3.2 provides a mapping to show which of the research questions (as stated in Chapter 1) are covered in each of the articles. It should be noted that each article is presented as a stand-alone unit without any cross-referencing to the rest of the thesis report. Each article is formatted according to the guidelines of the specific publication for which it was prepared. It should also be noted that each article was/will be published under the names of both the researcher who conducted this study and the study promoter. Consequently, there are some instances where the word ‘we’ are used in reference to the dual authorship (instead of constant reference to ‘the researcher’ or ‘the first author’). The use of the word ‘we’ should therefore not be regarded as an indication that the research was conducted by both

authors. All data collection and data analysis activities for this study were solely conducted by the researcher (unless specifically indicated otherwise), with inputs from the study promoter where deemed necessary.

Table 3.2 – Research questions covered by articles

Article	**Research Questions**
Article 1	RQ4 (a): What are the major SCC difficulties experienced by senior CS students? RQ4 (b): How can knowledge of these difficulties be used to identify SCC bottlenecks that should ideally be addressed in introductory programming courses?
Article 2	RQ5 (a): What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC? RQ5 (b): What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students?
Article 3	RQ6 (a): What are the explicit mental strategies (techniques and reasoning) that CS experts employ while comprehending source code? RQ6 (b): How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards narrowing the gap between expert and novice thinking with regard to efficient SCC?

Chapter 4 – (Article 1)

Decoding source code comprehension: Bottlenecks experienced by senior Computer Science students[1]

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

1 Introduction

Despite the continuous efforts of committed instructors to share the intricacies of their academic disciplines and their students' desperation to succeed, many students still struggle to master course material [31]. The specific points where students' learning gets interrupted can be referred to as bottlenecks [10,28]. A bottleneck typically occurs when students are unsure about how to approach a problem and consequently follow inappropriate strategies [31]. In an attempt to assist instructors in addressing students' learning bottlenecks, Middendorf and Pace [28] devised the Decoding the Disciplines (DtDs) paradigm. One of the underlying principles of this paradigm is that each discipline has unique ways of thinking [28]. 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 [31]. Despite the recent uptake in decoding research conducted in other disciplines [39,42], 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 [3,11]. One such difficulty – which has been researched extensively – relates to the way in which students (also referred to as novice

[1] An edited version of this article was published as: Khomokhoana, P. J., & Nel, L. (2020) Decoding Source Code Comprehension: Bottlenecks Experienced by Senior Computer Science Students. In: Tait B., Kroeze J., Gruner S. (eds.) ICT Education. SACLA 2019. Communications in Computer Science, vol 1136. Springer, Cham. https://doi.org/10.1007/978-3-030-35629-3_2

programmers) interpret pieces of source code [8,23]. This action – commonly referred to as source code comprehension (SCC) – is regarded a vital skill that novice programmers have to master [38]. Most of the previous SCC studies, however, focused on the evaluation of difficulties that students enrolled for introductory programming courses experience [26,37]. Pace [31] 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 Related Work

The first step of Middendorf and Pace's [28] seven-step DtDs framework is to identify students' learning bottlenecks. The identification of discipline-specific bottlenecks allows instructors to identify specific areas in a module where they need to seriously intervene in order to facilitate maximum learning [29,31]. 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 [31]. Within the DtDs paradigm [28], 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 [29], instructors themselves identify bottlenecks based on specific student problems they discover during their teaching of a specific module [33]. Instructors can also identify bottlenecks by focusing on a single assignment. In the History discipline, Pace [31] identified a specific difficulty while grading a writing assignment, while Shopkow [39] 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, Richert [19] identified creating Entity Relationship diagrams, reasoning in MySQL and dualism as the main student learning bottlenecks. At Indiana State University, Menzel [27] used her vast experience in teaching an introductory CS module to identify recursion (a threshold concept in CS [37]) as the main bottleneck that her students experienced. For a follow-up module, her colleague Adrian German [14] 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 [42] 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 [41] first asked instructors to identify potential bottlenecks. Their eventual selection of the top bottlenecks was based on the responses of 4th year Taxation students who were asked to rate the 40 potential bottlenecks 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 [10,40] have used individual interviews with instructors to identify common discipline-specific bottlenecks. Wilkinson [44] 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. [36] 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 [29] 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 [26] 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 [43]. In an attempt to further understanding of the difficulties experienced by students, the McCracken group [26] 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 [23] 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" [23] (p. 132).

The questions that the ITiCSE 2004 working group [23] 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 [18] 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 [13] found arrays to be featuring among the top three difficulties experienced by students. Other studies [2,24] have also identified arrays as a challenging concept for novice programmers.

All the ITiCSE 2004 questions [23] 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 [30], many novice programmers struggle to comprehend basic control structures. Various studies have reported the specific difficulties that students experienced while interpreting looping (repetition) structures [5,16,18,24]. Garner et al. [13] 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 [17]. Programming logic involves executing statements contained in a given piece of code one after another in the order in which they are written. Though still logical and correct, there are some programming control structures that may violate this execution order [9]. It is therefore not surprising that students struggle with logical reasoning in solving computer programming related problems [5]. The logical flow of the source code statements is closely related to the control flow of such statements [13]. This implies that for programmers to fully comprehend a computer program, they 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 [23]. It is therefore especially novices who struggle to follow a program's execution [11,35] and control flow [13].

As the proponents of the DtDs paradigm [28] argue that bottlenecks directly relate to difficulties hindering the learning of many students, 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 unclear: Where exactly are students getting stuck? Why are they getting stuck? What are they doing wrong? Which strategies do they resort to when they get stuck? More in-depth knowledge regarding the nature of these difficulties can thus be invaluable in determining teaching and learning gaps related to SCC.

3 Research Methods

3.1 Design

Within the scope of a DtDs-based research design, the study described in this paper followed an approach based on Plowright's [34] 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 [28] where these students were experiencing SCC difficulties with the goal of identifying common and useful SCC bottlenecks.

3.2 Phase 1 Participants, data collection and analysis

The sample for Phase 1 consisted of the 40 students registered for the 3rd year Internet Programming module. The selection of this sample can be described as both purposeful and convenient [32]. 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 [23]. 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 [6]] 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, the 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 first 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 followed the approach suggested by Creswell and Creswell [7]. Upon data transcription, the principal researcher cleansed the data by searching for faults and repairing them accordingly [45]. 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) [45]. 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 [25] 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 the 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 [22] recommends the use of 'units 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 domain 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. 4.1). (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 are grouped into three categories: arrays, programming logic, and control structures.

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. 4.1. Question 3 from the set of 12 MCQs

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 participants had difficulties to interpret simple array indices with a total of nine occurrences identified. Participant 1 (P1) had the most difficulties in this regard, with three occurrences identified. In her interpretation of `b[i]`, she regarded `i` as a value contained in array `b` instead of 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.

Array length. The length of an array refers to the maximum number of values that can be stored in a given array. Three 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 the values `true` or `false`. Five occurrences of Boolean array difficulties were identified, with P7 being the most challenged (with three identified occurrences). Overall, the identified difficulties ranged from the declaration of the Boolean array to basic understanding regarding the effects of operations performed on such arrays.

P7 got stuck at the Boolean array declaration in Line 2 and 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[i]` 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. 4.1). 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. This difficulty, which was observed with three participants, typically arises when programmers misinterpret programming logic [20]. Due to P1's struggle to interpret the array indices (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 algorithmic 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 (non-algorithmic) 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 previously observed by both Fitzgerald, Simon and Thomas [12] and Lister et al. [23], 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 given code. At that stage, he was unable to show how he arrived at the chosen answer and had to be prompted by the interviewer to re-explain his reasoning.

Mathematical expressions. When a line of code contains a mathematical expression, the misinterpretation of an operator can interfere with the comprehension of program 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 the 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 [16], it is not surprising that some participants experienced difficulties in this regard. However, one area of concern is the level of difficulty that these senior students experienced in comprehending basic `for` repetition structures.

5 Identification of 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. [23] 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 [2, 13, 18, 24]. Our students particularly struggled to interpret the array indices – 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. [13] also observed. While some students failed to recognise when and how to terminate the loops [16], 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 [21] 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 [15], 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 [40] 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. [23], 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 [4] or fail to follow the source code logic [1]. 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" [40] 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 CS 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 [28], future research is needed, firstly 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 to receive feedback on their efforts, instructors can assess students' efforts to determine whether they have benefited from the implemented strategies or not. The ultimate goal of this suggested research protocol is to help students to master the mental actions they need to be successful in the CS discipline.

7 References

1. Alston, P., Walsh, D., Westhead, G.: Uncovering "Threshold Concepts" in Web Development: An Instructor Perspective. ACM Trans. Comput. Educ. 15(1), 1–18 (2015). https://doi.org/10.1145/2700513
2. Anyango, J. T., Suleman, H.: Teaching Programming in Kenya and South Africa: What is difficult and is it universal? In: Proceedings of the 18th Koli Calling International Conference on Computing Education Research. ACM, Koli (2018). https://doi.org/10.1145/3279720.3279744
3. 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). https://doi.org/10.1109/TLA.2017.8070426
4. Boustedt, J., Eckerdal, A., McCartney, R., Mostrm, J.E., Ratcliffe, M., Sanders, K., Zander, C.: 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). https://doi.org/10.1145/1227504.1227482
5. 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)
6. Charters, E.: The use of think-aloud methods in qualitative research: An Introduction to think-aloud methods. Brock Educ. 12(2), 68–82 (2003). https://doi.org/10.26522/brocked.v12i2.38
7. Creswell, J.W., Creswell, J.D.: Research Design: Qualitative, Quantitative, and Mixed Methods Approaches. Sage, Thousand Oaks (2017)
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). https://doi.org/10.1145/3105726.3106190
9. Deitel, P.J., Deitel, H., Deitel, A.: Visual Basic 2012 How to Program. Pearson Education, Inc., Hoboken (2013)
10. 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). https://doi.org/10.2307/25095328
11. Du Boulay, B.: Some difficulties of learning to program. J. Educ. Comput. Res. 2(1), 57–73 (1986). https://doi.org/10.2190/3lfx-9rrf-67t8-uvk9
12. 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). https://doi.org/10.1145/1089786.1089793
13. Garner, S., Haden, P., Robins, A.: My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In: Australasian Computing Education Conference, pp. 173–180. Australian Computer Society, Inc., Newcastle (2005)
14. German, A., Menzel, S., Middendorf, J., Duncan, F.J.: How to decode student bottlenecks to learning in Computer Science (abstract only). In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, p. 733. ACM, Atlanta (2014). https://doi.org/10.1145/2538862.2544228

15. Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., Zilles, C.: 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). https://doi.org/10.1145/1352135.1352226
16. Grover, S., Basu, S.: Measuring Student Learning in Introductory BlockBased 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). https://doi.org/10.1145/3017680.3017723
17. 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
18. 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)
19. IUBCITL: Team-Based Learning For Practice and Motivation (2016). https://www.youtube.com/watch?v=1obB-n6JZ8k. Accessed 18 Oct 2018
20. 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). https://doi.org/ 10.1145/3137065.3137085
21. 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). https://doi.org/10.1145/2676723.2677219
22. Klenke, K.: Qualitative Research in the Study of Leadership, 2nd edn. Emerald Group Publishing Limited, Bingley (2016)
23. Lister, R., Seppl, O., Simon, B., Thomas, L., Adams, E.S., Fitzgerald, S., Sanders, K.: A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bull. 36(4), 119–150 (2004). https://doi.org/10.1145/1041624.1041673
24. Malik, S. I., Coldwell-Neilson, J.: A model for teaching an introductory programming course using ADRI. Educ. Inf. Technol. 22(3), 1089–1120 (2017). https://doi.org/10.1007/s10639-016-9474-0
25. Marshall, C., Rossman, G.B.: Designing Qualitative Research, 6th edn. Sage Publications, Inc.,Thousand Oaks (2016)
26. McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Utting, I.: 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). https://doi.org/10.1145/572139.572181
27. Menzel, S.: ISSOTL 2015: Recursion as a Bottleneck Concept (2017). https://www.youtube.com/watch?v=iNvQlm9phEI. Accessed 2 Sept 2018
28. Middendorf, J., Pace, D.: Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. New Dir. Teach. Learn. 98, 1–12. Wiley Periodicals, Inc., Hoboken (2004). https://doi.org/10.1002/tl.142
29. Middendorf, J., Shopkow, L.: Overcoming Student Learning Bottlenecks: Decode Your Disciplinary Critical Thinking. Stylus Publishing, LLC, Sterling (2018)
30. Milne, I., Rowe, G.: Difficulties in Learning and Teaching Programming – Views of Students and Tutors. Educ. Inf. Technol. 7(1), 55–66 (2002). https://doi.org/10.1023/A:1015362608943
31. Pace, D.: The Decoding the Disciplines Paradigm: Seven Steps to Increased Student Learning. Indiana University Press, Bloomington (2017)
32. Patton, M. Q.: Qualitative research & evaluation methods: Integrating theory and practice, 4th edn. Sage Publications, Thousand Oaks (2015)
33. Pinnow, E.: Decoding the Disciplines: An Approach to Scientific Thinking. Psychol. Learn. Teach. 15(1), 94–101 (2016). https://doi.org/10.1177/1475725716637484
34. Plowright, D.: Using mixed methods: Frameworks for an integrated methodology. Sage Publications, London (2011)
35. Qian, Y., Lehman, J.: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. ACM Trans. Comput. Educ. 18(1), 1–24 (2017). https://doi.org/10.1145/3077618
36. 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). https://doi.org/10.15173/ijsap.v1i1.3061
37. 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)
38. 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). https://doi.org/10.1287/isre.6.3.286
39. Shopkow, L.: How many sources do I need? Hist. Teach. 50(2), 169–200 (2017). http://www.societyforhistoryeducation.org/pdfs/F17 Shopkow.pdf
40. 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)

41. 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)
42. Verpoorten, D., Devyver, J., Duchteau, D., Mihaylov, B., Agnello, A., Ebrahimbabaye, P., Focant, J.: Decoding the disciplines – A pilot study at the University of Lige (Belgium). In: The 2nd EuroSoTL conference, pp. 263–267. EuroSoTL, Lund (2017)
43. Whalley, J.L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P.K.A., Prasad, C.: 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)
44. Wilkinson, A.: Decoding learning in law: Collaborative action towards the reshaping of university teaching and learning. Educ. Media Int. 51(2), 124–134 (2014). https://doi.org/10.1080/09523987.2014.924665
45. Willes, K.L.: Data Cleaning. In: Allen, M. (ed.) The SAGE Encyclopedia of Communication Research Methods. Sage Publications, Inc., Thousand Oaks (2017)

Chapter 5 – (Article 2)

Decoding the explicit cognitive strategies of expert instructors: Mental scaffolding techniques for efficient source code comprehension[2]

ABSTRACT

Many novice programmers fail to comprehend source code and its related concepts in the same way that their instructors do. As emphasised in the Decoding the Disciplines (DtDs) framework, each discipline (including Computer Science) has its own unique set of mental operations. However, instructors often take certain important mental operations for granted and do not explain these explicitly when modelling problem solutions. Better understanding of the nature of the cognitive processes and related strategies employed by experts during source code comprehension (SCC) could ultimately be utilised to identify the 'hidden' mental steps. Within the realm of the DtDs framework, this study employed decoding interviews, followed by thematic data analysis, to uncover a variety of explicit cognitive processes and related strategies utilised by a select group of experienced programming instructors during a SCC task. The insights gained were then used to propose a set of mental scaffolding techniques for efficient SCC. Programming instructors can use these techniques as a SCC teaching aid to convey expert ways of thinking more explicitly to their students. Insight into the general cognitive strategies utilised by expert programmers is an important step towards further exploration of the more detailed step-by-step procedures followed by experts during SCC.

Keywords:
Source-code comprehension, cognitive processes, decoding the disciplines, Computer Science Education, mental scaffolding

---

[2] Publishable manuscript.

Categories:
• **Social and professional topics~Computer science education** • *Social and professional topics~CS1*

1 INTRODUCTION

Source code comprehension (SCC) is a core skill that many Computer Science (CS) students continue to struggle with [1, 2]. SCC generally refers to the reading and interpreting of pieces of source code [3, 4]. Some authors [5, 6] describe it as a skill that requires efficient application of a series of complex cognitive processes. Due to the complex nature of SCC, a 'scaffolding' process [7] – where instructors gradually guide their students in mastering these cognitive processes – could be instrumental in getting students to perform tasks that were initially beyond their capacity. According to [8], each academic discipline has its own distinctive set of mental operations that stakeholders follow when performing discipline-specific tasks. The explicit nature of these operations is, however, often so deeply buried in the unconscious minds of the discipline experts that it causes an 'expert blind spot' [9]. As a result, vital mental operations become so natural to the experts that they often omit crucial and even quite simple steps when explaining concepts and procedures to others [10]. Such omissions during instruction can lead to novices developing mental blocks ('bottlenecks') in mastering the steps involved in completing discipline-specific tasks [9].

Decoding the Disciplines (DtDs) [8] is a seven-step process that can be used to overcome specific student-learning bottlenecks. After identification of a specific bottleneck (Step 1), the disciplinary unconsciousness is systematically decoded in order to reveal the explicit steps followed by experts (e.g. instructors) when performing tasks related to the identified bottleneck (Step 2). These steps are then broken down into their component parts and each operation is modelled in a way that will be understandable to students so that it can be used to facilitate effective learning and understanding (Step 3). Students are then provided with opportunities to practise the modelled operations and get feedback on their efforts (Step 4). Throughout the process, specific strategies are employed to motivate students to follow the modelled operations (Step 5) and to assess whether they have mastered these operations (Step 6). In line with the principles

of the Scholarship of Teaching and Learning [11], DtDs practitioners are encouraged to then share with other stakeholders what they have learned (Step 7).

Within the CS discipline, one of the most common and significant SCC bottlenecks identified [12, 13], relates to students' inability to reliably work their way through the long chain of reasoning required to comprehend a piece of source code. Instead of using Step 2 of the DtDs process to uncover the explicit steps that experts follow in dealing with tasks related to this bottleneck, the complex cognitive processes required to comprehend source code [6] have led us to first focus on exposing specific strategies used within these cognitive processes. If the strategies revealed by expert programmers during a SCC task are assumed as typical of the basic cognitive operations required for efficient SCC, then these ways of thinking could point to crucial techniques employed by experts during SCC. More explicit awareness of the techniques instructors typically use to comprehend source code could help them to avoid their own 'blind spots' when sharing SCC strategies with their students. These techniques could also be used as mental scaffolds to help students overcome related 'bottlenecks' [8] and better prepare them for SCC tasks. Within the DtDs context, identification of general cognitive strategies could serve as a pre-step for revealing the explicit steps followed by experts in comprehending source code. This paper therefore attempts to answer the following two questions:

- What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC?
- What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students (as novice programmers)?

The remainder of this paper is organised as follows: Section 2 provides an overview of the basic cognitive processes involved in processing information and how these relate to SCC. In the discussion of the research design and method in Section 3, detail is provided about the selection of the experts and the SCC questions used in the decoding interviews. Important detail is also provided about the nature of the interviews and the way in which the interview questions were

linked to cognitive processes in the data analysis. The presentation of findings (Section 4) takes place according to the main categories of cognitive processes recognised during analysis of the transcribed interview data. As part of the discussion (Section 5), we summarise the experts' cognitive process strategies and propose techniques that could be used as part of a mental scaffolding process by instructors while modelling efficient SCC strategies to their students. Conclusions are presented in Section 6.

2 BASIC COGNITIVE PROCESSES AND THE RELATION TO SCC

Cognitive processes are defined as procedures that process all the information (multiple, complex or otherwise) human beings receive from their surrounding environment [14]. The processing is done with the objective to transform the information into easily manageable cognitive tasks [15]. The basic cognitive processes discussed in the following sub-sections include attention, perception, memory, reading, speaking and listening as well as those processes related to reflective cognition. All of these processes are highly relevant to SCC and also to this study, as will be illustrated in the discussion.

2.1 Attention

Attention is a cognitive process in which certain things are selected (triggered by single or multiple stimuli) from a host of available possibilities at a certain point in time while doing something [16]. When applying attention, one can use either the intensity or selectivity component [17]. The intensity component enables a person to sustain concentration on one activity over time (sustained attention). The selectivity component enables a person to choose to focus on competing stimuli. This means that the attention may be divided and therefore not fully focused on the current activity.

During SCC, expert programmers often focus their attention on complex lines or sections of code [18]. These complex sections of code are likely to contain dynamic representations such as literals, comparisons, operators, and keywords [19]. Experts will typically identify these sections by quickly scanning through the code from top to bottom [20].

Irrespective of type, attention is normally dependent on information that is relevant to the current task a person is performing [21]. Van Someren, Barnard and Sandberg [22] point out that in performing almost any task/activity, there will be irrelevant and distracting stimuli. As such, any individual involved in performing such a task should focus their attention by being conscious (e.g. recognise, differentiate, assemble things together, be assertive, be orientated and even suggest), alert, aware, and responsive (reactive) in order to be successful [23].

2.2 Perception

Perception refers to the process of acquiring information from the world around us and transforming it into real experiences [24]. Preece et al. [21] point out that perception is a complex process that also involves other processes such as memory, attention, and language. Although perception can be used under normal circumstances, human beings have a tendency to use their perception when there is a breakdown in other cognitive processes [25]. Perception can also change while a specific task is performed. During this process, perceptual span increases when a person obtains useful information, while it decreases when encountering information that is difficult to comprehend [26]. Choi and Gordon [27] argue that when perceptual span decreases, human beings will typically skip such troublesome information (e.g. words or text) and jump to sections that are not bothersome. With regard to comprehending source code, programmers can have different perceptions based on whether they employ a bottom-up [28]; top-down [29]; knowledge-based [30]; systematic [31]; micro [32]; as-needed [31]; or integrated [33] source code comprehension strategy.

2.3 Memory

Memory is a cognitive process that involves the recall of different kinds of knowledge that guide human beings to act or react in a specific way to certain stimuli [21]. Knowledge can be recalled from either the long- or the short-term memory [34, 35]. It is, however, important to note that not all knowledge is stored in memory. A filtering process is used to decide what is processed and stored and what is not [36]. Since cognitive processes tend to overlap, the more attention a person pays to a certain aspect, the more likely it is that this aspect will be

remembered [21]. Programmers typically use strategies such as reading/re-reading specifications; thinking of possible test cases; and reasoning aloud to enhance the memorability of concepts [37, 38]. Other strategies, such as highlighting or colouring some lines of code or text [39]; writing comments [40]; pattern recognition [37]; and making drawings or annotations (doodles) [12] are often utilised by programmers to readily and easily remember or determine the values of variables or other information without heavily engaging their memory.

2.4 Reading, speaking, and listening

Reading, speaking, and listening are three interrelated cognitive processes [21] that can be identified through facial expressions, vocal behaviour, verbal consent, pauses or segregates (e.g. 'hmm'), posture or stance, eye behaviour, hand gestures, and head movements [41, 42]. A person can typically understand something (e.g. a given piece of code) well by using any one or a combination of these processes. An attempt to comprehend something that is written down and spoken, requires more cognitive effort than just listening to it [43]. However, many people prefer to listen, as they consider it the easiest mode to comprehend something. In contrast, if something is written down, it is easier to re-read the information if it is not understood [21]. Analogous to strategies used in other cognitive processes – in reading source code, programmers will mark some lines of code [39]; write comments [40]; draw illustrations [12]; and/or read through the code multiple times [38] in an attempt to enhance their comprehension. During code reading, experienced programmers typically concentrate on the semantic features of the code, while non-experienced programmers tend to focus more on the syntactic features [33].

2.5 Reflective cognition

Planning, reasoning, and decision making are interrelated cognitive processes that enable individuals to reflect on their cognition [21]. During reflection, initial thoughts and/or responses should be examined carefully before conclusions can be made. In doing so, a person will typically ask the following questions [44, 45, 46]:

1. What should I do? (Cognitive planning)

2. What alternative courses of action do I have available? (Cognitive planning)
3. Which alternative courses of action should I select to use? (Cognitive reasoning)
4. Why should I use these (selected) alternative courses of action? (Cognitive reasoning)
5. What are the consequences of using these alternatives (selected)? (Cognitive decision making)

Inherently, questions 1 and 2 form part of cognitive planning. In addressing question 1, individuals actively and consciously engage their thought processes and use all resources available to them, such as discussions with others or using artefacts (e.g. books, papers, and the Internet) [21]. This is done with the objective to better understand the nature of the task in question in order to avoid ill-informed comprehension [47]. With regard to resources, Lister et al. [12] recommend that a programmer should make some drawings or annotations (artefacts) in order to better comprehend source code. According to Hayes-Roth and Hayes-Roth [48], question 2 is addressed in two stages. Firstly, a person decides in advance "a course of action aimed at achieving some goal" (pp. 275-76). Secondly, the execution of the plan is continuously monitored and guided to ensure success. This implies that the current course(s) of action can be revised over time based on new conditions encountered in the subsequent parts of the task at hand (e.g. SCC) [49].

Questions 3 and 4 essentially constitute cognitive reasoning. According to Evans [50], an ability to arrive at the preferred alternatives involves some intelligent thinking. This implies that it is not the solution that is retrieved from the memory, but the relevant information. A person then needs to work out how best to apply it. Use of connectives such as *and*, *if*, *or*, *all*, *some*, *none* and *not* can be used as a basis in making some cognitive reasoning decisions in order to arrive at a solution to a problem [51]. It is important to note that during the cognitive reasoning process, a person (e.g. programmer) creates logical as well as systematic arguments and makes judgement based on these arguments [52].

Evaluating different arguments to decide which one is the best option involves actively and exhaustively processing information to ultimately decide on cost-effective courses of action for the task in question [53].

Cognitive decision making is addressed by question 5. According to [21], addressing this question involves working through different scenarios and gauging the good as well as bad points of each alternative. The 12 multiple-choice questions (MCQs) used in the study by Lister et al. [12], are examples of situations where different scenarios – in this case missing pieces of source code – are weighed against each other. Measures to mitigate unfavourable elements for each alternative are identified and documented at this stage. To make decisions, a person does not necessarily have to consider all details included in the text or scenario. Instead, the focus can be placed on only a few key indicators [21]. By doing so, a person provides justification for all decisions arrived at [45].

Planning, reasoning, and decision making can be regarded as steps in the process of solving a problem. This process is characterised by certain actions that a person performs prior to and throughout solving a problem. Due to the cognitive nature of problem solving, a person should continuously engage and stimulate their thought processes when solving a problem [54]. To be successful in problem solving, Frederick [55] recommends that people should have "the ability or disposition to resist reporting the response that first comes to mind" (p. 35). This emphasises the argument that, after identifying a solution to a programming problem, the solution should be evaluated, implemented, and re-evaluated. These stages should also be revisited frequently during the iterative implementation of the solution and the discovery of more knowledge that was not apparent to the programmer [56].

Having provided some background on what constitutes the aforementioned cognitive processes, the next section discusses the research design and the procedure that was followed to uncover the cognitive processes and related strategies employed by expert programmers in this study.

3 RESEARCH DESIGN AND METHOD

The design of this study was narrative in nature and focused on the 'asking questions' data collection strategy, as described in Plowright's [57] Frameworks for an Integrated Methodology (FraIM). A case study was deemed the most appropriate data source management strategy, since only a small number of participants would be used. The population included CS instructors from a selected South African higher education institution. The sample consisted of five instructors who were purposefully [58] selected based on the fact that they were all experienced CS instructors who had been involved in teaching programming to novices (as part of CS1 and/or CS2 courses) for at least three years. 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 [59], since the selected participants were in the proximity of the principal researcher (the first author) and could therefore be reached easily.

3.1 Data collection

As part of the 'asking questions' data collection strategy, primary data was collected by means of decoding interviews [8] and supplemented by a short questionnaire. Experts are characterised as individuals who perform critical thinking tacitly and implicitly in their own disciplines [8, 9]. As such, the aim of the decoding interviews was to uncover the explicit mental steps that expert programming instructors would go through in order to accomplish tasks that students find difficult to execute. A decoding interview is typically conducted by at least two interviewers [9]. Given the format of a decoding interview, a single interviewer might get lost in the details, while two minds could better keep the interviewing process on track [60]. During the interview process, both interviewers should be able to verbalise their thinking; challenge the explanations given by the interviewees; and summarise their thinking back to the interviewees on an abstract level [61]. For this reason, Middendorf and Pace [8] describe the interview process as the most intellectually demanding of all the DtDs steps.

Since members of the same discipline have a tendency to share common expert blind spots, Pace [9] recommends that the second interviewer should ideally come from outside the discipline. The second interviewer is then more likely to see when a specific mental step has not been fully explained. However, given the complex nature of the cognitive processes involved in SCC and our inability to find a suitable person with relevant decoding interview experience from outside the discipline, we had to make an alternative arrangement. For the decoding interviews in this study, the principal researcher acted as the principal interviewer, with the support of a non-teaching CS researcher who had some decoding interview experience as the second interviewer.

3.2 Data collection procedure

All participants completed an informed consent form (as stipulated in the ethical clearance authorisation granted by the institution) before participating in the decoding interview. The proceedings of each interview were audio recorded with the permission of the participant. In each interview, the participants were first asked to explain the process they would go through when they needed to comprehend any given piece of source code. Whenever the interviewers felt that the participants were not clearly verbalising all their mental operations, one of them would intervene with a probing question. After about 30 minutes, a specific SCC question was presented to the participants, asking them to verbally illustrate the general SCC process they had just explained in answering this question. Where necessary, further probing questions were asked. At the end of the interview session, the participants completed the short questionnaire to provide basic demographic data and information regarding their programming and teaching experience.

Although the original plan was to include three SCC questions in this part of the decoding interview, a pilot of the entire data collection procedure revealed that it would take too long and that sufficient data could be collected if just one question was used. Question 6 (see Figure 1) from the original set of 12 MCQs developed by the ITiCSE 2004 working group for their multi-national study of reading and tracing skills in novice programmers [12], was therefore selected. This question was identified as the second most challenging question in Lister et

al.'s [12] study. While the most challenging question (Question 12) mostly focuses on arrays, Question 6 covers a variety of programming concepts (including Boolean variables, `for` loops, array indexes, and `return` statements to terminate a `for` loop). In answering Question 6, the missing piece of source 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 it from the original Java to C# (the programming language that all the participants were familiar with). The line numbers as illustrated in Figure 1 were not part of the question given to participants and are only included here for ease of reference in the results discussion to follow.

3.3 Data analysis

Following the decoding interview proceedings, a narrative data-analysis approach as suggested by Creswell and Creswell [62] was used to transcribe the audio recordings made during the decoding interview sessions and to analyse the data. After transcribing the data, we cleansed it by searching for faults and repairing them [63]. As the discussions were open-ended, the transcripts also contained some illogical and repeated statements. We therefore decided to use fuzzy validation [64], which allowed us to make some corrections to the data if there was a close match or known answer. After this, we familiarised ourselves with the data [65] by listening and re-listening to the audio recordings numerous times, as well as intensively reading and re-reading the transcripts. This helped us to decide on a coding plan where the analysis was guided by the data as it relates to the first research question. At this point, we imported the five validated transcripts into NVivo 12 for further analysis. We then started to develop codes (by creating several nodes) for each cognitive process recognised in the data.

As suggested by Saldaña [66], we then coded the data by highlighting and/or underlining sections/passages (e.g. words/keywords, sentences, paragraphs) from which cognitive processes could be extracted (under the guidance of the theoretical guidelines as identified from the literature). We populated the developed codes by moving the necessary text into them. Consequently, some themes started to emerge which revealed important information about the data set in relation to the first research question [67].

Continuing with this process led to the emergence of recurrent themes. Finally, we used NVivo to generate frequencies of occurrence for each of the developed themes.

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;
```

Figure 10: SCC question used in decoding interview

4 FINDINGS AND INTERPRETATION

Given the large amount of data collected during the decoding interviews, this paper focuses on data collected during the time when the participants were tackling the given SCC question (see Figure 1). The discussion in the following sub-

sections focuses on the five cognitive process categories that were observed, as well as evidence of their occurrence.

4.1 Reflective cognition

The category for reflective cognition processes had the highest number of occurrences (73). Participant 1 (P1) employed cognitive strategies extensively in this category, with 25 occurrences. Findings on the four cognitive processes constituting this category are discussed next.

4.1.1 Cognitive planning

During the cognitive planning process, a person needs to decide on the course of action to follow in order to arrive at a solution to a problem [48]. P3 demonstrated some elements of planning, as is evident from the following excerpt:

> "*You have already told me what the output should be. So now I have a question: What is it supposed to do? But there is a mistake, what is missing that would create the correct output? Now I know I need to look at this code for an error. The first thing I need to do is figure out which thing is doing what and how they are working together, and then I can find the mistake – unless the mistake is something obvious, in which case I can quickly find it."*

It can be seen from this excerpt that P3 first familiarised himself with the problem specifications so that he would know what was expected of him. The problem statement gave him an idea of what the missing source code should do. He also inferred that there might be errors in some of the source code fragments that would make them fail to produce the desired output if they were to be placed in the `isSorted` method. Further, by recognising that some pieces of code may have conditions or errors that could disqualify them from being the correct piece of missing code, P3 was already deciding on what he would do to ultimately arrive at the correct piece of missing code. Moreover, this strategy could help him spend considerably less time to ultimately get to the final answer, as he seemed to be applying efficient planning.

4.1.2 Cognitive reasoning

The cognitive reasoning process requires a person to integrate all the information at their disposal in order to arrive at a solution to a problem [50]. P1 exhibited this type of reasoning when he started by focusing on the opening statement of the question and the method signature in Line 1 (see Figure 1). He examined and read the various aspects of the question and specifically employed a cognitive reasoning process based on what he found. For example, to decide on the parameters of the `isSorted` method, he read, interpreted, and comprehended the method signature of the question. This, in turn, enabled him to apply some reasoning throughout the tracing task process. The likelihood of identifying the correct missing source code could have been low if this method signature was not well understood.

Furthermore, P1 made logical and systematic arguments [51] as he proceeded with the interpretation of the question. Evidence of this argument is contained in the following excerpt:

> *"The `IsSorted` method will receive an array of integers. It will receive it in x and now I will need to look at each one of these options here to see what goes in the missing code."*

Referring to 'options' in the above excerpt is already an indication that P1 was working towards an elimination strategy [37] to get rid of incorrect options.

4.1.3 Cognitive decision making

During the decision-making process, a person will focus on those problem details that can improve understanding of the problem so that well-informed decisions are made in tackling the problem at hand [21]. For example, P1 made a strategic decision based on the length of the question:

> "*So, in such a case, when it is a long question like this, I would quickly scan through the options to see which one is most probably going be the correct one. And then I start with that one instead of doing the first one and run through the entire thing."*

As can be seen from this excerpt, P1 also employed an effective 'quick scan' strategy as suggested by Bauer et al. [53]. This strategy allowed him to start

working towards an answer at an opportune place, hence expediting the process of finding the right answer.

As another example of decision making, P3 familiarised himself with the actual code included in the question by reading through the pieces of source code in order to determine the meaning of each statement. In this regard, he said:

> *"Now I pick a set of inputs 0, 1, 2, and I see that we are obviously looping through the array. So, I can see that we are stopping at the second-last element. This is the case again where I do not need to look through this in great detail, because I can see that thing. That is easy access to my array, because you said it there. It is my array length minus 1, which means the second-last element and we are going less than that. So, I understand it to be saying that we are going through each element in the array up until the second-last element, and I do not have to look at that loop again."*

From the above excerpt, it is also interesting to note that P3 decided to use some test cases or sample values to identify the limits of the array in question. Moreover, recognising that lines 6, 14, 21, 28 and 34 were identical, eliminated the need for P3 to interpret each one of them. This means that he interpreted the statement on line 6 and applied that interpretation to all similar statements.

4.1.4 Problem solving

Given the close relation between planning, reasoning, and decision making as part of the problem-solving process [21], it is worthwhile to consider how one of our experts utilised each of these cognitive processes as part of her problem-solving strategy.

While reading through the problem specification, P2 started to formulate her problem-solving strategy by identifying what she was asked to do and what her first action should be. The following excerpt illustrates her cognitive planning process: "*So I first try to understand the question (reading the question) ... otherwise the method should return* `false`*, right! So which one of the following is the missing source code from the method* `isSorted`*? ... So what I am going to do is to scan through each of my options quickly and see if I can eliminate others very quickly.*"

Her main aim with this strategy was to quickly identify the '*obvious*' incorrect options so that she would not have to spend unnecessary time on in-depth interpretation of those code segments.

In her scanning of the initial lines of code (reading the code sequentially), she said: "*All of them [alternative answers] are returning values, all of them are determining Booleans. So, I can't eliminate an alternative based on that*". In this regard, she was using cognitive reasoning to formulate a set of elimination criteria. She further pointed out that these criteria were based on both the question and what she saw in the given code fragments. After considering the above, she continued her reasoning process to examine additional aspects of the code alternatives: "*Can I eliminate one? Yes or No? If I can't eliminate one, now I start looking in more detail*". This examination of the possible code options eventually led her to apply her decision-making skills when she tentatively marked option B as the possible missing piece of code: "*So I will mark B as a possible solution. Because at the moment, without going really in depth and using a test case, it looks to me like it will work*". However, instead of settling for option B right away, she continued as follows: "*So I will just go to B and I will check it again*". This is illustrative of the thoroughness strategy, as suggested by Fitzgerald et al. [37]. This is the strategy where a person, upon initially recognising an answer, checks further for the correctness or incorrectness of answers just to be convinced of the final answer.

As illustrated in this problem-solving example, P2 followed an informed problem-solving process that enabled her "to resist reporting the response that first came to her (sic) mind" [55, p. 35] . Moreover, from the observation notes and all the steps that P2 followed in answering the question, it could be seen that she had mastered the problem-solving skill.

4.2 Attention

The attention process is characterised by selecting single or multiple options from a host of available possibilities [16]. In this study, 52 occurrences of using the attention cognitive process were observed in the participants. P1 employed this category the most, with 21 occurrences. As an indication of paying attention, P1 uttered the following statement: "*Now I see that B and E do not have that*

declaration and it will only do a comparison in the if statement and then return a specific value". By using an attention cognitive process, he was therefore able to identify that only three of the options (A, C and D) included a declaration for the Boolean variable b.

P2 used her attention process to realise that all five options to the question were returning values, as is evident from the following excerpt: "*All of them are returning values, all of them are determining Booleans".* As the cognitive processes tend to overlap [21], she immediately switched from attention to decision making (a reflective strategy) by saying, "*so I cannot eliminate an option based on that".*

As an indication that P3 was paying attention while reading the question specifications, he said:

> "*First thing, I must make sure that I understand the question. So, 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, sorted – so those are words that immediately come to mind. Other words glue everything together."*

It should be noted that the 'boldfaced words' referenced by P3 were actually a different font face used to ensure that code statements and values would stand out from the normal sentence text. With his attention drawn to these words, P3 identified those as important sections [68]. He proceeded further by identifying even more 'keywords' (e.g. array, method and sorted) to help focus his attention while tackling the problem.

4.3 Reading, speaking and listening

As part of the decoding interview, participants had to listen to the interviewers' questions and verbally explain their SCC processes. For the purpose of this discussion, these listening and speaking actions are not regarded as part of the experts' natural SCC strategies. We instead focus on the 17 other occurrences of these cognitive processes as observed in the participants. These actions included body movements (hand, eye, and head); facial expressions; and the utterance of some words [41, 42]. P1 applied this category the most, with six occurrences.

While reading the specifications, P1 focused on the semantic processing of the method signature (see Line 1) to determine that `isSorted` was a static method of a `bool` type which would receive an array of integers. Similar to our other experts, he realised the importance of the method signature and chose to focus on semantics instead of syntax. This is in contrast to a novice programmer who would typically rely more on syntactic aspects or even choose to ignore the method signature and its parameters completely [33]. As further evidence of semantic processing, P3 referred to the specifications as a 'rule' that guides the entire process of working through the comprehension task and arriving at the desired answer. He explicitly shared the information that he gathered from reading the problem specifications. He also advised that it is recommendable to read problem specifications at least twice, as also suggested by [38].

P4 regarded speaking and listening as fundamental strategies in his way of doing things (incl. code comprehension): "*I always speak aloud! My wife came to me this morning. She said: 'You are getting mad'. She always hears me talking to myself*". He further confirmed that even for general reasoning, he uses the think-aloud technique. He argued that the technique helps him to go through the logic fast and also ensures that he does not skip the logic steps as he is listening to himself. This combination of cognitive reading, speaking, and listening strategies also allows him to commit the information to memory [21].

4.4 Memory

The memory cognitive process requires the use of strategies that make it easier for a person to remember and/or recall values or knowledge when necessary [21]. Many of the attention, reading, speaking and writing strategies as discussed above, also helped the experts in this study to enhance their memory. Based on this notion, 11 occurrences of this cognitive process were identified.

During the decoding interview, P2 was observed making some pen-holding hand gestures as if she was writing computations in the air. In response to a question asking if she was "*passing values in her head as she went through the code*", she replied "*Yes*" – confirmation that she was using her working memory. Essentially, it is not easy to remember things that are not written down. Being compelled to do so can lead to a working memory overload [69]. As further

evidence that P2 was doing some comparisons in her mind, she was observed comparing the last two elements of the sample array she created herself. When asked whether she considered the previous values, she said: "*I compared them in my mind*". P3 also did not show the test cases he used. However, when asked if he was doing "*the test cases in his mind*", he admitted that he was.

The behaviour exhibited by P2 and P3 is interesting, because Wiedenbeck [70] observed that experienced programmers required less mental attention. However, experts in this study mostly kept things in their working memory. A possible explanation for this type of behaviour could be that these things were still manageable (i.e. within the number 5 plus minus 2 items according to Miller [35]). When the information became too much to keep in memory, the experts resorted to other strategies [71] to help them remember values and/or keep track of the program logic.

Due to the prominence of the integer array x in Q6, most of the participants used some type of doodle [12] to represent the array elements (e.g. □□□□□). Others just wrote down the values of variables (e.g. '2, 7, 5'), plainly as an indication that they were using these arbitrary values as test cases to help them determine outputs for the given code fragments. To enhance his memory, P5 resorted to pattern recognition [37] after realising that the `for` loops in all five alternative fragments of code (Lines 6, 14, 21, 28 and 34) were identical. During the interview, P5 was asked the following question by interviewer 2: "*Did you use the pattern from option A, and apply it to option B?*" In response, P5 said: "*Yes, I did not have to check whether this condition makes sense again*". This confirms that he was using some recall of things or actions he did before. He went on to apply the same pattern(s) he had seen in the previous options, and to apply them in subsequent instances. Inherently, these strategies helped the expert programmers to easily recall variable values and/or important information when they wanted to use it.

4.5 Perception

Perception is often used as a strategy when other cognitive processes fail to help us make sense of what we are trying to understand or achieve [25]. With reference to this notion, a total of eight occurrences of the application of the perception

process were identified with the participants. P4 applied this process the most, with four occurrences. In most of these instances, the participants' perception caused them to focus on issues that were not directly related to the given SCC task.

One of the participants (P4) was concerned that the given question did not follow the naming conventions he was teaching to students: "*I will make sure that the name of the method, which is wrong here anyway, follows conventions. When using a predicate method, there strictly should be a capital, not a lower case".* P5 did not agree with the use of 'breaks' in a `for` loop. He notably indicated that he would rather have used a compound `while` instead of the `return false` to break out of the `for` loop (see Figure 1 - Lines 17 through 19): *"Well, we do emphasise that there are two ways to break out of the* `for` *loop, but we never use the one .... So, we do not use the breaks, though we have a* `for`*. What we do instead, is that we convert to a while loop with a compound condition. So, I would have it as* `b = true`*, while* `b` *and* `i < x.Length - 1`".

P1 was concerned that the given SCC question would be too difficult for novice programmers to answer. He expressed this by pointing out that the argument of the `isSorted` method contained in its signature [see `(int [] x)` in Line 1 of Figure 1] was "*a higher-level concept*". His concern was based on his knowledge of the content that is covered in elementary programming courses. Another element of perception raised by P1 was related to his confidence in the steps that he followed in answering the SCC question. In this regard, he said: "*If I have eliminated them, I am quite sure they are not necessary*". He therefore chose to ignore the possibility that he could have made a mistake (e.g. logical or otherwise) somewhere that might have caused him to arrive at an incorrect final answer.

It should be noted that the perceptions identified from the experts in this study, caused them to focus on issues that were not directly related to the SCC question they were answering. Consequently, these identified perception cognitive strategies can be regarded as unrelated to the aim of this paper, and were therefore not considered further.

5 COGNITIVE STRATEGIES IN SOURCE CODE COMPREHENSION

During the SCC task, our expert programmers mostly relied on four cognitive processes to efficiently comprehend the provided source code: reflective cognition, attention, reading and memory. For each of these cognitive processes, the experts followed very specific strategies. In the sub-sections to follow, we first summarise these cognitive strategies and then extract the specific techniques that could be used by instructors as part of a mental scaffolding process while modelling efficient SCC strategies to their students.

5.1 Reflective cognition

The experts interviewed in this study often asked themselves guiding questions, as also suggested by literature [44, 45, 46]. These questions allowed them to form logical arguments to solve the given SCC problem. Furthermore, they did not seem to take for granted any information included as part of a problem. Instead, they used every single piece of information to formulate their logical and systematic arguments [51]. In this regard, it was important for them to not only figure out the meaning of each piece of code, but also how all the individual parts were linked together.

Our expert programmers spent ample time to familiarise themselves with the task requirements by intensively reading (and even re-reading) the question. They clearly wanted to be sure that they fully comprehend the problem before attempting to solve it [47]. This ultimately enabled them to decide on the best strategies to use in tackling the given problem. Additionally, they also employed time-saving reasoning strategies such as ‘quick scan’ (reading through the code quickly just to get an overview) [53] and purposive elimination of identical code segments (see Section 4.1.3) in an attempt to reduce the time and effort they needed to solve a problem. However, their main focus was not to arrive at the answer as quickly as possible. Instead, they patiently followed appropriate and robust strategies, as suggested by literature [37, 55], to exhaust all possibilities to verify and ensure the correctness of their final answer (see Section 4.1.4). In most cases, this was achieved by double-checking the logic they followed to arrive at an answer.

5.2 Attention

In solving SCC problems, the experts who participated in this study were seen to be paying attention in the true sense. As a result, they were able to identify some aspects (e.g. similar lines of code, used data structures, etc.) that readily informed them on how to best tackle a given problem. In addition, these experts were observed to possess a skill that helped them to immediately switch to other strategies or alternatives [25] based on encountered information. This helped them to avoid getting stuck in certain areas of the source code or problem description. As a result, there were no signs of frustration, discomfort, and disorganisation [13] observed with our experts while solving the given SCC task.

Similar to the findings of Busjahn et al. [19], our experts paid more attention to complex code statements and functional details than to other simple or superficial details. By focusing their attention, they were able to temporarily ignore non-vital details at opportune moments (as suggested by Preece et al. [21]). Moreover, cases of switching from one cognitive process to another were observed in our experts. A typical example was when one expert (P2) switched from attention to decision making (see Section 4.2). Switching attention (e.g. strategy, alternative or cognitive process) in general seemed to have been helpful to our experts throughout their SCC process.

5.3 Reading

In reading the given code comprehension scenarios, our experts made sure that they understood what they were reading by re-reading the text of such scenarios, even if they thought they understood it. It was interesting that the experts who could be considered the most experienced, made sure of this and even emphasised the importance of doing this. As reading or re-reading is observed through eye or pen or hand movements [41, 42], our experts even mentioned that they were re-reading the details or some code fragments to confirm their initial understanding. They also interpreted and re-interpreted the various components of the scenario, implying that they did not automatically rely on their very first interpretations. This practice seems to have happened effortlessly with the experts, suggesting that their brains were already ‘wired’ or prepared for such practices.

5.4 Memory

From the observations made (see excerpts in Section 4.4), our experts seemed to possess innate knowledge that they applied if they did not note or write things down [71]. Otherwise, they wrote down everything they believed necessary or might be required during the SCC process. This helped them to easily update current variable values as and when necessary without straining their memory [2]. Another strategy our experts used was to consider limited scenarios at a time (e.g. comparing a worst- and best-case scenario). By minimising the amount of details kept in their memory, they did not necessarily have to write things down as they did not overload their working memory. Our experts also indicated that they used strategies such as think-aloud to help them remember certain information. Of particular interest was the expert (P4) who indicated that think-aloud was a fundamental technique he uses daily in almost everything he does (see Section 4.4).

5.5 Mental scaffolding techniques for the modelling of efficient SCC

As illustrated in the above discussions, we uncovered a number of cognitive processes and related strategies that were employed by our experts during the given SCC task. If these strategies are assumed as typical of the mental operations required to efficiently perform the discipline-specific task [8] of SCC, then these ways of thinking could point to crucial techniques that must be explicitly taught to students. The 17 identified techniques are presented in Table 1 with the related cognitive process(es) indicated for each. As indicated in Table 1, these techniques are all related to one or more of the reflective cognition processes (planning, cognitive reasoning and/or decision making) required for problem solving. Problem solving (as a vital element of efficient SCC) is one of the skills which are typically not taught to students explicitly, as instructors tend to concentrate more on core course content than on external aspects of the learning process [56]. Since so many CS students continue to struggle with SCC [1, 2], these are not techniques that should just be given to students to master on their own. Due to the complex cognitive nature of the SCC process [5, 6], we propose a more social context where these techniques (as presented in Table 1) are modelled to students as part of a

Table 1: Mental scaffolding techniques

Mental scaffolding techniques for the modelling of efficient SCC	Cognitive Processes						
	Planning	Reasoning	Decision Making	Reading	Speaking/ Listening	Attention	Memory
Read through the problem specification at least twice (or until you understand what you are asked to do).	✓			✓			
While reading through the specifications, mark/highlight important words.	✓			✓		✓	
Do a quick scan of the provided source code. Mark important sections AND complex sections of code.	✓			✓		✓	
For any complex sections of code, first make sure that you understand the meaning/working thereof BEFORE you continue to solve the problem.	✓			✓			✓
Identify any code segments that appear more than once (repeated code).	✓					✓	
Read through all the provided code at least twice to make sure that you understand everything.	✓			✓			
Write down any additional information that might be relevant in solving the problem.	✓					✓	✓
Identify at least two strategies you could follow to solve the problem.	✓					✓	
Compare the possible strategies and select ONE that you think will work best to solve the problem.		✓	✓				
Do not be afraid to adapt or change your strategy if it is not working.	✓	✓	✓				
For questions with multiple answer options, first focus on evaluating options that seem 'more correct' at first glance. Leave the 'possibly incorrect' options for later (if none of the 'more correct' options turn out to be the correct answer).		✓	✓			✓	
If you get stuck, consider the wider context in which the code or piece of information appears/is used.	✓	✓	✓			✓	
Define and use your own test case values (if not provided).		✓					✓
While tracing through the code, keep track of changes in variable values on paper.		✓					✓
Draw a diagram to visualise your understanding of the program logic.		✓					✓
Think aloud while working on solving the problem (if possible).		✓			✓		✓
Once you have arrived at an answer, double-check your reasoning to confirm the correctness of your answer.		✓	✓				

mental scaffolding process. By gradually removing the 'scaffolds', the learning process can move from full instructor-assisted modelling to a point where students have developed the necessary competence to perform techniques that were initially "beyond [their] unassisted efforts" [7, p. 90].

6 CONCLUSION AND FUTURE WORK

In focusing on Step 2 of the DtDs framework, this study utilised decoding interviews to identify four categories of cognitive processes (*attention; memory; reading, speaking and listening; and reflective cognition*) and related strategies that are essential for efficient SCC. By regarding these strategies as typical of the basic cognitive operations required for efficient SCC, we were able to identify 17 crucial SCC techniques. Although the techniques may seem quite simple (in the eyes of expert programmers and expert programming instructors), it could be indicative of SCC 'blind spots' – those crucial mental operations that instructors take for granted and fail to share explicitly with their students. By creating awareness regarding all the mental processes required for efficient SCC, we hope to, firstly, make other CS experts and instructors more aware of their own potential SCC 'blind spots'. Secondly, the uncovered SCC strategies highlight basic techniques that should be explicitly modelled to and practised by students in an attempt to help them overcome related SCC bottlenecks and better prepare them for SCC tasks. Thirdly, by using these techniques as part of a mental scaffolding process, instructors can gradually guide their students in mastering the basic mental operations needed for efficient SCC. The ultimate goal is to get students to perform tasks that were initially beyond their individual capacity [7], thereby mastering the core disciplinary skills of SCC.

Decoding interviews is just one of numerous methods that can be utilised in Step 2 of the DtDs process to systematically decode the disciplinary subconscious of experts [60]. All of the proposed methods are, however, aimed at revealing the explicit *steps* that an expert *would follow* when performing tasks related to an identified bottleneck [8]. In this study, the complex cognitive processes required to comprehend source code [6] have led us to instead focus (in the first part of the decoding interview) on exposing *specific strategies* that our experts *would follow* within these cognitive processes. The nature of the SCC process, however, allowed us to include an additional component in the second part of our decoding interviews. The addition of a think-aloud component provided us

with a unique opportunity to directly observe our experts' *actual execution* of the exposed cognitive strategies during a real SCC task. This provided us with even more insight into the nature of the cognitive processes and related strategies required for efficient SCC. Knowledge of these general cognitive strategies (as presented in Table 1) could therefore serve as a pre-step for revealing the explicit *steps* followed by experts in comprehending source code in future research.

7 REFERENCES

[1] R. Mccartney, J. Boustedt, A. Eckerdal, K. Sanders, and C. Zander, "Can first-year students program yet? A study revisited," in *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 2013, pp. 91–98. San Diego, California: ACM. doi: https://doi.org/10.1145/2493394.2493412

[2] B. Xie, G. L. Nelson, and A. J. Ko, "An explicit strategy to scaffold novice program tracing," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 344–349. Baltimore, Maryland: ACM. doi: https://doi.org/10.1145/3159450.3159527

[3] T. Busjahn and C. Schulte, "The use of code reading in teaching programming," in *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, 2013, pp. 3–11. Koli, Finland: ACM. doi: https://doi.org/10.1145/2526968.2526969

[4] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy," *SIGCSE Bull.*, vol. 38, no. 3, pp. 118–122, Jun. 2006. Bologna, Italy: ACM. doi: https://doi.org/10.1145/1140124.1140157

[5] P. A. Orlov, R. Bednarik, and L. Orlova, "Programmers' experiences with working in the restricted-view mode as indications of parafoveal processing differences," in *PPIG*, 2016, pp. 96–105.

[6] A. Praveen, "Program comprehension and analysis," *Int. J. Eng. Appl. Comput. Sci.*, vol. 01, no. 01, pp. 17–21, 2016. doi: https://doi.org/10.24032/ijeacs/0101/04

[7] D. Wood, J. S. Bruner, and G. Ross, "The role of tutoring in problem solving," *J. Child*

Psychol. Psychiatry, vol. 17, no. 2, pp. 89–100, 1976. doi: https://doi.org/10.1111/j.1469-7610.1976.tb00381.x

[8] J. K. Middendorf, D. Pace, and J. K. Middendorf, "Decoding the disciplines: A model for helping students learn disciplinary ways of thinking," *New Dir. Teach. Learn.*, vol. 2004, no. 98, pp. 1–12, 2004. doi: https://doi.org/10.1002/tl.142

[9] D. Pace, *The decoding the disciplines paradigm: Seven steps to increased student learning*. Bloomington: Indiana University Press, 2017.

[10] M. J. Nathan and A. Petrosino, "Expert blind spot among preservice teachers," *Am. Educ. Res. J.*, vol. 40, no. 4, pp. 905–928, 2003. doi: https://doi.org/10.3102/00028312040004905

[11] P. Felten, "Principles of good practice in SoTL," *Teach. Learn. Inq. ISSOTL J.*, vol. 1, no. 1, pp. 121–125, 2013. doi: https://doi.org/10.2979/teachlearninqu.1.1.121

[12] R. Lister *et al.*, "A multi-national study of reading and tracing skills in novice programmers," *ACM SIGCSE Bull.*, vol. 36, no. 4, pp. 119–150, 2004. doi: https://doi.org/10.1145/1041624.1041673

[13] P. J. Khomokhoana and L. Nel, "Decoding Source Code Comprehension: Bottlenecks Experienced by Senior Computer Science Students," in *ICT Education*, B. Tait, J. Kroeze, and S. Gruner, Eds. Cham: Springer, 2020, pp. 17–32. https://doi.org/10.1007/978-3-030-35629-3_2

[14] Cognifit, "Cognitive processes: What are they? Can they improve?," *Cognition and Cognitive Science*, 2019. [Online]. Available: https://www.cognifit.com/cognition. [Accessed: 04-Sep-2019].

[15] A. Newen, "What are cognitive processes? An example-based approach," *Synthese*, vol. 194, pp. 4251–4268, 2015. doi: https://doi.org/10.1007/s11229-015-0812-3

[16] K. R. Chandrika, J. Amudha, and S. D. Sudarsan, "Recognizing eye tracking traits for source code review," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1–8. Limassol, Cyprus: IEEE. doi: https://doi.org/10.1109/ETFA.2017.8247637

[17] S. Pero, C. Incoccia, B. Caracciolo, P. Zoccolotti, and R. Formisano, "Rehabilitation of attention in two patients with traumatic brain injury by means of 'attention

process training,'" *Brain Inj.*, vol. 20, no. 11, pp. 1207–1219, 2006. doi: https://doi.org/10.1080/02699050600983271

[18] T. D. Itoh *et al.*, "Towards generation of visual attention map for source code," in *Proceedings of the 11th annual conference organized by Asia-Pacific Signal and Information Processing Association (APSIPA)*, 2019. [Online]. Available: http://arxiv.org/abs/1907.06182 [Accesssed: 28-Jul-2019].

[19] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, 2011, pp. 1–9. Koli, Finland: ACM. doi:10.1145/2094131.2094133

[20] H. Uwano, M. Nakamura, A. Monden, and K. I. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, 2006, pp. 133–140. San Diego, California: ACM. doi: https://doi.org/10.1145/1117309.1117357

[21] J. Preece, Y. Rogers, and H. Sharp, *Interaction design: Beyond human-computer interaction*, 4th ed. New Delhi: John Wiley & Sons, Inc., 2015.

[22] M. W. Van Someren, Y. F. Barnard, and J. A. Sandberg, *The think aloud method: A practical guide to modelling cognitive processes*, 1st ed. London: Academic Press, 1994.

[23] F. Oyebode, *Sims' symptoms in the mind: Textbook of descriptive psychopathology*, 6th ed. New York: Elsevier Ltd., 2018.

[24] M. Dhingra and V. Dhingra, "Perception: Scriptures' perspective," *J. Hum. Values*, vol. 17, no. 1, pp. 63–72, 2011. doi: https://doi.org/10.1177/097168581001700104

[25] M. M. Sohlberg, "Psychotherapy approaches," in *Neuropsychological management of mild traumatic brain injury*, S. A. Raskin and C. A. Mateer, Eds. New York: Oxford University Press, 2000, pp. 137–156.

[26] P. A. Orlov and R. Bednarik, "The role of extrafoveal vision in source code comprehension," *Perception*, vol. 46, no. 5, pp. 541–565, 2017. doi:

https://doi.org/10.1177/0301006616675629

[27] W. Choi and P. C. Gordon, “Word skipping during sentence reading: Effects of lexicality on parafoveal processing,” *Atten. Percept. Psychophys.*, vol. 76, no. 1, pp. 201–213, 2013. doi: https://doi.org/10.3758/s13414-013-0494-1

[28] N. Pennington, “Comprehension strategies in programming,” in *Empirical studies of programmers: Second workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Norwood: Ablex Publishing Corporation, 1987, pp. 100–113.

[29] R. Brooks, “Towards a theory of the comprehension of computer programs,” *Int. J. Man. Mach. Stud.*, vol. 18, no. 6, pp. 543–554, 1983. doi: https://doi.org/10.1016/S0020-7373(83)80031-5

[30] S. Letovsky and E. Soloway, “Delocalized plans and program comprehension,” *IEEE Softw.*, vol. 3, no. 3, pp. 41–49, May 1986. doi: https://doi.org/10.1109/MS.1986.233414

[31] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, “Mental models and software maintenance,” *J. Syst. Softw.*, vol. 7, no. 4, pp. 341–355, 1987. doi: https://doi.org/10.1016/0164-1212(87)90033-1

[32] S. Letovsky, “Cognitive processes in program comprehension,” *J. Syst. Softw.*, vol. 7, no. 4, pp. 325–339, 1987. doi: https://doi.org/10.1016/0164-1212(87)90032-X

[33] A. Von Mayrhauser, A. M. Vans, A. Von Mayrhauser, and A. M. Vans, “Program understanding : Models and experiments,” *Adv. Comput.*, vol. 40, pp. 1–38, 1995. doi: https://doi.org/10.1016/S0065-2458(08)60543-4

[34] N. Gage and B. Baars, *Fundamentals of cognitive neuroscience: A beginner’s guide*. New York: Academic Press, 2018.

[35] G. A. Miller, “The magical number seven, plus or minus two: Some limits on our capacity for processing information,” *Psychol. Rev.*, vol. 101, no. 2, pp. 343–352, 1956. doi: https://doi.org/10.1037/h0043158

[36] E. F. Barkley, *Student engagement techniques : A handbook of college faculty*. California: Jossey-Bass, 2010.

[37] S. Fitzgerald, B. Simon, and L. Thomas, “Strategies that students use to trace code: An analysis based in grounded theory,” in *Proceedings of the First International*

Workshop on Computing Education Research, 2005, pp. 69–80. Seattle, Washington: ACM. doi: https://doi.org/10.1145/1089786.1089793

[38] D. Moore, K. Zabrucky, and N. E. Commander, "Validation of the metacomprehension scale," *Contemp. Educ. Psychol.*, vol. 22, pp. 457–471, 1997. doi: https://doi.org/10.1006/ceps.1997.0946

[39] N. Powell, D. Moore, J. Gray, J. Finlay, and J. Reaney, "Dyslexia and learning computer programming," *SIGCSE Bull.*, vol. 36, no. 3, p. 242, Jun. 2004. doi: https://doi.org/10.1145/1026487.1008072

[40] S. Scalabrino *et al.*, "Improving code readability models with textual features," in *Proceedings of the 24th IEEE International Conference on Program Comprehension*, 2016, pp. 1–10. Austin, Texas: IEEE. doi: https://doi.org/10.1109/ICPC.2016.7503707

[41] S. Bonaccio, J. O'Reilly, S. O'Sullivan, and F. Chiocchio, "Nonverbal behavior and communication in the workplace: A review and an agenda for research," *J. Manage.*, vol. 42, no. 5, pp. 1044–1074, 2016. doi: https://doi.org/10.1177/0149206315621146

[42] M. L. Knapp, J. A. Hall, and T. G. Horgan, *Nonverbal communication in human interaction*. Boston, Massachusetts: Cengage Learning, 2014.

[43] A. Colter and K. Summers, "Low literacy users," in *Eye Tracking in User Experience Design*, J. R. Bergstrom and A. J. Schall, Eds. Boston: Morgan Kaufmann, 2014, pp. 331–348. doi: https://doi.org/10.1016/B978-0-12-408138-3.00013-3

[44] F. Eisenführ, M. Weber, and T. Langer, *Rational decision making*. Berlin: Springer-Verlag Berlin Heidelberg, 2010.

[45] J. W. Herrmann, "Rational decision making," in *Wiley StatsRef: Statistics Reference Online*, N. Balakrishnan, T. Colton, B. Everitt, W. Piegorsch, F. Ruggeri, and J. L. Teugels, Eds. John Wiley & Sons Ltd., 2017, pp. 1–9. doi: https://doi.org/10.1002/9781118445112.stat07928

[46] F. C. Uzonwanne, "Rational model of decision making," in *Global Encyclopedia of Public Administration, Public Policy, and Governance*, A. Farazmand, Ed. Cham: Springer International Publishing AG, 2016, pp. 1–6. doi:

https:/doi.org/10.1007/978-3-319-31816-5_2474-1

[47] C. Atherden, *Can do problem solving year 1*, Revised. Cheltenham: Nelson Thornes, 2014.

[48] B. Hayes-Roth and F. Hayes-Roth, "A cognitive model of planning," *Cogn. Sci.*, vol. 3, no. 4, pp. 275–310, 1979. doi: https://doi.org/10.1207/s15516709cog0304_1

[49] M. G. Guerrero and R. Puche-Navarro, "The emergence of cognitive short-term planning: Performance of preschoolers in a problem-solving task," *Acta Colomb. Psicol.*, vol. 18, no. 2, pp. 13–27, 2015. doi: https://doi.org/10.14718/ACP.2015.18.2.2

[50] J. S. B. T. Evans, "The cognitive psychology of reasoning: An introduction," *Q. J. Exp. Psychol. Sect. A*, vol. 46, no. 4, pp. 561–567, 1993. doi: https://doi.org/10.1080/14640749308401027

[51] M. Knauff, "How our brains reason logically," *Topoi*, vol. 26, no.1, pp. 19–36, 2007. doi: https://doi.org/10.1007/s11245-006-9002-8

[52] P. Gabor, "Management theory and rational decision making," *Manag. Decis.*, vol. 14, no. 5, pp. 274–281, 1976.

[53] T. Bauer, B. Erdogan, J. Short, and M. Carpenter, *Principles of management*. New York: Flat World Knowledge, 2016.

[54] M. Jones, "The redesign of the delivery of an introductory programming unit," *Innov. Teach. Learn. Inf. Comput. Sci.*, vol. 6, no. 4, pp. 169–182, 2007. doi: https://doi.org/10.11120/ital.2007.06040169

[55] S. Frederick, "Cognitive reflection and decision making," *J. Econ. Perspect.*, vol. 19, no. 4, pp. 25–42, 2005. doi: https://doi.org/10.1257/089533005775196732

[56] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 1449–1461. doi: https://doi.org/10.1145/2858036.2858252

[57] D. Plowright, *Using mixed methods: Frameworks for an integrated methodology*. London: Sage Publications, 2011.

[58] D. Cooper and P. Schindler, *Business research methods*, 12th ed. New York: McGraw-Hill Education, 2013.

[59] M. Q. Patton, *Qualitative research & evaluation methods: Integrating theory and practice*, 4th ed. Thousand Oaks: Sage, 2015.

[60] J. Middendorf and L. Shopkow, *Overcoming student learning bottlenecks: Decode your disciplinary critical thinking*. Sterling, Virginia: Stylus Publishing, LLC., 2018.

[61] L. Shopkow, A. Diaz, J. Middendorf, and D. Pace, "From bottlenecks to epistemology in History: Changing the conversation about the teaching of History in colleges and universities," in *Changing the Conversation about Higher Education*, R. Thompson, Ed. New York: Rowman & Littlefield Publishers, 2013, pp. 15–38.

[62] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, 5th ed. Thousand Oaks: Sage, 2017.

[63] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, "Data cleaning: Overview and emerging challenges," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 2201–2206. doi: https://doi.org/10.1145/2882903.2912574

[64] E. S. Parcell and K. A. Rafferty, "Interviews, recording and transcribing," in *The SAGE Encyclopedia of Communication Research Methods*, M. Allen, Ed. Thousand Oaks: Sage Publications, Inc., 2017. doi: http://dx.doi.org/10.4135/9781483381411.n275

[65] P. Liamputtong, "Qualitative data analysis: conceptual and practical considerations," *Heal. Promot. J. Aust.*, vol. 20, no. 2, pp. 133–139, 2009. doi: https://doi.org/10.1071/HE09133

[66] J. Saldaña, *The coding manual for qualitative researchers*. London: Sage, 2016.

[67] V. Braun and V. Clarke, "Using thematic analysis in Psychology," *Qual. Res. Psychol.*, vol. 3, no. 2, pp. 77–101, 2006. doi: https://doi.org/10.1191/1478088706qp063oa

[68] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "The roles beacons play in comprehension for novice and expert programmers," in *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group, 2002,* pp. 58–73.

[69] T. De Jong and T. Jong, "Cognitive load theory, educational research, and

instructional design: Some food for thought," *Instr. Sci.*, vol. 38, pp. 105–134, 2010. doi: https://doi.org/10.1007/s11251-009-9110-0

[70] S. Wiedenbeck, "Novice/expert differences in programming skills," *Int. J. Man. Mach. Stud.*, vol. 23, no. 4, pp. 383–390, 1985. doi: https://doi.org/10.1016/S0020-7373(85)80041-9

[71] J. Bransford, A. Brown, and R. Cocking, *How people learn: Brain, mind, experience, and school*, Expanded. Washington, DC: National Academy Press, 2000.

Chapter 6 – (Article 3)

Narrowing the gap between expert and novice thinking: A step-by-step framework for efficient source code comprehension[3]

Abstract. The Decoding the Disciplines (DtDs) philosophy is based on the premise that each discipline (including Computer Science) has its own unique set of mental operations. In many cases, these operations have become invisible to instructors as they tend to perform them automatically based on years of experience. If the nature of these operations is not made explicit to students, it is likely that they will develop learning 'bottlenecks' which could prevent them from mastering key disciplinary practices. One of the key bottlenecks identified in the CS discipline relates to students' inability to reliably work their way through the long chain of reasoning necessary to comprehend source code. In an attempt to narrow the existing gap between expert and novice thinking in this regard, the study utilised decoding interviews with five expert programmers (who were also experienced instructors) to systematically deconstruct the explicit mental techniques and reasoning strategies necessary for efficient source code comprehension (SCC). Thematic analysis of the mental operations performed by these experts during an SCC activity, led to the identification of 11 key strategies. Knowledge of these strategies as well as the explicit mental operations were then used to devise a step-by-step framework for efficient SCC. The purpose of this framework is to create awareness among instructors regarding the explicit mental operations required for efficient SCC and to serve as source of further research and refinement. Moreover, within the realm of the DtDs philosophy, 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.

---

[3] Publishable manuscript.

Keywords: Source code comprehension, expert programmers, decoding the disciplines, decoding interview, computer science education.

1. Introduction

Source code comprehension (SCC) is a term that is generally used to refer to the reading, interpreting, and understanding of existing source code (Lister et al., 2004; Lister, Simon, Thompson, Whalley & Prasad, 2006; Maalej, Tiarks, Roehm & Koschke, 2014). Despite SCC being a critical disciplinary skill (Siegmund et al., 2012; Tiarks, 2011), 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 (Khomokhoana & Nel, 2020; Lister et al., 2004). This student-learning bottleneck can be attributed to students' fragile knowledge of basic programming concepts (de Raadt, 2007; Perkins & Martin, 1986). A bottleneck refers to specific places where the learning of a significant number of students gets interrupted (Diaz, Middendorf, Pace & Shopkow, 2008; Middendorf & Pace, 2004). This interruption happens when students are not sure how to approach a given problem, and as a result apply improper strategies (Pace, 2017).

The severity of a bottleneck can be further intensified when instructors are unable to accurately portray disciplinary ways of thinking to students (Middendorf & Pace, 2004). This may occur 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 (Nathan & Petrosino, 2003). Decoding the Disciplines (DtDs) is a process that focuses on increasing student learning "by bridging (sic) the gap between novice and expert thinking" (Middendorf & Shopkow, 2018, p. 12). The seven-step DtDs framework can be used to firstly expose the nature of such 'hidden' operations (linked to a specific bottleneck) and to create awareness among instructors regarding the steps or operations they typically omit when teaching their students. In subsequent steps of this framework, instructors are guided to devise ways of helping students master these operations and hence overcome specific learning bottlenecks (Middendorf & Pace, 2004; Pace, 2017).

Bottlenecks are typically identified in Step 1 of the DtDs framework, while Step 2 focuses on exploring the detailed or explicit mental steps that experts in a given field would go through to accomplish the task(s) identified as a bottleneck (Middendorf &

Pace, 2004; Pace, 2017). In executing Step 2, several artefacts and/or techniques (such as decoding interviews, rubrics, metaphors/analogies, mind maps, reflective writing, and non-verbal modelling) can be used to uncover the explicit steps followed by experts (Middendorf & Pace, 2004; Middendorf & Shopkow, 2018). However, decoding interviews are cited as the most rigorous and effective technique in this regard (Pace, 2017). A decoding interview is a special type of interview where disciplinary experts (typically experienced instructors) are intellectually guided to reveal the explicit steps they follow in order to get through a predetermined learning bottleneck (Middendorf & Baer, 2019; Middendorf & Pace, 2004; Pace, 2017). As a first step in dealing with a previously defined discipline-specific learning bottleneck (novice programmers' inability to reliably think and work their way through a long chain of reasoning to efficiently comprehend a piece of source code), the study described in this paper focuses on Step 2 of the DtDs framework. Consequently, this paper attempts to answer the following two questions:

- What are the explicit mental strategies (techniques and reasoning) 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 narrowing the gap between expert and novice thinking with regard to efficient SCC?

The remainder of this paper is structured as follows: Section 2 presents an overview of previously identified SCC strategies that are relevant to the specific learning bottleneck under investigation. In the discussion of the research design and method in Section 3, detail is provided regarding the selection of the decoding-interview participants and interview panel. Special attention is also given to the validation steps that were included to enhance the validity of the identified SCC strategies and steps. In presenting the findings (Section 4), care is taken to link the identified strategies as closely as possible with existing knowledge regarding SCC strategies. Based on the findings, Section 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 Section 6.

2. 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 (Littman, Pinto, Letovsky & Soloway, 1987; Pennington, 1987; Von Mayrhauser & Vans, 1995). Novice programmers and expert programmers have been shown to favour different strategies, with top-down mostly observed with novices (Mosemann & Wiedenbeck, 2001). 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 or strategies required for efficient SCC. In previous studies, researchers used observations (Feigenspan, Siegmund & Fruth, 2011; Dunsmore & Roper, 2000; Siegmund, Kástner, Apel, Brechmann & Saake, 2013) and think-alouds (Anderson, Bachman, Perkins & Cohen, 1991) to gather more detailed information regarding the specific strategies employed during SCC.

In a study that evaluated the ways in which students (as novice programmers) answered code-based multiple-choice questions, Fitzgerald, Simon and Thomas (2005) identified 19 strategies used by students (see Table 1). Although this list includes a number of good SCC strategies, as also identified by other researchers (Cunningham, Blanchard, Ericson & Guzdial, 2017; Lister et al., 2004; Moore, Zabrucky & Commander, 1997; Whalley, Prasad & Kumar, 2007), the novices did not always execute these strategies in an optimum way. However, the strategies identified by Fitzgerald et al. (2005) are much more specific than the generic SCC strategies alluded to above. Although students and experts do not necessarily follow the same SCC strategies, these strategies could serve as a starting point for identifying more explicit details regarding the exact mental operations required for effective SCC.

In Fitzgerald et al.'s (2005) study, the novices used the first four strategies (see S1, S2, S3, and S4 in Table 1) to acquaint themselves with some elements of the code-based questions they were answering. This type of *self-orientation* is a typical strategy used by people to familiarise themselves with the elements of the problem to be tackled or question to be answered (Illeris, 2003). Simon, Lopez, Sutton and Clear (2009) emphasise the importance of reading programs or pieces of code in order to comprehend it. In this regard, Moore et al. (1997) even suggest reading through question specifications or a piece of source code twice (Moore, Zabrucky & Commander, 1997). To further enhance comprehension of a task, strategies such as

highlighting, underlining and colouring some words or text can also be used (Powell, Moore, Gray, Finlay & Reaney, 2004; Sarkar, 2015). These important words or pieces of text could be regarded as key for a programmer's comprehension of either the problem description or the code in question. However, *use of keywords* is one SCC strategy that has not been observed with Fitzgerald et al.'s (2005) novices.

Table 1: Strategies employed by novice programmers in answering code-based MCQs

No.	Strategy	Explanation
S1	Reading the question	Previewing the question and looking for what is asked.
S2	Previewing the code by identifying data structures	Identifying where variables/constants/objects are first encountered (i.e. declared) in a program.
S3	Previewing the code by identifying the initialisation of data structures	Identifying where variables/constants/objects are assigned initial values in a program.
S4	Previewing the code by identifying control structures	Identifying branching or decision-making constructs (e.g. loops) which control the execution flow of the program.
S5	Understanding new concepts (semantic)	Understanding something new by relating prior understood knowledge to less understood knowledge or by using an example.
S6	Pattern recognition: Temporally self-referential	Syntactically recognising that this looks like something from another question in this test; multiple-choice distractors.
S7	Pattern recognition: Outside knowledge	Syntactically recognising something familiar from outside knowledge.
S8	Pattern recognition: Seeking higher levels of meaning from the code	What the code really does at a higher semantic level.
S9	Walkthroughs	Tracing, testing boundary or error conditions.
S10	Strategising	Asking questions like 'How would I write the code?' or 'What would I need to do?'
S11	Grouping	Looking for similarities and differences in the answers and selecting more than one answer at once (for possible elimination or inclusion).
S12	Differentiation	Noticing differences between answers or lines of code or choices.
S13	Elimination	Ruling out specific choices.
S14	Guessing	Guessing an answer from the provided options.
S15	Thoroughness	After selecting an answer, verifying the correctness thereof. Could also include verification of the incorrectness of other answer options.
S16	Starting over	Getting lost or recognising an error and simply starting again.
S17	Coming back to the question later	Going on to another part of the test without completing this problem.
S18	Posing questions	Asking explicit questions regarding specific pieces of code that could impact on the process leading to the correct answer (e.g. 'Is this the end of a loop?').
S19	Doodling	Making drawings, sketches or annotations on a piece of paper.

[*Source: Adapted from* (Fitzgerald et al., 2005, p. 73)]

Littman, Pinto, Letovsky and Soloway (1987) 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 question to be answered before trying to understand the minute details of the task. This understanding already starts to develop in the early 'previewing' SCC strategies (e.g. S2 – *previewing the code by identifying data structures*) suggested by Fitzgerald et al. (2005). Comprehension can also be enhanced through *pattern recognition* where similar or related code elements are classified into categories (Kpalma & Ronsin, 2007). In answering code-based questions, patterns can be identified in terms of both syntax and semantics (Fitzgerald et al., 2005).

Similar to solving many real-world problems, the solving of SCC problems also requires the application of logical reasoning processes (Butler & Morgan, 2007). In SCC problems, test cases can, for example, be used to check the logic of both basic and advanced conditions (Srikant & Aggarwal, 2014). To evaluate these test cases, a programmer will typically conduct a *walkthrough* of the given source code (see S9 in Table 1). Other ways in which programmers can apply logical reasoning, is by *strategising* (S10) about how they would solve the given problem if they had to write the code from scratch or by asking themselves specific questions (e.g. S18 – *Posing questions*) that could help with their comprehension of a given piece of code. Examples of such questions include: What should I do? What alternative courses of action do I have available? Which alternative courses of action should I select to use? Why should I use these alternative courses of action? What are the consequences of using these alternatives? (Eisenführ, Weber & Langer, 2010; Herrmann, 2017; Uzonwanne, 2016).

Strategies such as *grouping* (S11), *differentiation* (S12), *elimination* (S13), and *guessing* (S14) are typically associated with the answering of MCQs, because there are multiple answer options to choose from (Complete Test Preparation Inc., 2014). On the other hand, the *thoroughness* strategy (S15), where an answer is re-checked to confirm the correctness thereof, can be applied to any type of question. Frederick (2005, 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 the correctness thereof.

Given the relative simplicity of the code-based questions used in Fitzgerald et al.'s (2005) study, the *starting over* (S16) and *coming back to the question later* (S17) strategies could be regarded as typical novice strategies that one would not necessarily expect to observe with expert programmers (unless the complexity of the questions was really high).

The *doodling* strategy (S19) was identified by Lister et al. (2004) as a strategy typically used by programming experts to better comprehend source code. In various other studies, it has been found that students who make annotations or sketches during SCC were more successful than those who did not (Cunningham et al., 2017; Fitzgerald et al., 2005). However, in Xie et al.'s (2018) study, some of the participants specifically mentioned that they did not use doodles because they found it time-consuming and unnecessary.

From this discussion, it is evident that although there are various SCC strategies that have previously been identified, some of these strategies seem to be more novice-specific. The format in which the SCC question is presented could also impact the choice of comprehension strategies used by the programmer. As such, it is possible that not all of the strategies discussed here would necessarily result in efficient SCC. The next section presents the research design and procedure that was followed to identify strategies that expert programmers follow while comprehending source code.

3. Research Design and Method

A narrative research approach based on Plowright's (2011) Frameworks for an Integrated Methodology (FraIM) was adhered to in this study. The data source management strategy was a case study. Data was mainly collected through face-to-face decoding interviews (Middendorf & Pace, 2004) (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 (Cooper & Schindler, 2013) based on their experience in teaching programming courses. This sample can also be regarded as convenient (Patton, 2015), since the selected participants were in close proximity 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 typically regarded as intellectually very demanding (Middendorf & Pace, 2004), where a single interviewer can easily get lost in the details. Pace (2017) therefore recommends that this type of interview be conducted by at least two interviewers, as two minds will be in a better position to control the interview process. Both interviewers should be able to verbalise their thinking, challenge interviewees' explanations, and summarise the interviewees' thinking back to them at an abstract level (Shopkow, Middendorf & Pace, 2013). As members of the same discipline are more likely to share common expert blind spots, Pace (2017) also suggests using a second interviewer from outside the discipline in question. Such an interviewer will be more likely to notice when mental steps are not well explained. Since we were unable to 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 (the first author) acted as the main interviewer, while the second interviewer (as described above) played a supporting role. The proceedings of each interview were audio recorded with 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.

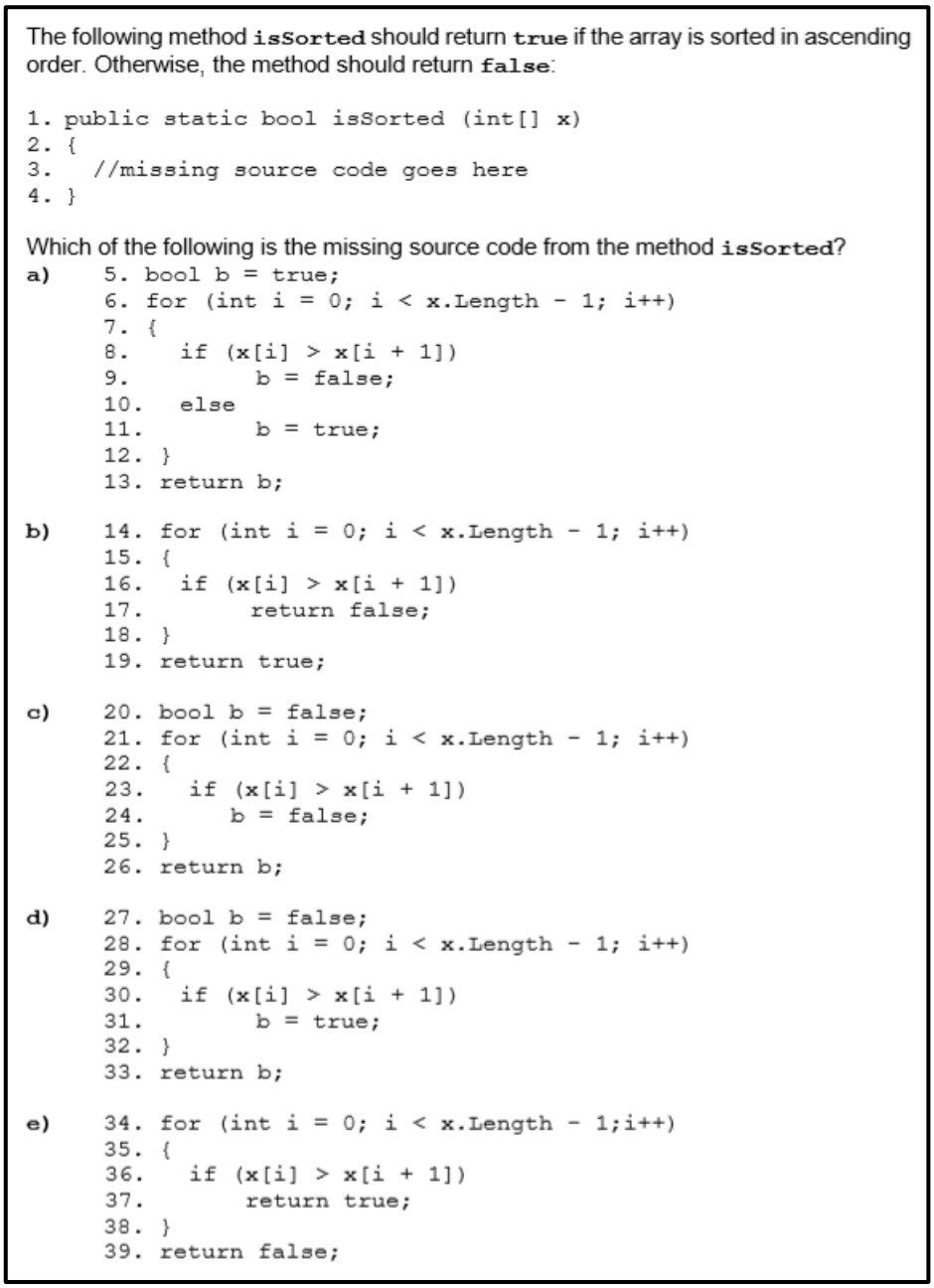
In order to uncover more explicit details regarding the nature of the shared strategies, it was necessary to present the participants with a real SCC task. In the second part of the interview, each participant was therefore presented with a real SCC problem and asked to illustrate how they would implement 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 (2004) study. The selected question (Question 6 – see Figure 1) was identified as the second most challenging question in the Lister et al. (2004) study. This question was particularly selected as it covers a wider range of concepts (e.g. Boolean variables, `for` loops, array indices, and use of `return` 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 Figure 1 were only added in aid of the discussion that follows in Section 4.

3.2. Data analysis

An adapted version of Creswell and Creswell's (2017) 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 (Parcell & Rafferty, 2017) 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 (Lincoln & Guba, 1985). Inherently, it is well-accepted that in dealing with narrative inquiries, the researcher is regarded as the instrument (Patton, 2015). As such, we had to immerse ourselves in the data to be fully familiar with its breadth and depth (Braun & Clarke, 2006). 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 were able to decide on a coding plan that would help with the analysis of the data in order to address the research questions. The five validated transcripts were imported into NVivo 12 and codes were created based on the strategies identified in the data. Subsequently, words and/or

short phrases (Saldaña, 2016) 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 (Saldaña, 2016), 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 (Lewins & Silver, 2007).



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;
```

Figure 1: Code tracing question used in decoding interview

3.3. Validation

After completion of the data analysis, an initial step-by-step framework for efficient SCC was compiled. In order 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 (Schwandt, Lincoln & Guba, 2007; Guba, 1981) of this study's research 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 Section 3.2). The purpose of this meeting was to further check for possible ambiguities in the proposed steps/strategies. In order 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. Based on the feedback received during this validation meeting, a few minor changes were made to the initial framework. The resulting, final framework is presented in Section 5.

4. Findings and Interpretation

Based on the analysis of the decoding-interview transcripts, 11 key SCC strategies (techniques or reasoning) were observed with our experts. In the following sub-sections, each of these strategies is presented together with evidence of its occurrence.

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. (2005). 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 source code and provided reasons for his actions. Linking to the importance of reading the question description and code statement (Fitzgerald et al., 2005; Simon et al., 2009), 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, 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 – (Moore, Zabrucky & Commander, 1997)] to ensure that he attained the correct understanding. Expert programmers tend to read the code at least twice (even if they understood it at first) just to confirm their original understanding (Simon, Lopez, Sutton & Clear 2009). 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 from P4 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 (O'Brien & Buckley, 2001).

P4 further pointed out that the self-orientation process may not necessarily be a smooth one, as there might be instances where the process would have to go a bit slower (e.g. if he is being challenged to understand some components of the code or problem). As sequential code reading is common with novice programmers (Letovsky,

1987), it is important to note that P4 would only resort to sequential code reading if there were compelling circumstances. Otherwise, he would apply high-level strategies such as global reasoning and conceptual understanding, as suggested by Fitzgerald et al. (2005).

4.2. Keyword identification

A keyword is a word or 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. (2004), there are different ways that 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 obviously a lot easier to identify the keywords."* This suggests that while using an integrated development environment (IDE) such as Visual Studio that provides this colour functionality, P2 would take the advantage of the built-in syntax highlighting (Sarkar, 2015) 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 Figure 1), words that referred to specific coding concepts were formatted in a different type face – 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.

4.3. Data structure identification

The identification of data structures entails locating the 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 identified by Fitzgerald et al. (2005). If a data structure is unfamiliar to a programmer, they could resort to using the *understanding new concept (semantic) strategy*, thereby suggesting that S2, S3, and S5 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 (Goldman et al., 2008; Jimoyiannis, 2011; Litvinov et al., 2017; Ma, Ferguson, Roper & Wood, 2011; Izu, Weerasinghe & Pope, 2016). 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 exact same steps she would personally follow while teaching her students. This action could have a negative impact on the SCC understanding of her students (Ambrose, Bridges, DiPietro, Lovett & Norman, 2010).

4.4. 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 (2005) *understanding new concepts (semantic)* (S5) strategy. In order to deduce meaning from the problem context, expert programmers follow various courses of action. Three of the participants in this study (by means of four occurrences) shared what they would do if they came across difficult or 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. (2005) and observed with P4, experts do not have a problem to proceed 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.

4.5. 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 (Grundy, 2014). Activities performed as part of the *strategising* (S10) and *posing questions* (S18) strategies, as suggested by Fitzgerald et al. (2005), require high-level reasoning (Lister et al., 2004). 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 (Butler & Morgan, 2007). 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` *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 if the conditions under which such a repetition structure would operate are not met.

4.6. Walkthroughs

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" (Jeffries, 1982, p. 12). 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 (2013). 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 in another trace table. If I have the code on a piece of paper, I will actually 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 actually 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 (Chou & Sun, 2013) and also modelling what was happening in her mind (Hertz & Jump, 2013).

With regard to 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 (2014), 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 for '*testing boundary or error conditions*' as specified by Fitzgerald et al. (2005).

4.7. 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 complete understanding of concepts and to integrate the various aspects contained in the question to be answered and/or 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 (Miller, 1956), to say there is no need to revisit the previous stages if the information needed can be recalled from the working memory.

4.8. Doodling

Doodles or annotations refer to the making of some drawings, sketches or writings (i.e. of some variable values) while solving a problem. These 'ideas' can then easily be referenced when deemed necessary (Lister et al., 2004). The use of doodles has also been confirmed as a valuable strategy in correctly solving SCC problems (Cunningham et al., 2017; Fitzgerald et al., 2005; Lister et al., 2004). Interpreting doodles can also help to reveal the programmer's thinking patterns while solving a problem (Hertz & Jump, 2013). In this study, the use of annotations or doodles was only observed in one participant (P2). In addition to the trace table she drew (see Section 4.6), there was another observed occurrence of doodling where she just wrote 'worst', 'best' and, 'avg' and circled them. This indicated that she was thinking of the three test cases she could use to confirm her answer. Due to an oversight by the interview panel, these artefacts were not retained for further analysis.

4.9. Thoroughness

The thoroughness strategy is characterised by the careful examination of the attained solution to a problem in order to confirm the correctness of the solution (or answer) (Fitzgerald et al., 2005). Three occurrences of this strategy were observed with P2. While trying to find the correct answer, she tentatively identified option B as the correct answer: "*So, I will mark B as a possible solution. Because at the moment, without going really in depth and using a test case, it looks to me like it will work*." What is of importance here is that she did not stop there. Instead, she continued and said: "*So I will just go to B and I will check it again.*" In this way, she resisted presenting the answer she first arrived at as her final answer (Frederick, 2005).

4.10. Pattern recognition

During pattern recognition, similar or related code elements are identified and consequently thought of and treated collectively (Kpalma & Ronsi, 2007). Two occurrences of pattern recognition were observed in P5. While studying answer option A (see Figure 1), he was observed thoroughly checking Line 8, but when he got to Line 16, he just hovered his pen over that line. As a result of this observation, the second interviewer posed the following question to him: "*Did you use the pattern from option A and apply it to option B?*" In response, P5 said: "*Yes, I did not have to check whether this condition makes sense again.*" During SCC, both syntactical and

semantical patterns can be valuable in helping programmers make decisions that could help them arrive at the desired comprehension or answer to a code-related question. Ideas about these patterns could come from the programmers' prior knowledge or their previous interactions with code syntax and semantics (Fitzgerald et al., 2005).

4.11. Group answer options

The *grouping* (S11), *differentiation* (S12) and *elimination* (S13) strategies are often used together during SCC (Fitzgerald et al., 2005). In following these strategies, programmers ultimately aim to either eliminate answer options or identify answer options to focus on first. Instances of all three these strategies were identified among the expert programmers in this study. Given the close relation between these strategies, they are categorised under one main strategy – *group answer options*.

4.11.1. Grouping

When applying a grouping strategy during SCC, a programmer specifically tries to identify similarities and differences in the provided answer options in order to form answer groups. This is done with the objective of either including or excluding entire groups of answer options as possible answers (Fitzgerald et al., 2005). Two occurrences of grouping were observed in this study. While solving the given SCC problem, P1 was observed checking lines 5, 14, 20, 27 and 34 (see Figure 1) and was consequently asked what he was doing. He explained that in answer options A, C, and D, the boolean variable `b` was declared before the `for` loop, while it was not declared before the `for` loop in options B and E. In doing so, he was trying to establish which group of answer options he should focus on first (as a time-saving strategy). Although the actions of P1 could also be regarded as pattern recognition (see Section 4.10), the reason he provided for the grouping led to this action being classified as an example of using a 'grouping' strategy. This also corresponds with Fitzgerald et al.'s (2005) explanation of the grouping strategy.

4.11.2. Differentiation

When applying the differentiation strategy, a programmer explicitly looks for the differences between the given alternatives or lines of code (Fitzgerald et al., 2005). There were 15 occurrences of this strategy identified in all the participants. P1

employed the strategy the most, with seven occurrences. He specifically said: "*What I am also trying to do is to find out what the differences are in the five options.*" Understanding these differences helped him to decide which alternative answers to pursue further (i.e. possibly correct) and which ones to discard (i.e. possibly incorrect) immediately.

4.11.3. Elimination

In the elimination strategy, a person uses some criterion to judge specific alternatives as undesirable (Fitzgerald et al., 2005). A total of 16 occurrences of this strategy were observed in all the participants. P1 employed this strategy the most, with seven occurrences. Using the term 'discard' for elimination, P5 said: "*But I can immediately discard this option [Option A], being inefficient … let us see if I can just take a global approach and see one of them that I can immediately discard.*" As can be seen from the excerpt, P5 wanted to group efficient and inefficient alternative answers so that he could start by eliminating the inefficient ones first.

5. Framework for Efficient Source Code Comprehension

By using the strategies identified in Section 4, as well as insights gained from observing the explicit SCC strategies followed by the expert participants during the decoding interviews, a step-by-step framework for efficient SCC (see Table 2) was formulated. This framework contains 10 key steps linked to each of the relevant mental strategies (techniques and reasoning strategies) used by the experts in executing each of these steps (as discussed in Section 4). However, within some of the main steps, there are several sub-steps that 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 2: Proposed step-by step framework for efficient SCC

	Step-by-step framework for answering a source code comprehension question	
Strategies Applied	**Steps**	**Description**
• Self-orientation • Keyword identification • Strategic thinking • Revisit previous stages • Doodling	**1**	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**	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 • Doodling • Pattern recognition	**3**	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**	If there is code syntax that you do not understand, consult any additional resource(s) for clarification.
• Deduction of meaning from context	**5**	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 • Doodling	**6**	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).

	Step-by-step framework for answering a source code comprehension question (continued)	
Strategies Applied	**Steps**	**Description**
• Strategic thinking • Group answer options • Pattern recognition	**7**	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 • Doodling	**8**	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**	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**	Repeat **Step 8** to confirm the correctness of your final answer.
		• Use your own test case values (if not provided).

6. Conclusion

The gap that exists between the ways in which novice and expert programmers comprehend source code continues to be a challenge. Better understanding of those mental operations that have become invisible to instructors (because they perform it automatically), could be valuable in narrowing this gap. This can be achieved by uncovering the explicit nature of the mental techniques and reasoning 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 11 key strategies that expert programmers typically employ during SCC. What also became apparent, is that experts each approach SCC differently. At any stage during the SCC process, the experts' prior knowledge or experience can trigger them to use specific strategies. The experts also find it easy to switch to a completely different strategy based on what they are currently thinking, as well as information or details that they are encountering.

The SCC techniques and reasoning strategies identified in this study, in combination with existing knowledge (from literature and based on the authors' own experience), were used to develop a step-by-step framework for efficient SCC. The main purpose of this framework is to create awareness among instructors regarding the explicit mental operations required for efficient SCC. Knowledge of the nature of these mental operations could firstly help instructors to better understand their own expert blind spots. Moreover, as a practical contribution within the realm of the DtDs philosophy (Middendorf & Pace, 2004), 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 proposed 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 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 – as a scientific way of thinking and learning that is gaining popularity worldwide – can be used in the investigation of classroom practices as suggested by Middendorf and Pace (2004). Consequently, such a research approach should hold particular appeal for instructors working in the Science, Technology, Engineering and Mathematics (STEM) education fields.

7. References

Ambrose, A. S., Bridges, W. M., DiPietro, M., Lovett, C. M., & Norman, K. M. (2010). *How learning works : Seven research-based principles for smart teaching*. California: John Wiley & Sons. https://doi.org/10.1002/mop.21454

Anderson, N. J., Bachman, L., Perkins, K., & Cohen, A. (1991). An exploratory study into the construct validity of a reading comprehension test: Triangulation of data sources. *Language Testing - LANG TEST*, *8*(1), 41–66. https://doi.org/10.1177/026553229100800104

Braun, V., & Clarke, V. (2006). Using thematic analysis in Psychology. *Qualitative Research in Psychology*, *3*(2), 77–101. https://doi.org/10.1191/1478088706qp063oa

Butler, M., & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In R. Atkinson, C. McBeath, A. S. Swee Kit, & C. Cheers (Eds.), *Proceedings of ascilite Singapore 2007 ICT: Providing Choices for Learners and Learning* (pp. 99–107). Nanyang Singapore: Nanyang Technological University.

Chou, C. Y., & Sun, P. F. (2013). An educational tool for visualizing students' program tracing processes. *Computer Applications in Engineering Education*, *21*(3), 432–438. https://doi.org/10.1002/cae.20488

Complete Test Preparation Inc. (2014). *GED Test Strategy! Winning multiple choice strategies for the GED: Winning Multiple Choice Strategies for the GED Exam*. Victoria BC: Complete Test Preparation Inc.

Cooper, D., & Schindler, P. (2013). *Business research methods* (12th ed.). New York: McGraw-Hill Education.

Creswell, J. W., & Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches* (5th ed.). Thousand Oaks: Sage.

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using tracing and sketching to solve programming problems. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 164–172). https://doi.org/10.1145/3105726.3106190

de Raadt, M. (2007). A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education*, *17*(3), 201–213. https://doi.org/10.1080/08993400701538104

Diaz, A., Middendorf, J., Pace, D., & Shopkow, L. (2008). The history learning roject: A department "decodes" its students. *Journal of American History*, *94*(4), 1211–

1224. https://doi.org/10.2307/25095328
Dunsmore, A., & Roper, M. (2000). A comparative evaluation of program comprehension measures. *The Journal of Systems and Software*, *52*(3), 121–129.
Eisenführ, F., Weber, M., & Langer, T. (2010). *Rational decision making*. Berlin: Springer-Verlag Berlin Heidelberg.
Feigenspan, J., Siegmund, N., & Fruth, J. (2011). On the role of program comprehension in embedded systems. *Softwaretechnik-Trends*, *31*(2).
Fitzgerald, S., Simon, B., & Thomas, L. (2005). 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). New York: Association for Computing Machinery. https://doi.org/10.1145/1089786.1089793
Frederick, S. (2005). Cognitive reflection and decision making. *Journal of Economic Perspectives*, *19*(4), 25–42. https://doi.org/10.1257/089533005775196732
Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 256–260). Portland, Oregon: Association for Computing Machinery. https://doi.org/10.1145/1352135.1352226
Grundy, T. (2014). *Demystifying strategic thinking: lessons from leading CEOs*. New Delhi: Kogan Page Publishers.
Guba, E. G. (1981). Criteria for assessing the trustworthiness of naturalistic inquiries. *Educational Communication and Technology*, *29*(2), 75–91.
Herrmann, J. W. (2017). Rational decision making. In N. Balakrishnan, T. Colton, B. Everitt, W. Piegorsch, F. Ruggeri, & J. L. Teugels (Eds.), *Wiley StatsRef: Statistics Reference Online* (pp. 1–9). John Wiley & Sons Ltd. https://doi.org/10.1002/9781118445112.stat07928
Hertz, M., & Jump, M. (2013). Trace-based teaching in early programming courses. In *Proceedings of the 44th ACM technical symposium on Computer science education* (pp. 561–566). Denver, Colorado: ACM. https://doi.org/10.1145/2445196.2445364
Illeris, K. (2003). Learning, identity and self-orientation in youth. *Nordic Journal of Youth Research*, *11*(4), 357–373. https://doi.org/10.4324/9781315620565-6
Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of AERA annual meeting* (pp. 1–17). New York: American Educational Research Association.
Jimoyiannis, A. (2011). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education*, *4*(2), 53–74.
Khomokhoana, P. J., & Nel, L. (2020). Decoding source code comprehension: Bottlenecks experienced by senior computer science students. In B. Tait, J. Kroeze, & S. Gruner (Eds.), *ICT Education* (pp. 17–32). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-35629-3_2
Kpalma, K., & Ronsin, J. (2007). An overview of advances of pattern recognition systems in computer Vision. In G. Obinata & A. Dutta (Eds.), *Vision systems: Segmentation and pattern recognition* (pp. 357–382). Vienna, Austria: IntechOpen. https://doi.org/10.5772/4960
Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of*

Systems and Software, *7*(4), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X
Lewins, A., & Silver, C. (2007). *Using software in qualitative research: A step-by-step guide*. London: Sage Publications.
Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. California: Sage Publications.
Lister, R., Fone, W., McCartney, R., Seppälä, O., Adams, E. S., Hamer, J., … Sanders, K. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119–150. https://doi.org/10.1145/1041624.1041673
Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, *38*(3), 118–122. https://doi.org/10.1145/1140123.1140157
Littman, D. C., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental models and software maintenance. *Journal of Systems and Software*, *7*(4), 341–355. https://doi.org/10.1016/0164-1212(87)90033-1
Litvinov, S., Mingazov, M., Myachikov, V., Ivanov, V., Palamarchuk, Y., Sozonov, P., & Succi, G. (2017). 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). Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda. https://doi.org/10.5220/0006336902350240
Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, *21*(1), 57–80. https://doi.org/10.1080/08993408.2011.554722
Maalej, W., Tiarks, R., Roehm, T., Koschke, R., Feigenspan, J., Siegmund, N., … Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering Methodology*, *23*(4), 31:1-31:37. https://doi.org/10.1145/2622669
Middendorf, J., & Baer, A. (2019). Bottlenecks of information literacy. In Craig Gibson & S. Mader (Eds.), *Building teaching and learning communities: Creating shared meaning and purpose* (pp. 51–68). Chicago: ACRL Publications.
Middendorf, J., & Pace, D. (2004). Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. *New Directions for Teaching and Learning*, *2004*(98), 1–12. https://doi.org/10.1002/tl.142
Middendorf, J., & Shopkow, L. (2018). *Overcoming student learning bottlenecks: Decode your disciplinary critical thinking*. Sterling, Virginia: Stylus Publishing, LLC.
Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *101*(2), 343–352. https://doi.org/10.1037/h0043158
Moore, D., Zabrucky, K., & Commander, N. E. (1997). Validation of the metacomprehension scale. *Contemporary Educational Psychology*, *22*, 457–471. https://doi.org/10.1006/ceps.1997.0946
Mosemann, R., & Wiedenbeck, S. (2001). Navigation and comprehension of programs by novice programmers. In *Proceedings of the 9th International Workshop on Program Comprehension* (pp. 79–88). Toronto, Ontario: IEEE. https://doi.org/10.1109/WPC.2001.921716
Nathan, M. J., & Petrosino, A. (2003). Expert blind spot among preservice teachers.

American Educational Research Journal, *40*(4), 905–928. https://doi.org/10.3102/00028312040004905

O'Brien, M. P., & Buckley, J. (2001). Inference-based and expectation-based processing in program comprehension. In *Proceedings of the 9th International Workshop on Program Comprehension*. Toronto: IEEE Computer Society Press. https://doi.org/10.1109/WPC.2001.921715

Pace, D. (2017). *The decoding the disciplines paradigm: Seven steps to increased student learning*. Bloomington: Indiana University Press.

Parcell, E. S., & Rafferty, K. A. (2017). Interviews, recording and transcribing. In M. Allen (Ed.), *The SAGE Encyclopedia of Communication Research Methods*. Thousand Oaks: Sage Publications, Inc. https://doi.org/10.4135/9781483381411.n275

Patton, M. Q. (2015). *Qualitative research & evaluation methods: Integrating theory and practice* (4th ed.). Thousand Oaks: Sage Publications.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In *Proceedings of the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers* (pp. 213–229). Washington, D.C.: Ablex Publishing Corp.

Plowright, D. (2011). *Using mixed methods: Frameworks for an integrated methodology*. London: Sage Publications.

Pope, C., Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 251–259). Melbourne, VIC: Association for Computing Machinery. https://doi.org/10.1145/2960310.2960324

Powell, N., Moore, D., Gray, J., Finlay, J., & Reaney, J. (2004). Dyslexia and learning computer programming. *SIGCSE Bull.*, *36*(3), 242. https://doi.org/10.1145/1026487.1008072

Saldaña, J. (2016). *The coding manual for qualitative researchers*. London: Sage Publications.

Sarkar, A. (2015). 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). Bournemouth, UK: Psychology of Programming Interest Group.

Schwandt, T. A., Lincoln, Y. S., & Guba, E. G. (2007). Judging interpretations: But is it rigorous? Trustworthiness and authenticity in naturalistic evaluation. *New Directions for Evaluation*, *2007*(114), 11–25. https://doi.org/10.1002/ev

Shopkow, L., Middendorf, J., Pace, D., Diaz, A., Middendorf, J., & Pace, D. (2013). From bottlenecks to epistemology: Changing the conversation about the teaching of history in colleges and Universities. In R. Thompson (Ed.), *Changing the conversation of higher education* (pp. 15–38). New York: Rowman & Littlefield.

Siegmund, J., Brechmann, A., Apel, S., Kästner, C., Liebig, J., Leich, T., & Saake, G. (2012). Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (pp. 1–4). Cary, North Carolina: Association for Computing Machinery.

https://doi.org/10.1145/2393596.2393624
Siegmund, J., Kástner, C., Apel, S., Brechmann, A., & Saake, G. (2013). Experience from measuring program comprehension - Toward a general framework. In S. Kowalewski & B. Rumpe (Eds.), *Software Engineering 2013 - Fachtagung des GI-Fachbereichs Softwaretechnik. GI-Edition - Lecture Notes in Informatics (LNI)* (Vol. P-213, pp. 239–257). Bonn: Gesellschaft für Informatik e.V.
Simon, B., Lopez, M., Sutton, K., & Clear, T. (2009). 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). Wellington, New Zealand: Australian Computer Society, Inc.
Srikant, S., & Aggarwal, V. (2014). 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). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/2623330.2623377
Tiarks, R. (2011). What maintenance programmers really do: An observational study. *Softwaretechnik-Trends*, *31*(2), 1–2.
Uzonwanne, F. C. (2016). Rational model of decision making. In A. Farazmand (Ed.), *Global Encyclopedia of Public Administration, Public Policy, and Governance* (pp. 1–6). Cham: Springer International Publishing AG. https://doi.org/10.1007/978-3-319-31816-5_2474-1
Von Mayrhauser, A., & Vans, A. M. M. (1995). Program understanding: Models and experiments. *Advances in Computers*, *40*, 1–38. https://doi.org/10.1016/S0065-2458(08)60543-4
Whalley, J., Prasad, C., & Kumar, A. (2007). Decoding doodles: Novice programmers and their annotations. In *Proceedings of the conferences in Research and Practice in Information Technology Series* (Vol. 66, pp. 171–180). Ballarat, Victoria, Australia: Australian Computer Society, Inc.
Xie, B., Nelson, G. L., & Ko, A. J. (2018). An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 344–349). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/3159450.3159527

Chapter 7 – Conclusions and Recommendations

7.1 Introduction

Building on the reality that SCC is a vital disciplinary skill that many higher education students continue to struggle with, this study was set out to explore how a systematic decoding approach can be used to uncover cognitive strategies for efficient SCC by novice programmers. In order to address the stated aim, this study was directed by two main research questions:

RQ1: What are the SCC challenges experienced by novice programmers?

RQ2: How can a systematic decoding approach be used to devise cognitive strategies that could be used to address these challenges?

In order to address these main research questions, the following nine subsidiary research questions were formulated:

- **Subsidiary research questions – (guiding the literature review)**

  SRQ1: What are the strategies that programmers (novices and experts) follow during the SCC process?

  SRQ2: What are the challenges that influence the development of novice programmers' SCC skills?

  SRQ3: How do cognitive and metacognitive practices influence SCC?

- **Subsidiary research questions – (directing the empirical investigations)**

  SRQ4 (a): What are the major SCC difficulties experienced by senior CS students?

  SRQ4 (b): How can these difficulties be used to identify more common SCC bottlenecks that should ideally already be addressed in introductory programming courses?

  SRQ5 (a): What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC?

SRQ5 (b): What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students?

SRQ6 (a): What are the explicit mental strategies (techniques and reasoning) that CS experts employ while comprehending source code?

SRQ6 (b): How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards narrowing the gap between expert and novice thinking with regard to efficient SCC?

In this chapter, a synthesis of the study findings (covering both the literature review and empirical investigations) is provided. This discussion is grouped according to the relevant research questions as stipulated above. Thereafter, the major contributions of this study are highlighted, and the study limitations are discussed. Next, recommendations for future research are outlined. Finally, an overall conclusion to the study is provided.

7.2 Synthesis of findings

In this section, a synthesis of the findings from the research study is presented. The discussion is structured around the individual research questions.

7.2.1 Literature review

A comprehensive literature review was conducted to provide answers to the first three subsidiary research questions.

SRQ1: What are the strategies that programmers (novices and experts) follow during the SCC process?

This part of the study established the three taxonomies of common SCC strategies that programmers use during the SCC process, which are bottom-up, top-down, and opportunistic (or mixed) strategies. However, it has become evident that the specific SCC strategies employed by programmers are dependent on their level of expertise as well as the resources they have at their disposal (see Section 2.2). In support of Storey et al.'s (2000) findings regarding SCC strategies, it could be suggested that

any strategy used by programmers to comprehend source code should allow them to: (1) use a single comprehension strategy or a combination of strategies as needed; (2) switch between SCC strategies when necessary; and (3) reduce their cognitive overload as much as possible while comprehending source code.

SRQ2: What are the challenges that influence the development of novice programmers' SCC skills?

Lack of prior knowledge, lack of problem solving skills, and lack of strong mental models have been identified as the main challenges that could influence the SCC ability of novice programmers (see Section 2.3). Regarding these challenges, it could be suggested that instructors should effectively engage students throughout the course period and ensure that course activities (e.g. assignments and learning tasks) compel students to continuously make efforts to gain and retain knowledge for future use (see Section 2.4.1.1). This can be achieved by integrating relevant content (e.g. topics, examples, and concepts) into the instructional strategies to help students learn SCC skills. Although the use of pragmatic comprehension strategies (see Section 2.2.1) seems to serve programmers of different levels of expertise well, it may not be ideal for students at elementary levels of programming to use such strategies. Instead, some standard guidelines should be developed for use by these students. Such guidelines should preferably be as detailed and explicit as possible.

SRQ3: How do cognitive and metacognitive practices influence SCC?

In an endeavour to understand the influence of cognitive and metacognitive practices on SCC (see Section 2.4), the literature review highlighted how critical these practices are in any learning environment (regardless of the discipline). It was noted in the study that nurturing these practices is not always easy. However, students who achieve success because of the practices, find it much easier to plan, monitor, and regulate the mental processes in their learning (Bergin et al., 2005; Pintrich, 1999; Simons & Bolhuis, 2004). Modelling has also been identified as an instructional strategy that can be used to foster metacognition (Ellis, Denton & Bond, 2014; Kistner et al., 2010).

The particular activities suggested as part of the planning, monitoring, and regulation strategies (Ambrose et al., 2010), could be incorporated as part of a 'modelling' instructional strategy. In executing such a modelling strategy, instructors could focus

on demonstrating or modelling the specific steps and/or best practice strategies that may be intrinsic to proper planning, monitoring, and regulation. The objective would be for students to experience first-hand how instructors (who can be regarded as experts) manoeuvre through these steps or strategies. Afterwards, students can be provided with opportunities to practise or implement these steps/strategies. Such a modelling strategy could be further enhanced by coupling it with scaffolding techniques.

For example, while modelling how best to comprehend source code, instructors can share with students the possible strategies they think can help them. After this, they can give reasons to students why such strategies might be useful. Ultimately, students could then select a strategy (or combination of strategies) that they think might work best for them. Instructors should also demonstrate flexibility – thereby encouraging their students to change to a different strategy if a selected strategy does not work for some reason. These are just some of the mental scaffolds believed to be vital for the modelling of efficient SCC. As such, this enhanced modelling strategy could be referred to as 'integrated' modelling. This strategy could be regarded as one of the most effective ways to foster cognitive and metacognitive practices in the learning process. Apart from using a think-aloud technique to demonstrate mental or expert moves by instructors (Middendorf & Pace, 2004), the integrated modelling strategy also combines all the advantages of the three metacognitive promotion strategies.

7.2.2 Empirical findings

The other six subsidiary research questions were answered as part of Article 1, Article 2, and Article 3. A discussion of how this was achieved is provided in the sub-sections that follow.

Article 1

Article 1 was focused on answering the following two subsidiary research questions:

SRQ4 (a): What are the major SCC difficulties experienced by senior CS students?

SRQ4 (b): How can knowledge of these difficulties be used to identify SCC bottlenecks that should ideally be addressed in introductory programming courses?

According to the literature, students experience several SCC difficulties. These difficulties are related to simple programming concepts as well as more advanced concepts. Premised on the research activities performed in this part of the study (i.e. SCC questions tackled), three main categories of specific difficulties with SCC were identified as experienced by senior CS students. These difficulties were related to arrays (array index, length of an array, Boolean array, and decomposition); programming logic (the ripple effect, guessing, and mathematical expressions); and program control (*for* loop). Although most students were able to comprehend these concepts when viewed in isolation, the level of misunderstanding often intensified when well-understood concepts were integrated with unknown concepts into a single piece of code. For example, if a student is able to compute the length of an array, but does not quite understand the working of an array index, and these two concepts are put together in a piece of code to be comprehended, the student easily gets even more confused.

Additionally, many students did not fully understand how a `for` repetition structure works – especially the difference between pre- and post-incrementation of the loop counter variable. While comprehending source code, many students seem to expect only concepts that they have studied before. As such, if they encounter something new, they get completely disorganised, and hence their thinking patterns become limited to some extent. When students fail to understand one line of code, they often forget that what a program achieves in the end is a collective of the individual lines of code contained in the program. Consequently, they deal with this confusion by completely ignoring challenging code statements. Ultimately, they then base their overall comprehension of a specific fragment of source code only on those parts that they were able to understand. Overall, the specified difficulties revealed teaching and learning gaps that instructors should not ignore in teaching SCC skills to novice programmers.

The aforementioned difficulties, in combination with knowledge from supporting literature and personal experiences of the researcher, resulted in the identification of six SCC bottlenecks (see Section 5 of Chapter 4). The study concluded that these bottlenecks should already be addressed in elementary programming courses. With regard to these bottlenecks, many students are not aware of effective strategies to

follow while trying to comprehend code. Consequently, they end up using unreliable strategies that are insufficient to successfully complete an SCC task. Moreover, due to a lack of proper strategies, the problem gets amplified when students are presented with larger SCC tasks.

Article 2

The focus of Article 2 was on answering the following two subsidiary research questions:

SRQ5 (a): What are the cognitive processes and related cognitive strategies employed by expert programmers during SCC?

SRQ5 (b): What does insight into these cognitive process strategies suggest in terms of mental scaffolding techniques for the modelling of efficient SCC strategies to students?

The literature revealed five relevant cognitive process categories (attention, perception, memory, reading, speaking and listening, and reflective cognition) that can be used for any problem-solving process in any discipline. Planning, cognitive reasoning, and decision making are distinct cognitive processes within the reflective cognition category. Although these cognitive processes may appear to be common, the study revealed fresh perspectives by which CS instructors can view and apply these processes in teaching. Noteworthy is that the perceptions identified in the experts in this study, caused them to focus on issues that were not directly related to the SCC question they were answering. Consequently, the perception-cognitive process was not included in the proposed mental scaffolding techniques for the modelling of efficient SCC.

Based on the aforementioned cognitive processes, 17 mental scaffolding techniques (directly linked to one or more of these cognitive processes) for efficient SCC were developed (see Table 1 in Chapter 5). It is proposed that programming instructors should use these techniques as an SCC teaching aid to convey expert ways of thinking and doing more explicitly to novice programmers. It is believed that execution of these techniques could act as a way to scaffold students' processes to better comprehend

source code, thereby improving their cognitive and metacognitive process skills in this regard.

Article 3

The focus of Article 3 was on answering the following two subsidiary research questions:

SRQ6 (a): *What are the explicit mental strategies (techniques and reasoning) that CS experts employ while comprehending source code?*

SRQ6 (b): *How can knowledge of these strategies be applied in the formulation of a step-by-step framework that could ultimately contribute towards narrowing the gap between expert and novice thinking with regard to efficient SCC?*

Premised on the nature of this part of the study, a set of mental strategies identified from the literature was used as a starting point for the identification of specific strategies employed by programming experts during SCC. Analysis of the data collected during the decoding interviews led to the identification of 15 key strategies for efficient SCC (see Section 4 in Article 3). Since the use of doodles or annotations was cited by Lister et al. (2004) as a strategy that experienced programmers typically use during SCC, it was surprising that this strategy was not used extensively by the experts in this study. In discussing the results of their study, Xie et al. (2018) noted that doodles are sometimes regarded as time-wasting by novice programmers. Under stringent time limitations, expert programmers might also regard doodles as time-consuming. Since this is not conclusive, it remains to be further investigated why doodles were not extensively used by the expert programmers in this study, particularly because they were not subjected to stringent time constraints. It was also interesting to note that, while Crosby and Stelovsky (1990) mentioned that both experienced and novice programmers pay least attention to the keywords in the source code text, the experts in this study regarded the identification of keywords from the problem description and code fragments provided to them, as one of their key strategies (see Section 4 in Chapter 6).

By coupling the 15 identified strategies with the insights gained regarding the explicit mental steps performed by the experts during the decoding interviews, a step-by-step framework for efficient SCC was compiled (see Table 2 in Chapter 6). The main intention of this framework is to create awareness among CS instructors regarding the explicit mental operations required for efficient SCC. Within the realm of the DtDs philosophy, the framework can also serve as a starting point for the planning of instructional strategies to explicitly model these mental operations (steps) to students and to help them master each of the identified strategies. It is believed that if the content of the framework is included in the planning of instructional activities in order to teach efficient SCC, students are highly likely to be equipped with skills that will enable them to think about and perform SCC tasks like experts.

7.2.3 Summary

This research study was originally set out to answer the following two main research questions:

RQ1: What are the SCC challenges experienced by novice programmers?

RQ2: How can a systematic decoding approach be used to devise cognitive strategies that could be used to address these challenges?

This study has revealed numerous SCC challenges and difficulties as experienced by novice programmers (both from literature and the empirical investigations of this study). Based on the Phase 1 and Phase 2 research activities, the eight principal SCC difficulties related to the concepts of arrays, programming logic, and programming control were identified. These were used to develop six useful bottlenecks – describing the main SCC challenges experienced by novice programmers. As a starting point for the systematic decoding approach followed in this study, it was shown that students can be a valuable source for the identification of bottlenecks that may hamper their understanding of the discipline-specific skill of SCC.

Given the variety of concepts covered by these six bottlenecks, the remainder of the empirical investigations focused mainly on Bottleneck 6 (*Students cannot reliably think their way through a long chain of reasoning required to comprehend a piece of source code*). As such, decoding interviews with expert programmers were used to explore the explicit steps that these programmers would follow to overcome the stated

bottleneck. The decoding interviews, performed as part of Step 2 of the DtDs framework, followed a different structure than suggested by the proponents of this framework (Middendorf & Pace, 2004). In this study, the decoding interviews started off in the typical manner – asking the participants to explain the steps they would follow while executing a discipline-specific task (to predict the output of any piece of source code provided on a piece of paper). The second part of the decoding interview was, however, less conventional. Here, each participant was presented with a real SCC problem and asked to illustrate how they would implement their previously shared SCC strategy in solving the given problem. By putting the participants in a situation where they had to illustrate their strategy in a 'real' situation, the researcher believes that he was able to gain much deeper insight into the explicit cognitive strategies required for efficient SCC.

Systematic analysis of all the data gathered as part of these interviews, guided by the relevant research questions, led to the development of the proposed mental scaffolding techniques for the modelling of efficient SCC (see Section 5.5 in Chapter 5), as well as a step-by-step framework for efficient SCC (see Section 5 in Chapter 6). It is envisioned that the proposed scaffolding techniques and the step-by-step framework could serve as a starting point for CS instructors to model the explicit strategies and steps as proposed by Step 3 of the DtDs framework. Ultimately, such modelled strategies could then be incorporated as part of instructional strategies specifically aimed at helping novice programmers to overcome the identified bottleneck and become more efficient in the comprehension of source code.

Given the selected focus on Bottleneck 6, explicit cognitive strategies that could help novice programmers overcome the other five bottlenecks, were not specifically explored in this study. It should, however, be noted that both the proposed mental scaffolding techniques for the modelling of SCC (see Chapter 5) and the step-by-step framework (see Section 5 of Chapter 6) contain information that could be of relevance in addressing some aspects of the other five bottlenecks.

Overall, this study has illustrated how a systematic decoding approach (encompassing adapted versions of Step 1 and Step 2 of the DtDs framework) can be used to devise cognitive strategies for efficient SCC. The identified cognitive strategies (and steps)

could ultimately be used by CS instructors to address some of the main SCC challenges experienced by novice programmers.

7.3 Contributions of the study

The main contributions of this research study can be described in three main categories. Firstly, this study has made a contribution to the practice of Computer Science Education by identifying specific SCC bottlenecks that point to student learning difficulties, which instructors should focus on in teaching introductory programming courses. As a way to assist these instructors, the study has proposed mental scaffolding techniques and a step-by-step framework for efficient SCC. These techniques and strategies could be integrated as part of instructional plans. The objective would be to create opportunities for students to better comprehend source code upon execution of these explicit steps and strategies. It is believed that applying these techniques and implementing the framework could ultimately help students to think about and perform SCC tasks more like experts.

Secondly, the study contributes to the enhancement of the DtDs paradigm in the following ways:

1. Create awareness among instructors regarding the role that a systematic decoding approach can play in exposing the mental processes or operations necessary for a complex discipline-specific task.
2. Show how a think-aloud technique (as part of Step 1 of the DtDs framework) can be used in a scientific manner to uncover the core of students' learning bottlenecks.
3. Demonstrate a unique decoding-interview approach (as part of Step 2 of the DtDs framework) where experts are observed and questioned (regarding their mental actions) while they perform an actual discipline-specific task.
4. Illustrate how decoding interviews (as part of Step 2 of the DtDs framework) can be used to also identify the actual cognitive processes followed by experts (in addition to just exposing the mental steps skipped by experts).

5. Create awareness of the possibilities the DtDs paradigm holds not only for the CS discipline, but also for other disciplines.

Third, the study contributes to the theory of Computer Science Education, as the proposed cognitive processes as well as steps and strategies can lead to new debates and potential new research directions. Specifically, the six identified SCC bottlenecks could lead to more vital insights and further investigations into ways students can be helped to overcome these bottlenecks. Moreover, this study has established various cognitive teaching and learning aids for instructors to use in their efforts to help students improve their SCC skills.

7.4 Limitations of the study

Narrative data-related threats, including trustworthiness issues such as credibility, transferability, dependability, confirmability, and integrity (see Section 3.4 – Chapter 3), as well as potential issues related to the specific research procedures (data source management strategies, population and sampling strategies, data collection methods, and data analysis) (see Section 3.3 – Chapter 3) have already been discussed. However, six perceived limitations of this study are worth pointing out.

First, the use of the selected set of 12 MCQs resulted in the identification of bottlenecks that were specifically related to the concepts tested in these questions. Bottlenecks related to other difficult programming concepts, such as recursion (Sanders & Mccartney, 2016), could not come out from the data set of this study. This suggests that the bottlenecks identified in this study are not comprehensive. The study did, however, illustrate the usefulness of a novel research design and methodology to identify bottlenecks specific to the CS discipline.

Second, the original intention was to engage one decoding interviewer from outside the CS discipline, as suggested by the proponents of the DtDs framework (Middendorf & Pace, 2004). However, due to the unavailability of an individual with the relevant decoding-interview experience, an interviewer from within the CS discipline was engaged. Using a second interviewer from outside the discipline could therefore have

resulted in additional insights regarding the explicit mental strategies and steps followed by expert programmers.

Third, the researcher also acknowledges his own limitations as far as insight and experience in the fields of Computer Science Education and general research are concerned. However, he believes that many of the personal limitations were mitigated by the valuable guidance and suggestions received from his experienced supervisor, other colleagues in the field, as well as those who evaluated and validated some of the findings (see Section 3.5.3 – Chapter 3).

Fourth, no full-time professional programmers were involved in the decoding interviews. It should, however, be noted that in DtDs studies, experienced educators or instructors are typically regarded and used as experts (Middendorf & Pace, 2004; Pace, 2017a).

Fifth, as an oversight on the side of the researcher, the limited doodles made by the expert programmers during the decoding interviews were not retained for further analysis. Having access to these artefacts as an additional source of data could have helped to enhance the discussion of the study findings.

Lastly, this research study was conducted within a specific context (a selected South African higher education institution). The study was also focused on identifying the SCC challenges experienced by a very specific population (senior CS students). Due to the exploratory nature of this research study, no claims can therefore be made to the generalisability of the study findings.

7.5 Recommendations for future research

The problems experienced by novice programmers and strategies to address these problems have been an ongoing research focus for the past 40 years. The focal point of this research study was to explore how a systematic decoding approach could be used to uncover cognitive strategies for efficient SCC by novice programmers. The investigation was based on a specific number of research activities that focused on a specific population of university students studying at a senior level. Natural extensions

of this work could be conducted in a different setting, perhaps with participants from a higher level of study (e.g. honours or master's). The objective would be to ascertain whether there would be variations in the findings when more advanced senior students are used as participants.

Furthermore, the inclusion of professionals solely from the programming industry (as expert participants) could also be considered. The objective would be to determine the extent to which full-time programmers do things differently or similarly (e.g. thought processes) to those who teach programming with only limited industry experience. A similar study can further be conducted with a different set of SCC questions. These questions can, for example, be obtained from the literature or past examination papers or be developed by researchers from scratch. The use of such questions could help to identify additional bottlenecks related to programming concepts not specifically covered in the 12 MCQs used in this study.

Although six bottlenecks were identified in Article 1 (see Chapter 4), only one bottleneck was explored further in this study. Therefore, similar studies can be conducted to address the other five bottlenecks. Based on the proposed step-by-step framework for efficient SCC, some of the identified strategies and/or steps (e.g. making a trace table; drawing a diagram to visualise an understanding of the program logic; and optimising the selection of test cases to use in confirming answers) could be explored even further. Such exploration might lead to even more simplistic and optimal steps to follow as part of an efficient SCC strategy. Insights gained from this study regarding the general cognitive and metacognitive strategies employed by expert programmers could serve as a stepping-stone for further exploration of the more detailed step-by-step procedures that experts follow while comprehending source code.

Natural extensions of this study could focus on the remaining DtDs steps not covered by the study. Since the study only focused on the first two steps of the seven-step DtDs framework, follow-up investigations are needed to explicitly model the identified steps and strategies in a format that will be usable/understandable to students (as part of Step 3). As part of Step 4, instructors can then develop instructional strategies (including specific assignments, team activities, and other learning exercises) that will

provide students with opportunities to practise each of the defined tasks and get feedback on their mastery of that skill. As part of Step 5, instructors can identify and test specific strategies to motivate students to incorporate the modelled strategies as part of their own processes. As part of Step 6, instructors need to develop assessment strategies that could be used to determine if and how the modelled tasks have helped students to overcome the stated bottleneck. Similar to the ways in which the researcher of this study has shared his experiences and findings (through this thesis report and the publications that have transpired from the research study), other researchers are encouraged to do the same (as part of Step 7 of the DtDs framework) (Middendorf & Pace, 2004).

7.6 Conclusion

The findings of this study are mostly pedagogical in nature and therefore present a promising avenue for a renewed focus on the explicit teaching and learning of SCC skills. The way in which this study was approached and how the investigations unfolded (the identification of the SCC bottlenecks, mental scaffolding techniques for efficient SCC, and step-by-step framework for efficient SCC), have revealed a further need for continuous research in this area. Specifically, empirical findings from this study in relation to the six SCC bottlenecks that have been identified, contain a noteworthy element of 'surprise' through which older theories or scholarly opinions can be improved or contested. It is believed that these fresh perspectives can help instructors to theoretically and practically enhance the field of Computer Science Education.

This study also intended to make a contribution to the field of Computer Science Education. The researcher’s efforts revealed and highlighted the complexity of the problems encountered by both CS students and instructors regarding the learning and teaching of SCC skills. Throughout the study, it was further established that there are no shortcuts in overcoming these problems. Instead, explicit scientific solutions that speak directly to the minds (i.e. cognition and metacognition) of students should be devised and used. The Computer Science Education field also needs more researchers who can continue the search for new ways in which discipline-specific teaching and learning problems can be uncovered and solutions sought. The DtDs

paradigm has been shown as a promising and alternative way of investigating these aspects. This approach can and should be further exploited, but there are certainly many other new ways to conduct research in Computer Science Education waiting to be uncovered or discovered. As we are on the threshold of the Fourth Industrial Revolution, ‘reimagining the future for all disciplines’ – in CS we also have to prepare our students for new developments in the field and for jobs that do not even exist at the moment (D2L, 2018). The CS instructors of today need to be innovative, creative, and proactive if they want to make a lasting contribution to the field of Computer Science Education beyond the 2020 era.

List of References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, *9*(4), 422–433. https://doi.org/10.3758/BF03197568

Adelson, B. (1983). *Structure and strategy in the semantically-rich domains*. Harvard University, Cambridge.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*(3), 483–495. https://doi.org/10.1037/0278-7393.10.3.483

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering*, *SE-11*(11), 1351–1360. https://doi.org/10.1109/TSE.1985.231883

Akturk, A. O., & Sahin, I. (2011). Literature review on metacognition and its measurement. *Procedia - Social and Behavioral Sciences*, *15*, 3731–3736. https://doi.org/10.1016/j.sbspro.2011.04.364

Alam, A., & Padenga, T. (2010). *Application software reengineering*. New Delhi: Dorling Kindersley (India).

Allert, J. (2004). Learning style and factors contributing to success in an introductory Computer Science course. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies* (pp. 385–389). Joensuu, Finland: IEEE. https://doi.org/10.1109/ICALT.2004.1357442

Alturki, R. A. (2016). Measuring and improving student performance in an introductory programming course. *Informatics in Education*, *15*(2), 183–204. https://doi.org/10.15388/infedu.2016.10

Ambrose, A. S., Bridges, W. M., DiPietro, M., Lovett, C. M., & Norman, K. M. (2010). *How learning works : Seven research-based principles for smart teaching*. California: John Wiley & Sons. https://doi.org/10.1002/mop.21454

Anderson, N. J., Bachman, L., Perkins, K., & Cohen, A. (1991). An exploratory study into the construct validity of a reading comprehension test: triangulation of data sources. *Language Testing*, *8*(1), 41–66. https://doi.org/10.1177/026553229100800104

Arksey, H., & Knight, P. (1999). *Interviewing for Social Scientists*. London: Sage Publications.

Astrachan, O., & Rodger, S. H. (1998). Animation, visualization, and interaction in CS1 assignments. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer Science Education* (pp. 317–321). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/273133.274321

Azevedo, R., Cromley, J. G., Winters, F. I., Moos, D. C., & Greene, J. A. (2005). Adaptive human scaffolding facilitates adolescents' self-regulated learning with hypermedia. *Instructional Science*, *33*(5–6), 381–412. https://doi.org/10.1007/s11251-005-1273-8

Bailey, J. (2008). First steps in qualitative data analysis: Transcribing. *Family Practice*, *25*(2), 127–131. https://doi.org/10.1093/fampra/cmn003

Barkley, E. F. (2010). *Student engagement techniques : A handbook of college faculty*. California: Jossey-Bass.

Basili, V. R., & Mills, H. D. (1982). Understanding and documenting programs. *IEEE Transactions on Software Engineering*, *3*(SE-8), 270–283. https://doi.org/10.1109/TSE.1982.235255

Bednarik, R., & Tukiainen, M. (2006). An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on eye tracking research & applications - ETRA '06* (pp. 125–132). https://doi.org/10.1145/1117309.1117356

Belanger, E. (2017). Using US Tuning to effect: The American Historical Association's Tuning Project and the first year research paper. *Arts and Humanities in Higher Education*, *16*(4), 385–402. https://doi.org/10.1177/1474022216628379

Bergin, S., Reilly, R., & Traynor, D. (2005). Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the First International Workshop on Computing Education Research* (pp. 81–86). Seattle,Washington: ACM. https://doi.org/10.1145/1089786.1089794

Bickhard, M. H. (2013). Scaffolding and self-scaffolding: Central aspects of development. In L. T. Winegar & J. Valsiner (Eds.), *Children's development within social context, Volume 1: Metatheory and theory; Volume 2: Research and methodology* (1st ed., pp. 33–52). New Jersey: Lawrence Erlbaum Associates.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of 1993 15th International Conference on Software Engineering* (pp. 482–498). Baltimore, MD: IEEE Computer Society Press. https://doi.org/10.1109/ICSE.1993.346017

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Bitsch, V. (2005). Qualitative research : A grounded theory example and evaluation criteria. *Journal of Agribusiness*, *23*(1), 75–91. https://doi.org/10.1002/cssc.201300370

Boman, J., Currie, G., MacDonald, R., Miller-Young, J., Yeo, M., & Zette, S. (2017). Overview of decoding across the disciplines. *New Directions for Teaching and Learning*, (150), 13–18. https://doi.org/10.1002/tl

Bosse, Y., & Gerosa, M. A. (2017). Difficulties of programming learning from the point of view of students and instructors. *IEEE Latin America Transactions*, *15*(11), 2191–2199. https://doi.org/10.1109/TLA.2017.8070426

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09* (p. 1589). Boston, Massachusetts: ACM. https://doi.org/10.1145/1518701.1518944

Bransford, J., Brown, A., & Cocking, R. (2000). *How people learn: Brain, mind, experience, and school* (Expanded). Washington, DC: National Academy Press. https://doi.org/10.4135/9781483387772.n2

Braught, G., Wahls, T., & Eby, L. M. (2011). The case for pair programming in the Computer Science classroom. *ACM Transactions on Computing Education*, *11*(1), 1–21. https://doi.org/10.1145/1921607.1921609

Braun, V., & Clarke, V. (2006). Using thematic analysis in Psychology. *Qualitative Research in Psychology*, *3*(2), 77–101. https://doi.org/10.1191/1478088706qp063oa

Brookhart, S. M. (2008). *How to give effective feedback to your students*. Alexandria: Association for Supervision and Curriculum Development.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Brooks, R. (1999). Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies*, *51*(2), 197–211. https://doi.org/10.1006/ijhc.1977.0306

Bryman, A. (2006). Integrating quantitative and qualitative research: how it is done. *Qualitative Research*, *6*(1), 97–113. https://doi.org/10.1177/1468794106058877

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, *7*(2), 115–156. https://doi.org/10.1023/A:1015297914742

Busjahn, T., & Schulte, C. (2013). The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (pp. 3–11). New York, NY: ACM. https://doi.org/10.1145/2526968.2526969

Busjahn, T., Schulte, C., & Busjahn, A. (2011). Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (pp. 1–9). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/2094131.2094133

Butler, M., & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In R. Atkinson, C. McBeath, A. S. Swee Kit, & C. Cheers (Eds.), *Proceedings of ascilite Singapore 2007 ICT: Providing Choices for Learners and Learning* (pp. 99–107). Nanyang Singapore: Nanyang Technological University.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human - Computer Studies*, *40*, 795–811. https://doi.org/10.1006/ijhc.1994.1038

Carroll, J. M. (2003). *Minimalism beyond the Nurnberg Funnel*. Cambridge, Massachusetts: MIT Press.

Charters, E. (2003). The use of think-aloud methods in qualitative research: An Introduction to think-aloud methods. *Brock Education*, *12*(2), 68–82. https://doi.org/10.1080/02602938.2010.496532

Chi, Michelene T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanationss: How students study and use examples in learning to solve problems. *Cognitive Science*, *13*, 145–182. https://doi.org/10.1207/s15516709cog1302_1

Chilisa, B., & Preece, J. (2005). *Research methods for adult educators in Africa*. Cape Town: Pearson Education, Inc.

Chu, X., Ilyas, I. F., Krishnan, S., & Wang, J. (2016). Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data* (pp. 2201–2206). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/2882903.2912574

Cimitile, A., Tortorella, M., & Munro, M. (1994). Program comprehension through the identification of abstract data types. In *Proceedings of the 1994 IEEE 3rd Workshop on Program Comprehension* (pp. 12–19). Washington, DC: IEEE. https://doi.org/10.1109/wpc.1994.341243

Cognifit. (2019). Cognitive processes: What are they? Can they improve? Retrieved September 4, 2019, from https://www.cognifit.com/cognition

Cooper, D., & Schindler, P. (2013). *Business research methods* (12th ed.). New York: McGraw-Hill Education.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, *3*(2), 199–222. https://doi.org/10.1080/10447319109526004

Council on Higher Education. (2016). *South African higher education reviewed: Two decades of democracy*. Pretoria: Council on Higher Education. https://doi.org/10.1080/02642060701453288

Creswell, J. W. (2014). *Research design: Qualitative, quantitative, and mixed methods approaches* (4th ed.). Thousand Oaks: Sage Publications.

Creswell, J. W., & Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches* (5th ed.). Thousand Oaks: Sage.

Creswell, J. W., & Plano Clark, V. L. (2011). *Designing and conducting mixed methods research* (2nd ed.). Thousand Oaks: Sage Publications.

Cronje, J. (2013). What is this thing called “design” in design research and instructional design. *Educational Media International*, *50*(1), 1–11. https://doi.org/10.1080/09523987.2013.777180

Crosby, M. E., & Stelovsky, J. (1990). How do we read algorithms?: A case study. *Computer*, *23*(1), 25–35. https://doi.org/10.1109/2.48797

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using tracing and sketching to solve programming problems. *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER ’17*, 164–

172. https://doi.org/10.1145/3105726.3106190

D2L. (2018). *The future of work and learning : In the age of the 4th Industrial Revolution*. Kitchener: D2L. Retrieved from www.D2L.com.

Davies, S. P. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies*, *32*(4), 461–481. https://doi.org/10.1016/S0020-7373(05)80143-9

Davis, E. A. (2000). Scaffolding students' knowledge integration: Prompts for reflection in KIE. *International Journal of Science Education*, *22*(8), 819–837. https://doi.org/10.1080/095006900412293

Deejring, K. (2015). The validation of web-based learning using collaborative learning techniques and a scaffolding system to enhance learners' competency in higher education. *Procedia - Social and Behavioral Sciences*, *174*, 34–42. https://doi.org/10.1016/j.sbspro.2015.01.623

Détienne, F. (1990). Expert programming knowledge: A schema-based approach. In J. M. Hoc, T. R. G. Green, R. Samurcay, & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 205–222). London: Academic Press. https://doi.org/10.1016/B978-0-12-350772-3.50018-5

Diaz, A., Middendorf, J., Pace, D., Shopkow, L., Díaz, A., Middendorf, J., … Shopkow, L. (2008). The History learning project: A department "decodes" its students. *Journal of American History*, *94*(4), 1211–1224. https://doi.org/10.2307/25095328

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Easton, K. L., McComish, J. F., & Greenberg, R. (2000). Avoiding common pitfalls in qualitative data collection and transcription. *Qualitative Health Research*, *10*(5), 703–707. https://doi.org/10.1177/104973200129118651

Eisenführ, F., Weber, M., & Langer, T. (2010). *Rational decision making*. Berlin: Springer-Verlag Berlin Heidelberg.

Ellis, A. K., Denton, D. W., & Bond, J. B. (2014). An analysis of research on metacognitive teaching strategies. *Procedia - Social and Behavioral Sciences*, *116*(2014), 4015–4024. https://doi.org/10.1016/j.sbspro.2014.01.883

Eranki, K. L. N., & Moudgalya, K. M. (2016). Program slicing technique : A novel approach to improve programming skills in novice learners. In *Proceedings of*

the 17th Annual Conference on Information Technology Education (pp. 160–165). New York: Association for Computing Machinery. https://doi.org/10.1145/2978192.2978215

Faux, R. (2001). Teaching problem solving techniques and software engineering concepts before programming. Retrieved January 16, 2018, from http://www.micsymposium.org/mics_2001/faux1.pdf

Feyzi-Behnagh, R., Azevedo, R., Legowski, E., Reitmeyer, K., Tseytlin, E., & Crowley, R. S. (2014). Metacognitive scaffolds improve self-judgments of accuracy in a medical intelligent tutoring system. *Instructional Science*, *42*(2), 159–181. https://doi.org/10.1007/s11251-013-9275-4

Fisler, K. (2014). The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (pp. 35–42). Glasgow, Scotland: ACM. https://doi.org/10.1145/2632320.2632346

Fitzgerald, S., Simon, B., & Thomas, L. (2005). 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). New York: Association for Computing Machinery. https://doi.org/10.1145/1089786.1089793

Flavell, J. H. (1976). Metacognitive aspects of problem solving. In L. Resnick (Ed.), *The Nature of Intelligence* (pp. 231–236). Hillsdale: Lawrence Erlbaum Associates.

Frith, C. D. (2012). The role of metacognition in human social interactions. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *367*(1599), 2213–2223. https://doi.org/10.1098/rstb.2012.0123

German, A., Menzel, S., Middendorf, J., & Duncan, F. J. (2014). How to decode student bottlenecks to learning in Computer Science (abstract only). In *Proceedings of the 45th ACM technical symposium on Computer Science Education* (pp. 733–733). Bloomington: ACM. https://doi.org/10.1145/2538862.2544228

Gibbs, G. R. (2018). *Analysing qualitative data* (2nd ed.). London: Sage Publications.

Gilmore, D. J., & Green, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology Section A*, *40*(3), 423–442. https://doi.org/10.1080/02724988843000005

Green, T. R. G., & Navarro, R. (1995). Programming plans, imagery, and visual

programming. In K. Nordby, P. Helmersen, D. J. Gilmore, & S. A. Arnesen (Eds.), *Human Computer Interaction. IFIP Advances in Information and Communication Technology* (pp. 139–144). Boston, MA: Springer.

Greeno, J. G., Collins, A. M., & Resnick, L. B. (1996). Cognition and Learning. In D. C. Berliner & R. C. Calfee (Eds.), *Handbook of Educational Psychology* (pp. 15–46). New York: Prentice Hall International.

Grossman, P. L. (1990). *The making of a teacher: Teacher knowledge and teacher education*. New York: Teachers College Press.

Guba, E. G. (1981). Criteria for assessing the trustworthiness of naturalistic inquiries. *Educational Communication and Technology*, *29*(2), 75–91.

Guba, E. G., & Lincoln, Y. S. (1982). Establishing dependability and confirmability in naturalistic inquiry through an Audit. In *Proceedings of the Annual Meeting of the American Educational Research Association*. New York: American Educational Research Association.

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In *Proceedings of the first workshop on empirical studies of programmers on empirical studies of programmers* (pp. 13–27). Washington, D.C.: Ablex Publishing Corp.

Hart, T. (2015). Technologies for conducting an online ethnography of communication: The case of Eloqi. In H. Shalin (Ed.), *Enhancing qualitative and mixed methods research with technology* (pp. 105–124). Hershey: Information Science Reference (an imprint of IGI Global).

Haworth, J. G., & Conrad, C. F. (1997). *Emblems of quality in higher education*. Boston: Allyn & Bacon.

Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to teaching Computer Science : An activity-based approach*. London: Springer-Verlag. https://doi.org/10.1017/CBO9781107415324.004

Hellawell, D. (2016). Using mixed methods Frameworks for an Integrated Methodology : Reviews. Retrieved September 14, 2017, from https://us.sagepub.com/en-us/nam/using-mixed-methods/book233419#reviews

Hennessey, M. G. (1999). Probing the dimensions of metacognition: Implications for conceptual change teaching-learning. In *Proceedings of the Annual Meeting of the National Association for Research in Science Teaching* (pp. 1–33). Boston, MA: The National Association for Research in Science Teaching.

Herrmann, J. W. (2017). Rational decision making. In N. Balakrishnan, T. Colton, B. Everitt, W. Piegorsch, F. Ruggeri, & J. L. Teugels (Eds.), *Wiley StatsRef: Statistics Reference Online* (pp. 1–9). John Wiley & Sons Ltd. https://doi.org/10.1002/9781118445112.stat07928

Hesse-Biber, S. N. (2010). *Mixed methods research: Merging theory with practice*. New York: Guilford Press.

Hinds, P. S., Vogel, R. J., & Clarke-Steffen, L. (1997). The possibilities and pitfalls of doing a secondary analysis of a qualitative data set. *Qualitative Health Research*, *7*(3), 408–424. https://doi.org/10.1177/104973239700700306

Holmes, R., & Walker, R. J. (2012). Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology*, *21*(4), 20:1-20:44. https://doi.org/10.1145/2377656.2377657

Holton III, R. A., & Swanson, E. F. (2005). *Research in organizations: Foundations and methods of inquiry*. San Francisco, CA: Berrett Koehler Publications.

Indiana University. (2019). Decoding the disciplines - Improving student learning. Retrieved December 20, 2019, from http://decodingthedisciplines.org/bibliography/

Ismail, M. N., Ngah, N. A., & Umar, I. N. (2010). Instructional strategy in the teaching of computer programming: A need assessment analyses. *Turkish Online Journal of Educational Technology*, *9*(2), 125–131.

IUBCITL. (2016). Team-based learning for practice and motivation. Retrieved October 18, 2017, from https://www.youtube.com/watch?v=1obB-n6JZ8k

Iv, D. H. S., Jagodzinski, F., Hao, Q., Liu, Y., & Gupta, V. (2019). Quantifying the effects of prior knowledge in entry-level programming courses. In *Proceedings of the ACM Conference on Global Computing Education* (pp. 30–36). Chengdu, Sichuan, China: ACM. https://doi.org/10.1145/3300115.3309503

Jansen, J. (2003). The state of higher education in South Africa: From massification to mergers. In J. Daniel, A. Habib, & R. Southall (Eds.), *State of the Nation: South Africa 2003-2004*. Pretoria: HRSC.

Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. In *Proceedings of AERA annual meeting* (pp. 1–17). New York: American Educational Research Association.

Jeffries, R., Turner, a. a., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive Skills and*

Their Acquisition (pp. 255–283). Hillsdale, NJ: Lawrence Erlbaum Associates.

Kim, M., Bergman, L., Lau, T., & Notkin, D. (2004). An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)* (pp. 83–92). Redondo Beach, CA: IEEE. https://doi.org/10.1109/ISESE.2004.1334896

King, K., Linkon, S., & Middendorf, J. (2013). Decoding the disciplines and threshold concepts. Retrieved January 21, 2017, from https://www.youtube.com/watch?v=Wqe_kKFoOq4

Kinnunen, P. (2009). *Challenges of teaching and studying programming at a university of technology - viewpoints of students, teachers and the university*. *Department of Computer Science and Engineering*. PhD thesis, Department of Computer Science and Engineering, Helsinki University of Technology.

Kirkpatrick, M. S., & Mayfield, C. (2017). Evaluating an alternative CS1 for students with prior programming experience. In *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE* (pp. 333–338). Seattle, Washington: ACM. https://doi.org/10.1145/3017680.3017759

Kistner, S., Rakoczy, K., Otto, B., Dignath-van Ewijk, C., Büttner, G., & Klieme, E. (2010). Promotion of self-regulated learning in classrooms: Investigating frequency, quality, and consequences for student performance. *Metacognition and Learning*, *5*(2), 157–171. https://doi.org/10.1007/s11409-010-9055-3

Ko, A. J., & Myers, B. A. (2004). Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 151–158). Vienna, Austria: ACM. https://doi.org/10.1145/985692.985712

Ko, A. J., & Uttl, B. (2003). Individual differences in program comprehension strategies in unfamiliar programming systems. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC: IEEE Computer Society Press. https://doi.org/10.1109/WPC.2003.1199201

Koenemann, J., & Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 125–130). New Orleans, Louisiana: ACM. https://doi.org/10.1145/108844.108863

Kuhn, J. (2014). *Fear and learning in America: Bad data, good teachers, and the*

attack on public education. New York: Teachers College Press.

Lahm, S., & Kaduk, S. (2016). Essay on decoding the disciplines as a starting point for research-based teaching and learning. In H. A. Mieg & J. Lehmann (Eds.), *Learning through research: A practical handbook*. FHP-Verlag.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGSCE Conference on Innovation and Technology in Computer Science Education* (pp. 14–18). Monte de Caparica, Portugal: ACM Press. https://doi.org/10.1145/1151954.1067453

Lai, E. R. (2011). Metacognition : A literature review research report. Retrieved October 20, 2018, from https://studylib.net/doc/12091843/metacognition--a-literature-review-research-report

LaToza, T. D., Garlan, D., Herbsleb, J. D., & Myers, B. A. (2007). Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 361–370). Dubrovnik, Croatia: ACM. https://doi.org/10.1145/1287624.1287675

Lee, M. J., & Ko, A. J. (2015). Comparing the effectiveness of online learning approaches on CS1 learning outcomes. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 237–246). Omaha, Nebraska: ACM. https://doi.org/10.1145/2787622.2787709

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software*, *7*(4), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

Letovsky, Stan, & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, *3*(3), 41–49. https://doi.org/10.1109/MS.1986.233414

Lewin, K. (1951). *Field theory in social science: Selected theoretical papers*. New York: Harper & Brothers.

Lewins, A., & Silver, C. (2007). *Using software in qualitative research: A step-by-step guide*. London: Sage Publications.

Liamputtong, P. (2009). Qualitative data analysis: conceptual and practical considerations. *Health Promotion Journal of Australia*, *20*(2), 133–139. https://doi.org/10.1071/HE09133

Liffick, B. W., & Aiken, R. (1996). A novice programmer's support environment. In

Proceedings of the 1st Conference on Integrating Technology into Computer Science Education (pp. 49–51). Barcelona, Spain: ACM. https://doi.org/10.1145/1013718.237525

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. California: Sage Publications.

Lister, R., Fone, W., McCartney, R., Seppälä, O., Adams, E. S., Hamer, J., … Thomas, L. (2004). *A multi-national study of reading and tracing skills in novice programmers*. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)* (Vol. 36). https://doi.org/10.1145/1041624.1041673

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 118–122). Bologna: ACM. https://doi.org/http://doi.acm.org/10.1145/1140124.1140157

Littman, D. C., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental models and software maintenance. *Journal of Systems and Software*, *7*(4), 341–355. https://doi.org/10.1016/0164-1212(87)90033-1

Littmann, P., Pinto, J., Letovsky, S., & Soloway, E. (1986). Software maintenance and mental models. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers*. New York: Ablex Publishing Corporation.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 101–112). Sydney, Australia: ACM. https://doi.org/10.1145/1404520.1404531

Lovett, M. (2008). Teaching metacognition. Retrieved January 28, 2018, from https://events.educause.edu/ir/library/pdf/ELI08104.pdf

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, *39*(1), 499–503. https://doi.org/10.1145/1227504.1227481

Maalej, W., Tiarks, R., Roehm, T., & Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering Methodology*, *23*(4), 1–37. https://doi.org/10.1145/2622669

MacMillan, M., Yeo, M., Currie, G., Pace, D., McCollum, B., & Miller-Young, J.

(2016). The decoding interview, live and unplugged. Retrieved August 23, 2018, from https://mruir.mtroyal.ca/xmlui/handle/11205/355

Malarz, L. (1998). Bilingual education: Effective programming for language-minority students. In ASCD (Ed.), *Curriculum Handbook*. Alexandria, VA: Association for Supervision and Curriculum Development.

Marshall, C., & Rossman, G. B. (2016). *Designing qualitative research* (6th ed.). Thousand Oaks: Sage Publications, Inc.

Mccartney, R., Boustedt, J., Eckerdal, A., Sanders, K., & Zander, C. (2013). Can first-year students program yet? A study revisited. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (pp. 91–98). San Diego, California: ACM. https://doi.org/10.1145/2493394.2493412

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., … Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, *33*(4), 125–180. https://doi.org/10.1145/572139.572181

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, *13*(3), 307–325. https://doi.org/10.1016/0010-0285(81)90012-8

Menzel, S. (2015). Recursion as a bottleneck in Computer Science. In *Proceedings of the 12th annual conference of the International Society for the Scholarship of Teaching and Learning*. Melbourne, Australia: The International Society for the Scholarship of Teaching & Learning (ISSOTL). Retrieved from https://eprints.usq.edu.au/28746/7/ISSOTL 2015 Program Book WEB.pdf

Meyer, J. H. F., & Land, R. (2003). Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. In C. Rust (Ed.), *Improving student learning – Ten years on* (pp. 1–16). Oxford: Oxford Centre for Staff and Learning Development (OCSLD).

Mhashi, M. M., & Alakeel, A. (2013). Difficulties facing students in learning computer programming skills at Tabuk University. In *Proceedings of the 12th International Conference on Education and Educational Technology* (pp. 15–24). Morioka City, Japan: WSEAS Press.

Middendorf, J. K., & Pace, D. (2004). Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. *New Directions for Teaching and*

Learning, *2004*(98), 1–12. https://doi.org/10.1002/tl.142

Miller-Young, J., & Boman, J. (Eds.). (2017). Foreword. In *Using the Decoding The Disciplines Framework for Learning Across the Disciplines*. San Francisco: Jossey-Bass.

Minnesota Historical Society. (2001). *Transcribing, editing and processing guidelines*. Saint Paul, Minnesota: Minnesota Historical Society.

Mishra, J., & Mohanty, A. (2012). *Software Engineering*. New Delhi: Dorling Kindersley (India) Pvt. Ltd.

Molenaar, I., Van Boxtel, C. A. M., & Sleegers, P. J. C. (2011). Metacognitive scaffolding in an innovative learning arrangement. *Instructional Science*, *39*, 785–803. https://doi.org/10.1007/s11251-010-9154-1

Moon, J. A. (2004). *A handbook of reflective and experiential learning: Theory and practice*. New York: Routledge.

Moore, D., Zabrucky, K., & Commander, N. E. (1997). Validation of the metacomprehension scale. *Contemporary Educational Psychology*, *22*, 457–471. https://doi.org/10.1006/ceps.1997.0946

Morse, J. M. (2003). Principles of mixed method and multimethod research design. In A. Tashakkori & C. Teddlie (Eds.), *Handbook of mixed methods in social & behavioral research* (pp. 189–208). Thousand Oaks: Sage Publications.

Mosemann, R., & Wiedenbeck, S. (2001). Navigation and comprehension of programs by novice programmers. In *Proceedings of the 9th International Workshop on Program Comprehension* (pp. 79–88). Toronto, Ontario: IEEE. https://doi.org/10.1109/WPC.2001.921716

Nanja, M., & Cook, C. R. (1987). An analysis of the on-line debugging process. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: second workshop* (pp. 172–184). New Jersey: Ablex Publishing Corporation.

Nathan, A. J., & Scobell, A. (2012). How China sees America. *Foreign Affairs*. London: Sage. https://doi.org/10.1017/CBO9781107415324.004

Nathan, M. J., & Koedinger, K. R. (2000). An investigation of teachers' beliefs of students' Algebra development. *Cognition and Instruction*, *18*(2), 209–237. https://doi.org/10.1207/S1532690XCI1802_03

Nathan, M. J., Koedinger, K. R., & Alibali, M. W. (2001). Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third International Conference on Cognitive Science* (pp. 644–648). Beijing,

China: USTC Press.

Nathan, M. J., & Petrosino, A. (2003). Expert blind spot among preservice teachers. *American Educational Research Journal*, *40*(4), 905–928. https://doi.org/10.3102/00028312040004905

Nilson, L. B. (2013). *Creating self-regulated learners: Strategies to strengthen student's self-awareness and learning skills*. Sterling: Stylus.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental Models* (pp. 241–244). New Jersey: Morgan Kaufmann Publishers, Inc.

O' Kelly, J., Bergin, S., Dunne, S., Gaughran, P., Ghent, J., & Mooney, A. (2004). Initial findings on the impact of an alternative approach to problem based learning in Computer Science. In *Proceedings of the PBL Conference*. Cancun, Mexico: Tecnologico De Monterrey.

O'Brien, M. (2003). *Software Comprehension - A review and research direction*. Department of Computer Science & Information Systems, University of Limerick, Limerick, Ireland.

Orlov, P. A., Bednarik, R., & Orlova, L. (2016). Programmers' experiences with working in the restricted-view mode as indications of parafoveal processing differences. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group* (pp. 96–105). St. Catharine's College, University of Cambridge, UK: PPIG.

Oroma, J., Wanga, H., & Ngumbuke, F. (2012). Challenges of teaching and learning computer programming in a developing country: Lessons from Tanzania. In *Proceedings of the 6th International Technology, Education and Development Conference* (pp. 3820–3826). Valencia, Spain: IATED. https://doi.org/10.13140/2.1.3836.6407

Pace, D. (2017a). *The decoding the disciplines paradigm: Seven steps to increased student learning*. Bloomington: Indiana University Press.

Pace, D. (2017b). Thoughts on history, tuning and the scholarship of teaching and learning in the United States. *Arts and Humanities in Higher Education*, *16*(4), 415–419. https://doi.org/10.1177/1474022216686508

Parcell, E. S., & Rafferty, K. A. (2017). Interviews, recording and transcribing. In M. Allen (Ed.), *The SAGE Encyclopedia of Communication Research Methods*. Thousand Oaks: Sage Publications, Inc.

https://doi.org/10.4135/9781483381411.n275

Parham, J., Gugerty, L., & Stevenson, D. E. (2010). Empirical evidence for the existence and uses of metacognition in Computer Science problem solving. In *Proceedings of the 41st ACM technical symposium on Computer Science Education* (pp. 416–420). Milwaukee, Wisconsin: ACM. https://doi.org/10.1145/1734263.1734406

Patil, S. P., & Goje, A. C. (2009). The effect of developments in student attributes on success in programming of management students. In *Proceedings of the 2009 International Conference on Education Technology and Computer* (pp. 191–193). Singapore: IEEE Computer Society Press. https://doi.org/10.1109/ICETC.2009.35

Patton, M. Q. (2015). *Qualitative research & evaluation methods: Integrating theory and practice* (4th ed.). Thousand Oaks: Sage Publications.

Pennington, N. (1987a). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 100–113). Norwood: Ablex Publishing Corporation.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

Perscheid, M. (2011). Dynamic service analysis : Test-driven views for enhancing software maintenance. In J. Holzl, L. Ribe-Baumann, & M. Bruckner (Eds.), *Joint Workshop of the German Research Training Groups in Computer Science* (p. 137). Berlin: GITO mbH Verlag.

Phillips, A. (2019). The quest for equity in higher education. *Pepperdine Policy Review*, *11*(4), 1–28.

Pillay, N., & Jugoo, V. R. (2005). An investigation into student characteristics affecting novice programming performance. *ACM SIGCSE Bulletin*, *37*(4), 107–110. https://doi.org/10.1145/1113847.1113888

Pintrich, P. R. (1999). The role of motivation in promoting and sustaining self-regulated learning. *International Journal of Educational Research*, *31*(6), 459–470. https://doi.org/10.1016/S0883-0355(99)00015-4

Piteira, M., & Costa, C. (2013). Learning computer programming: Study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication* (pp. 75–80). Lisboa,

Portugal: ACM. https://doi.org/10.1145/2503859.2503871

Plowright, D. (2011). *Using mixed methods: Frameworks for an integrated methodology*. London: Sage Publications.

Plowright, D. (2016a). Developing doctoral research skills for workplace inquiry. In M. Fourie-Malherbe, R. Albertyn, & E. Bitzer (Eds.), *Postgraduate Supervision-Future Foci for the knowledge society* (pp. 241–254). Stellenbosch: Sun Press. https://doi.org/10.18820/9781928357223/14

Plowright, D. (2016b). Making sense of research in higher education. In L. Frick, V. Trafford, & M. Fourie-Malherbe (Eds.), *Being Scholarly - Festschrift in honour of the work of Eli M Bitzer* (pp. 15–24). Stellenbosch: SUN MeDIA. https://doi.org/10.18820/9781928314219/01

Pope, C., Izu, C., Weerasinghe, A., & Pope, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 251–259). Melbourne, VIC: Association for Computing Machinery. https://doi.org/10.1145/2960310.2960324

Powell, N., Moore, D., Gray, J., Finlay, J., & Reaney, J. (2004). Dyslexia and learning computer programming. *ACM SIGCSE Bulletin*, *36*(3), 242. https://doi.org/10.1145/1026487.1008072

Powers, W. R. (2005). *Transcription techniques for the spoken word*. New York: Altamira Press.

Praveen, A. (2016). Program comprehension and analysis. *International Journal of Engineering and Applied Computer Science*, *01*(01), 17–21. https://doi.org/10.24032/ijeacs/0101/04

Preece, J., Rogers, Y., & Sharp, H. (2015). *Interaction design: Beyond human-computer interaction* (4th ed.). New Delhi: John Wiley & Sons, Inc.

Ratey, J. J. (2001). *A user's guide to the brain: Perception, attention, and the four theatres of the brain*. New York: Pantheon.

Raymond, E. (2017). *Learners with mild disabilities: A characteristic approach* (5th ed.). Massachusetts: Allyn & Bacon.

Reed, D., Miller, C., & Braught, G. (2000). Empirical investigation throughout the CS curriculum. *ACM SIGCSE Bulletin*, *32*(1), 202–206. https://doi.org/10.1145/331795.331855

Rist, R. S. (1986). Plans in programming: Definition, demonstration and

development. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 28–45). New Jersey: Ablex Publishing Corporation.

Robin, M. (2002). *A physiological handbook for teachers of Yogasana*. Tucson: Fenestra Books.

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, *20*(1), 37–71. https://doi.org/10.1080/08993401003612167

Roehler, L. R., & Cantlon, D. J. (1997). Scaffolding: A powerful tool in social constructivist classrooms. In K. Hogan & M. Pressley (Eds.), *Scaffolding student learning: Instructional approaches and issues* (pp. 6–42). Cambridge: Brookline Books.

Sagor, R. (2000). *Guiding school improvement with action research*. Alexandria: Association for Supervision and Curriculum Development.

Saha, B., & Ray, U. K. (2015). Learning programming : An Indian perspective. *International Journal of Information Science and Computing*, *2*(1), 21–32.

Sanders, K., & Mccartney, R. (2016). Threshold concepts in computing : Past, present, and future. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 91–100). Koli, Finland: ACM.

Sarkar, A. (2015). 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). Bournemouth, UK: Psychology of Programming Interest Group.

Sarpong, K. A., Arthur, J. K., & Amoako, P. (2013). Causes of failure of students in computer programming courses: The teacher – learner perspective. *International Journal of Computer Applications*, *77*(12), 27–32. https://doi.org/10.5120/13448-1311

Schlinger, H. D. (1995). *A behavior analytic view of child development*. New York: Springer Science & Business Media.

Schmidt, A. L. (1986). Effects of experience and comprehension on reading time. *International Journal of Man-Machine Studies*, 399–409.

Schwandt, T. A., Lincoln, Y. S., & Guba, E. G. (2007). Judging interpretations: But is it rigorous? Trustworthiness and authenticity in naturalistic evaluation. *New Directions for Evaluation*, *2007*(114), 11–25. https://doi.org/10.1002/ev

Sengupta, N., Bhattacharya, M. S., & Sengupta, R. N. (2012). *Managing change in organizations* (3rd ed.). New Delhi: PHI Learning Private Limited.

Sentance, S., & Csizmadia, A. (2016). Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies*, *22*, 469–495. https://doi.org/10.1007/s10639-016-9482-0

Shaft, T. M. (1995). Helping programmers understand computer programs: The use of metacognition. *Data Base Advances*, *26*(4), 25–46. https://doi.org/10.1145/223278.223280

Shaft, T. M., & Vessey, I. (1995). Research report - The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research*, *6*(3), 286–299. https://doi.org/10.1287/isre.6.3.286

Sheard, J., D'Souza, D., Lopez, M., Luxton-Reilly, A., Robbins, P., Teague, D., & Whalley, J. L. (2015). How (not) to write an introductory programming exam. In *Proceedings of the 17th Australasian Computing Education Conference (ACE 2015)* (pp. 137–146). Sydney, Australia: Australian Computer Society, Inc.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, *5*(2), 123–143. https://doi.org/10.1007/BF00975629

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems*. Massachusetts: Winthrop Publishers, Inc.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, *8*(3), 219–238. https://doi.org/10.1007/BF00977789

Shopkow, L. (2017). How many sources do I need? *The History Teacher*, *50*(2), 169–200.

Shopkow, Leah, Diaz, A., Middendorf, J., Pace, D., Díaz, A., Middendorf, J., & Pace, D. (2013). The History learning project "decodes" a discipline: The union of teaching and epistemology. In K. McKinney (Ed.), *Scholarship of Teaching and Learning in and Across the Disciplines*. Bloomington: Indiana University Press.

Shulman, L. S. (1986). Those who understand : Knowledge growth in teaching. *Educational Researcher*, *15*(2), 4–14. https://doi.org/http://www.jstor.org/stable/1175860

Siegmund, J. (2016). Program comprehension : Past, present, and future. In *Proceedings of the 23rd International Conference on Software Analysis,*

Evolution, and Reengineering (pp. 13–20). Suita, Japan: IEEE. https://doi.org/10.1109/SANER.2016.35

Siegmund, J., Kástner, C., Apel, S., Brechmann, A., & Saake, G. (2013). Experience from measuring program comprehension - Toward a general framework. In S. Kowalewski & B. Rumpe (Eds.), *Software Engineering 2013 - Fachtagung des GI-Fachbereichs Softwaretechnik. GI-Edition - Lecture Notes in Informatics (LNI)* (Vol. P-213, pp. 239–257). Bonn: Gesellschaft für Informatik e.V.

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., … Brechmann, A. (2014). Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 378–389). Hyderabad, India: ACM. https://doi.org/10.1145/2568225.2568252

Sillito, J., De Volder, K., Fisher, B., & Murphy, G. (2005). Managing software change tasks: an exploratory study. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering, ISESE'05* (pp. 23–32). Noosa Heads, Qld., Australia: IEEE. https://doi.org/10.1109/ISESE.2005.1541811

Simons, R., & Bolhuis, S. (2004). Constructivist learning theories and complex learning environments. *Oxford Studies in Comparative Education*, *13*(1), 13–25.

Singer, J., Lethbridge, T., Vinson, N., & Anquetil, N. (1997). An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research* (pp. 209–223). Toronto, Ontario: IBM Press. https://doi.org/10.1145/782010.782031

Singer, L. (2013). *Improving the adoption of software engineering practices through persuasive interventions*. PhD thesis, Fakultät für Elektrotechnik und Informatik, Gottfried Wilhelm Leibniz Universität Hannover.

Singh, V., Pollock, L. L., Snipes, W., & Kraft, N. A. (2016). A case study of program comprehension effort and technical debt estimations. In *Proceedings of the 24th International Conference on Program Comprehension (ICPC)* (pp. 1–9). Austin, TX: IEEE. https://doi.org/10.1109/ICPC.2016.7503710

Smith, K., & Davies, J. (2010). Qualitative data analysis. In L. Dahlberg & C. McCaig (Eds.), *Practical researcher and evaluation: A start-to finish guide for practitioners* (pp. 145–158). London: Sage Publications.

Soh, Z., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2013). Towards understanding how developers spend their effort during maintenance activities.

In *Proceedings of the Working Conference on Reverse Engineering (WCRE)* (pp. 152–161). Koblenz, Germany: IEEE. https://doi.org/10.1109/WCRE.2013.6671290

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858. https://doi.org/10.1145/6592.6594

Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glasser, & M. J. Farr (Eds.), *The nature of expertise* (pp. 129–152). Hillsdale: Lawrence Erlbaum Associates.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Soloway, E., Ehrlich, K., & Black, J. B. (1983). Beyond numbers: Don't ask "how many" ... ask "why." In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 240–246). Boston, Massachusetts: ACM. https://doi.org/10.1145/800045.801619

Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing* (pp. 52–57). Gaithersburg, Maryland: ACM. https://doi.org/10.1145/800049.801754

Soloway, E., Lampert, R., Letovsky, S., Littman, D., & Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM*, *31*(11), 1259–1267. https://doi.org/10.1145/50087.50088

Soloway, E., Lochhead, J., & Clement, J. (1982). Does computer programming enhance problem solving ability? Some positive evidence on Algebra word problems. In R. J. Sediel, R. E. Anderson, & B. Hunter (Eds.), *Computer Literacy* (pp. 171–201). New York: Academic Press, Inc. https://doi.org/10.1016/b978-0-12-634960-3.50023-3

Soloway, E., & Spohrer, J. C. (2013). *Studying the novice programmer*. New York: Psychology Press.

Sousa, D. A. (2006). *How the brain learns* (3rd ed.). California: Thousand Oaks.

Standish, T. A. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering*, *SE-10*(5), 494–497. https://doi.org/10.1109/TSE.1984.5010272

Storey, M. A. D., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements

to support the construction of a mental model during software visualization. *The Journal of Systems and Software*, *44*, 171–185. https://doi.org/10.1109/WPC.1997.601257

Storey, M. A. D., Wong, K., & Müller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, *36*(2), 183–207. https://doi.org/10.1016/S0167-6423(99)00036-2

Stringer, E. T. (2014). *Action research* (4th ed.). California: Sage Publications.

Sweller, J. (1988). Cognitive load during problem solving : Effects on learning. *Cognitive Science*, *12*, 257–285. https://doi.org/10.1016/0364-0213(88)90023-7

Sweller, J., Van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296. https://doi.org/10.1023/A:1022193728205

Teasley, B. M. (1993). Program comprehension skills and their acquisition: A call for an ecological paradigm. In E. Lemut, B. Du Boulay, & G. Dettori (Eds.), *Cognitive models and intelligent environments for learning programming* (pp. 71–79). New York: Springer-Verlag Berlin Heidelberg.

Teddlie, C., & Tashakkori, A. (2009). *Foundations of mixed methods research: Integrating quantitative and quantitative approaches in the social and behavioral sciences*. New York: Sage Publications.

ten Have, P. (2011). Transcribing talk-in-interaction. In P. ten Have (Ed.), *Introducing qualitative methods: doing conversation analysis* (pp. 94–115). London: Sage Publications, Ltd. https://doi.org/10.4135/9781849208895

The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Computer Society. (2013). *Computer Science curricula 2013: Curriculum guidelines for undergraduate degree programs in Computer Science*. New York: Association for Computing Machinery. https://doi.org/10.1145/2534860

Thorne, S. (2000). Data analysis in qualitative research. *Evidence-Based Nursing*, *3*(3), 68–70. https://doi.org/10.1136/ebn.3.3.68

Thurner, V., Zehetmeier, D., Hammer, S., & Böttcher, A. (2017). Developing a test for assessing incoming students' cognitive competences. *International Journal of Engineering Pedagogy*, *7*(4), 35. https://doi.org/10.3991/ijep.v7i4.7433

Tiarks, R. (2011). What maintenance programmers really do: An observational study. *Softwaretechnik-Trends*, *31*(2), 1–2.

Timmermans, J. A., & Meyer, J. H. F. (2019). A framework for working with university teachers to create and embed 'Integrated Threshold Concept Knowledge' (ITCK) in their practice. *International Journal for Academic Development*, *24*(4), 354–368. https://doi.org/10.1080/1360144X.2017.1388241

Tingerthal, J. (2013). *Applying the decoding the disciplines process to teaching structural mechanics: An autoethnographic case study*. PhD theisis, Northern Arizona University.

Tobias, S. (1992-1993). Disciplinary cultures and general education: What can we learn from our learners? *Teaching Excellence*, *4*(6), 1–3.

Trochim, W. M. K. (2006). Research methods knowledge base, 2nd edition. Retrieved April 2, 2019, from https://socialresearchmethods.net/kb/

Tucker, V. M. (2017). Threshold concepts and core competences in the library and information science (LIS) domain: Methodologies for discovery. *Library and Information Research*, *41*(125), 61–80. https://doi.org/10.29173/lirg750

Ulin, P. R., Robinson, E. T., & Tolley, E. E. (2005). *Qualitative methods in public health: A field guide for applied research*. San Francisco, CA: Jossey-Bass.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., … Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on innovation and technology in Computer Science education-working group reports* (pp. 15–32). Canterbury, England: ACM. https://doi.org/10.1145/2543882.2543884

Uzonwanne, F. C. (2016). Rational model of decision making. In A. Farazmand (Ed.), *Global Encyclopedia of Public Administration, Public Policy, and Governance* (pp. 1–6). Cham: Springer International Publishing AG. https://doi.org/10.1007/978-3-319-31816-5_2474-1

Van den Broeck, J., Cunningham, S. A., Eeckels, R., & Herbst, K. (2005). Data cleaning: Detecting, diagnosing, and editing data abnormalities. *PLoS Medicine*, *2*(10), 0966–0970. https://doi.org/10.1371/journal.pmed.0020267

Van Gorp, M. J., & Grissom, S. (2001). An empirical evaluation of using constructive classroom activities to teach introductory programming. *Computer Science Education*, *11*(3), 247–260. https://doi.org/10.1076/csed.11.3.247.3837

Van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. (1994). *The think aloud method: A practical guide to modelling cognitive processes* (1st ed.). London:

Academic Press.

Van Teijlingen, E., & Hundley, V. (2002). The importance of pilot studies. *Nursing Standard*, *16*(40), 33–36. https://doi.org/10.7748/ns2002.06.16.40.33.c3214

Van Teijlingen, E. R., Rennie, A. M., Hundley, V., & Graham, W. (2001). The importance of conducting and reporting pilot studies: The example of the Scottish Births Survey. *Journal of Advanced Nursing*, *34*(3), 289–295. https://doi.org/10.1046/j.1365-2648.2001.01757.x

Veerasamy, A. K., D'Souza, D., Lindén, R., & Laakso, M. J. (2018). The impact of prior programming knowledge on lecture attendance and final exam. *Journal of Educational Computing Research*, *56*(2), 226–253. https://doi.org/10.1177/0735633117707695

Verpoorten, D., Devyver, J., Duchâteau, D., Mihaylov, B., Agnello, A., Ebrahimbabaye, P., & Focant, J. (2017). Decoding the disciplines – A pilot study at the University of Liège (Belgium). In *Proceedings of the 2nd EuroSoTL conference - Transforming patterns through the scholarship of teaching and learning* (pp. 263–267). Lund, Sweden: Lund University Press.

Vogel, S., & Draper-Rodi, J. (2017). The importance of pilot studies, how to write them and what they mean. *International Journal of Osteopathic Medicine*, *23*, 2–3. https://doi.org/10.1016/j.ijosm.2017.02.001

Von Mayrhauser, A., & Vans, A. M. (1993). From program comprehension to tool requirements for an industrial environment. In *Proceedings of the Second Workshop on Program Comprehension* (pp. 78–86). Capri, Italy: IEEE Computer Society Press. https://doi.org/10.1109/WPC.1993.263903

Von Mayrhauser, A., & Vans, A. M. (1995a). Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, *10*(5), 171–182. https://doi.org/10.1049/sej.1995.0023

Von Mayrhauser, A., & Vans, A. M. (1995b). Program comprehension during software maintenance and evolution. *IEEE Computer*, *28*(8), 44–55. https://doi.org/10.1109/2.402076

Von Mayrhauser, A., & Vans, A. M. (1996). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, *22*(6), 424–437.

Von Mayrhauser, A., & Vans, A. M. (1997). Program understanding behavior during debugging of large scale software. In *Proceedings of the seventh workshop on*

Empirical studies of programmers (pp. 157–179). Alexandria, VA: ACM. https://doi.org/10.1145/266399.266414

Von Mayrhauser, A., Vans, A. M., & Howe, A. E. (1997). Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, *9*, 299–327. https://doi.org/10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S

Von Mayrhauser, A., Vans, A. M., & Lang, S. (1998). Program comprehension and enhancement of software. In *Proceedings of the 15th IFIP World Computing Congress - Information Technology and Knowledge Systems*. Vienna, Austria: Austrian Computer Society, Inc.

Vygotsky, L. S., & Cole, M. (1978). *Mind in society: The development of higher psychological processes*. Massachusetts: Harvard University Press.

Wallendorf, M., & Belk, R. W. (1989). Assessing trustworthiness in naturalistic consumer research. In E. C. Hirschman (Ed.), *Interpretive consumer research* (pp. 69–84). Provo, UT: Association for Consumer Research.

Waugh, N. C., & Norman, D. A. (1965). Primary memory. *Psychological Review*, *72*(2), 89–104. https://doi.org/10.1037/h0021797

Whalley, J., & Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in Computer Science Education* (pp. 279–284). Uppsala, Sweden: ACM. https://doi.org/10.1145/2591708.2591762

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). 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 - Volume 2* (pp. 243–252). Hobart, Australia: Australian Computer Society, Inc.

Widowski, D., & Eyferth, K. (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H. Willumeit (Ed.), *Human Decision Making and Manual Control*. North Holland: Elsevier Inc.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, *23*(4), 383–390. https://doi.org/10.1016/S0020-7373(85)80041-9

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, *39*(5), 793–812. https://doi.org/10.1006/imms.1993.1084

Wiedenbeck, S., LaBelle, D., & Kain, V. (2004). Factors affecting course outcomes in introductory programming. In E. Dunican & T. R. G. Green (Eds.), *Proceedings of the 16th Workshop of the Psychology of Programming Interest Group* (pp. 97–110). Carlow, Ireland: PPIG.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*(3), 255–282. https://doi.org/10.1016/S0953-5438(98)00029-0

Wills, C. E., Deremer, D., McCauley, R. A., & Null, L. (2010). Studying the use of peer learning in the introductory Computer Science curriculum. *Computer Science Education*, *9*(2), 71–88. https://doi.org/10.1076/csed.9.2.71.3811

Wilson, B. C. (2002). A study of factors promoting success in Computer Science including gender differences. *Computer Science Education*, *12*(1–2), 141–164. https://doi.org/10.1076/csed.12.1.141.8211

Wlodkowski, R. J., & Ginsberg, M. B. (2010). *Teaching intensive and accelerated courses : Instruction that motivates learning*. San Francisco: Jossey-Bass.

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x

Xie, B., Nelson, G. L., & Ko, A. J. (2018). An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 344–349). New York, NY: Association for Computing Machinery. https://doi.org/10.1145/3159450.3159527

Appendix A – Questionnaire for Senior Students (Phase 1)

Dear Senior Student,

Thank you for giving your attention to this questionnaire. The approximate time needed to complete this questionnaire is 60 - 90 minutes. The purpose of this questionnaire is to uncover source code comprehension challenges experienced by students.

By completing this questionnaire, you give the researcher consent to use your information for research purposes only. Responses will be confidential and your privacy will be protected to the maximum extent allowable by law.

Participation is voluntary and you can withdraw anytime when you do not feel like participating anymore. Completing or failing to complete this questionnaire has absolutely no bearing on your grade for any of the Computer Science modules you are enrolled into.

Section 1 – Source Code Comprehension Questions

Instructions:

Carefully study the following fragments of source code, and use the working spaces provided for each question to show your workings. Please write your answer to each question in the space provided at the end of each question.

Question 1
Consider the following source code fragment:

```
int[] x = { 2, 1, 4, 5, 7 };
int limit = 3;
int i = 0;
int sum = 0;
while ((sum < limit) && (i < x.Length))
{
     ++i;
     sum += x[i];
}
```

What value is in the variable **`i`** after this source code is executed?

a) 0
b) 1
c) 2
d) 3

Answer to Question 1: _____

Question 2
Consider the following source code fragment:

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.Length - 1;
int i2 = x2.Length - 1;
int count = 0;
while ((i1 > 0) && (i2 > 0))
{
   if (x1[i1] == x2[i2])
   {
      ++count;
      --i1;
      --i2;
   }

   else if (x1[i1] < x2[i2])
   {
      --i2;
   }

   else
   {
     // x1[i1] > x2[i2]
     --i1;
   }
}
```

After the above while loop finishes, **`count`** contains what value?
a) 3
b) 2
c) 1
d) 0

Answer to Question 2: _____

Question 3
Consider the following source code fragment:

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

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

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

int count = 0;

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

After this source code is executed, **`count`** contains:
a) 1
b) 2
c) 3
d) 4
e) 5

Answer to Question 3: _____

Question 4
Consider the following source code fragment:

```
int[] x1 = {0, 1, 2, 3};
int[] x2 = {1, 2, 2, 3};
int i1 = 0; int i2 = 0;
int count = 0;

while ((i1 < x1.Length) && (i2 < x2.Length))
{
  if (x1[i1] == x2[i2])
  {
    ++count;
    ++i2;
  }

  else if (x1[i1] < x2[i2])
  {
    ++i1;
  }

  else
  {

   // x1[i1] > x2[i2]
   ++i2;
  }
}
```

After this source code is executed, **`count`** contains:
a) 0
b) 1
c) 2
d) 3
e) 4

Answer to Question 4: _____

Question 5
Consider the following source code fragment:

```
int[] x = { 0, 1, 2, 3 };
int temp;
int i = 0;
int j = x.Length - 1;

while (i < j)
{
     temp = x[i];
     x[i] = x[j];
     x[j] = 2*temp;
     i++;
     j--;
}
```

After this source code is executed, array **`x`** contains the values:

a) { 3, 2, 2, 0 }
b) { 0, 1, 2, 3 }
c) { 3, 2, 1, 0 }
d) { 0, 2, 4, 6 }
e) { 6, 4, 2, 0 }

Answer to Question 5: _____

Question 6

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

```
public static bool isSorted (int[] x)
{
//missing source code goes here
}
```

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

a)
```
bool b = true;
for (int i = 0; i < x.Length - 1; i++)
{
 if (x[i] > x[i + 1])
    b = false;
 else
    b = true;
}
return b;
```

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

c)
```
bool b = false;
for (int i = 0; i < x.Length - 1; i++)
{
     if (x[i] > x[i + 1])
     b = false;
}
return b;
```

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

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

Answer to Question 6: _____

Question 7
Consider the following source code fragment:

```
int[] x = { 2, 1, 4, 5, 7 };
int limit = 7;
int i = 0;
int sum = 0;

while ((sum < limit) && (i < x.Length))
{
     sum += x[i];
     ++i;
}
```

What value is in the variable **`i`** after this source code is executed?
a) 0
b) 1
c) 2
d) 3
e) 4

Answer to Question 7: _____

Question 8
If any two numbers in an array of integers, not necessarily consecutive numbers in the array, are out of order (i.e. the number that occurs first in the array is larger than the number that occurs second), then that is called an inversion. For example, consider an array **x** that contains the following six numbers:

```
  4 5 6 2 1 3
```

There are 10 inversions in that array, as:

```
x[0]=4 > x[3]=2
x[0]=4 > x[4]=1
x[0]=4 > x[5]=3
x[1]=5 > x[3]=2
x[1]=5 > x[4]=1
x[1]=5 > x[5]=3
x[2]=6 > x[3]=2
x[2]=6 > x[4]=1
x[2]=6 > x[5]=3
x[3]=2 > x[4]=1
```

The skeleton source code below is intended to count the number of inversions in an array **x:**

```
int inversionCount = 0;

for (int i = 0; i < x.Length - 1; i++)
{
     for xxxxxx
     {
          if (x[i] > x[j])
          ++inversionCount;
     }
}
```

When the above source code finishes, the variable **inversionCount** is intended to contain the number of inversions in array **x**. Therefore, the **xxxxxx** in the above source code should be replaced by:

a) `(int j = 0; j < x.Length; j++)`
b) `(int j = 0; j < x.Length - 1; j++)`
c) `(int j = i + 1; j < x.Length; j++)`
d) `(int j = i + 1; j < x.Length - 1; j++)`

Answer to Question 8: _____

Question 9
The skeleton source code below is intended to copy into an array of integers called `array2` any numbers in another integer array `array1` that are even numbers. For example, if `array1` contained the numbers:

```
array1: 4 5 6 2 1 3
```

then after the copying process, `array2` should contain in its first three places:

```
array2: 4 6 2
```

The following source code assumes that `array2` is big enough to hold all the even numbers from `array1`:

```
int a2 = 0;

for (int a1 = 0; xxx1xxx; ++a1)
{
     // if array1[a1] is even
     if (array1[a1] % 2 == 0)
     {
          // array1[a1] is even,
          // so copy it
          xxx2xxx;
          xxx3xxx;
     }
}
```

The missing pieces of source code `xxx1xxx`, `xxx2xxx` and `xxx3xxx` in the above source code should be replaced respectively by:

a)
```
a1 < array1.Length
++a2
array2[a2] = array1[a1]
```

b)
```
a1 < array1.Length
array2[a2] = array1[a1]
++a2
```

c)
```
a1 <= array1.Length
array2[a2] = array1[a1]
++a2
```

d)
```
a1 <= array1.Length
++a2
array2[a2] = array1[a1]
```

Hint: in all four options above, the second and third parts are the same, just reversed.

Answer to Question 9: _____

Question 10
Consider the following source code fragment:

```
int[] array1 = { 2, 4, 1, 3 };
int[] array2 = { 0, 0, 0, 0 };
int a2 = 0;

for (int a1 = 1; a1 < array1.Length; ++a1)
{
     if (array1[a1] >= 2)
     {
          array2[a2] = array1[a1];
          ++a2;
     }
}
```

After this source code is executed, the array `array2` contains what values?
a) `{ 4, 3, 0, 0 }`
b) `{ 4, 1, 3, 0 }`
c) `{ 2, 4, 3, 0 }`
d) `{ 2, 4, 1, 3 }`

Answer to Question 10: _____

Question 11
Suppose an array of integers `s` contains zero or more different positive integers, in ascending order, followed by a zero. For example:

```
int[] s = { 2, 4, 6, 8, 0 };
```
or
```
int[] s = { 0 };
```

Consider the following "skeleton" source code, where the sequences of `xxxxxx` are substitutes for the correct C# source code:

```
int pos = 0;
while ( (xxxxxx) && (xxxxxx) )
   ++pos;
```

Suppose an integer variable **`e`** contains a positive integer. The purpose of the above source code is to find the place in `s` occupied by the value stored in `e`. Formally, when the above `while` loop terminates, the variable `pos` is determined as follows:

1. If the value stored in **`e`** is also stored in the array, then `pos` contains the index of that position. For example, `if e=6 and s = {2, 4, 6, 8, 0},` then `pos` should equal `2`.

2. If the value stored in **`e`** is NOT stored in the array, but the value in **`e`** is less than some of the values in the array then `pos` contains the index of the lowest position in the array where the value is larger than in **`e`**. For example, `if e=7 and s = {2, 4, 6, 8, 0}`, then `pos` should equal **`3`**.

3. If the value stored in **`e`** is larger than any value in **`s`**, then `pos` contains the index of the position containing the zero. For example, `if e=9 and s = {2, 4, 6, 8, 0},` then `pos` should equal `4`.

The correct Boolean condition for the above `while` loop is:
a) `(pos < e) && (s[pos] != 0)`
b) `(pos != e) && (s[pos] != 0)`
c) `(s[pos] < e) && ( pos != 0)`
d) `(s[pos] < e) && (s[pos] != 0)`
e) `(s[pos] != e) && (s[pos] != 0)`

Answer to Question 11: _____

Question 12
This question continues on from the previous question. Assuming we have found the position in the array `s` containing the same value stored in the variable **`e`**, we now wish to write source code that deletes that number from the array, but retains the ascending order of all remaining integers in the array. For example, given:

```
s = { 2, 4, 6, 8, 0 }; e = 6; pos = 2;
```

The desired outcome is to remove the `6` from **`s`** to give:

```
s = { 2, 4, 8, 0, 0 };
```

Consider the following "skeleton" source code, where `xxxxxx` is a substitute for the correct C# source code:

```
do {
++pos;
xxxxxx;
} while (s[pos] != 0 );
```

The correct replacement for `xxxxxx` is:

a) `s[pos+1] = s[pos];`
b) `s[pos] = s[pos+1];`
c) `s[pos] = s[pos-1];`
d) `s[pos-1] = s[pos];`
e) `None of the above`

Answer to Question 12: _____

Section 2 - Demographic information

Instructions:

Please circle your selected answers.

Tell us a little bit about yourself:

Gender: Male Female

Age: 18 19 20 21 22 23+

Student number: _____

Thank you for completing the questionnaire!

Appendix B – Aggregate Performance of Phase 1 participants

The following table shows an aggregate performance of Phase 1 participants (3rd Year students who answered the 12 original MCQs developed and used in the multi-national study by Lister et al. (2004, pp. 141-144). The five columns in this table can be described thus:

- Column1 – Question number as presented in the original Lister et al.'s (2004) paper.
- Column 2 – Percentage (%) of students who answered each question correctly.
- Column 3 – Number of students who answered each question correctly.
- Column 4 – Ranking of the questions according to the performance of this study's participants (Rank 1 was the hardest question, while Rank 12 was the easiest).
- Column 5 – Ranking of the questions according to the performance of Lister et al.'s (2004) participants.

MCQ	% correct	Number of students	Question ranking (Phase 2 of this study)	Question ranking (2004 ITiCSE working group)
1	61	23	(11/12) Easiest	8/9
2	38	14	4/5/6	6
3	38	14	4/5/6	7
4	49	18	9	5
5	41	15	7	12(Easiest)
6	30	11	3	2
7	51	19	10	10
8	22	8	2	3
9	62	23	(11/12) Easiest	11
10	46	17	8	8/9
11	38	14	4/5/6	4
12	19	7	1 (Hardest)	1 (Hardest)

Appendix C – Invitation Letter to Senior Students (Phase 2)

21 May 2018

Dear [**Student Initials and Surname**],

On Monday, 23 April 2018, you completed a short test on source code comprehension. The test was part of the first phase of a research study looking to uncover the source code comprehension challenges of novice programmers. Based on your performance in the Phase 1 test, you have been identified as a suitable participant for Phase 2 of the study.

We would, therefore, like to extend an invitation to you to participate in Phase 2 of this study. As a participant in Phase 2, you will be asked to answer three source code comprehension questions in a think-aloud manner – explaining your reasoning as your work through each of the questions. The session will last for approximately 30 minutes. In appreciation for your time, you will receive a R100 voucher for Treats on the Thakaneng Bridge.

Sessions will be scheduled in the time period from 05 to 29 June 2018. If you are willing to participate, please contact Mr Pakiso Khomokhoana (E-mail: khomo_khoana@yahoo.com or Physical: WWG308) to book your session.

Please feel free to contact me if you need additional information about this study.

Kind regards,

Prof Liezel Nel
Adjunct-professor: Department of Computer Science & Informatics

Appendix D – Case Study Protocol for Senior Students

(Phase 2)

Protocol to identify and determine the nature of relevant bottlenecks related to source code comprehension

Case study protocol introduction:

Thank you for agreeing to participate in this research activity. Prior to the activity you were sent a participant information sheet and two consent forms (one to sign and return; and another to keep). This activity will not last for longer than 30 minutes. However, you can take as much time as you like if you feel so. In answering the questions, please use the think-aloud protocol, that is talk to us and to yourself aloud as much as you can. Also scribble on your working paper as much as you can; and try to demonstrate your thinking as much as possible so that we are able to understand how and why you arrived at your answer. The main issue is not necessarily to test whether you can get the answers correct or not, but to analyse your thinking in the process. For purposes of capturing all occurrences of the experiment proceedings, the session will be audio-recorded. I hope you have read the participant information sheet sent out to you earlier. Do you have any questions on it or any other questions relating to the study? If there are no further questions, let us get started with the questions.

Participant instructions:

Answer the three source code comprehension questions. Use think-aloud to verbalise your reasoning in answering these questions.

Post-experiment questions

- What did you like the most about the questions?
- What do you think was the most challenging in the questions?
- How did you find hand analysing, working through and determining the output of the source code/aim or what the source code does (*hand executing source code?*) Please describe.
    - Easy? Difficult? Challenging?
- Any other comments?
    - Language?

Appendix E – Decoding Interview Protocol

(Phase 3 - Experts)

Interview Introduction

Thank you for agreeing to participate in this interview. We estimate that this interview will be approximately 60 minutes in length. Prior to the interview you were sent a participant information sheet and two consent forms (one to sign and return; and another to keep). Have you read the participant information sheet sent out to you earlier? Do you have any questions on it or any other questions relating to the study?

The ultimate aim of this study is to help novice programmers to improve their source code comprehension skills. As such, we want them to be able to overcome bottlenecks associated with source code comprehension. A bottleneck is something that an instructor understands well, but despite his/her best efforts to teach it to students, they (the students) still struggle to understand it. Essentially, the instructor fails to understand why students do not understand it. One possible explanation is that instructors might perform certain actions 'automatically' and consequently fail to make students aware of these actions or steps they follow to perform the actions. We are therefore trying to uncover these 'hidden' steps so that they can be explicitly modelled and taught to students.

In this study, we are specifically focusing on source code tracing as a bottleneck. Preliminary findings of this study suggest that even senior undergraduate students are unable to reliably read/work through a section of source code in order to predict the correct output (*tracing*). Do you have any questions regarding the bottleneck description?

Ok, let us get started.

In this interview, we will ask you a number of questions based on the steps that you follow or the mental/intellectual/cognitive moves you make in tracing through source

code. Basically, we want to explore the techniques or processes or strategies or tactics that you use when you trace through source code. Also note that you are used to being able to answer questions put to you as an expert. But in this interview, as we explore your tacit knowledge, you may find it difficult to answer some of our questions. That will be a good sign and will mean the interview will be going well, so do not worry if it happens. It is also possible that we can ask you the same question more than once. If it happens, we may want you to go deeper or give more details or provide more clarification. This is a normal process in this type of interview.

Ok, suppose you are presented with a piece of source code on a piece of paper and asked to read/work through it to predict its output (tracing). Can you explain to us how you would go about doing that?

…..

Assuming you are given the following question, how would you go about answering it? (*Researcher handing over the question to the participant*).

Possible probing questions

- What are the questions that come to your mind immediately when you see this problem?
- When faced with a similar problem, what do you do?
- How do you know what to pay attention to in the problem?
- What would students have to do to understand or overcome that?
- Is there any experience one needs to tackle that problem?
- Are there any particular concepts one has to focus on? Why?
- How would you make connections between these concepts?
- What do you think enables other students to do it correctly while others cannot?
- How do you know when you have found an answer? (i.e. when to stop?)

Appendix F – Ethical Clearance Approval

UNIVERSITY OF THE FREE STATE
UNIVERSITEIT VAN DIE VRYSTAAT
YUNIVESITHI YA FREISTATA

UFS·UV
NATURAL AND AGRICULTURAL SCIENCES
NATUUR- EN LANDBOUWETENSKAPPE

Faculty of Natural and Agricultural Sciences

20-Mar-2018

Dear **Mr Pakiso Khomokhoana**

Ethics Clearance: **Source code comprehension: Uncovering the cognitive challenges of novice programmers**

Principal Investigator: **Mr Pakiso Khomokhoana**

Department: **Computer Science and Informatics (Bloemfontein Campus)**

APPLICATION APPROVED

This letter confirms that a research proposal with tracking number: **UFS-HSD2018/0038** and title: **'Source code comprehension: Uncovering the cognitive challenges of novice programmers'** was given ethical clearance by the Ethics Committee.

Your ethical clearance number, to be used in all correspondence is: **UFS-HSD2018/0038**

Please ensure that the Ethics Committee is notified should any substantive change(s) be made, for whatever reason, during the research process. This includes changes in investigators. Please also ensure that a brief report is submitted to the Ethics Committee on completion of the research.
The purpose of this report is to indicate whether or not the research was conducted successfully, if any aspects could not be completed, or if any problems arose that the Ethics Committee should be aware of.

Note:

1. This clearance is valid from the date on this letter to the time of completion of data collection.
2. Progress reports should be submitted annually unless otherwise specified.

Yours Sincerely

Prof. RR (Robert) Bragg
Chairperson: Ethics Committee
Faculty of Natural and Agricultural Sciences

Natural and Agricultural Sciences Research Ethics Committee
Office of the Dean: Natural and Agricultural Sciences
T: +27 (0)51 401 2322 | +27 (0)82733 2696 | E: smitham@ufs.ac.za
Biology Building, Ground Floor, Room 9 | P.O. Box/Posbus 339 (Internal Post Box G44) | Bloemfontein 9300 | South Africa
www.ufs.ac.za

Appendix G – Participant Information Sheet (Phase 2 - Senior Students)

Source code comprehension: Uncovering the cognitive challenges of novice programmers

Dear Sir/Madam,

I hereby invite you to participate in a research study on source code comprehension. This study is being conducted as part of the Ph.D. research project of Mr. Pakiso J. Khomokhoana under the supervision of Prof Liezel Nel.

Participation in this research study requires signed consent from participants. Before you complete the consent form, you need to understand why the research study is being conducted, what your participation would require as well as potential benefits and risks. The Research Ethics Committee of the Faculty of Natural and Agricultural Science, University of the Free State has approved this research study. This information sheet and the attached consent form are only part of the process of informed consent.

Please take time to read the following information carefully. Feel free to ask questions if anything is not clear or if you required more details about any aspect of the study.

1. **What is the aim of this study?**

The aim of this study is to use part of the seven-step Decoding the Disciplines (DtDs) framework to investigate how instructors can help CS programming students (especially novices) to improve their SCC skills.

2. **Why have I been invited to participate?**

You have been invited to participate because you are currently registered for or have successfully completed an advanced programming module.

3. What will I be asked to do?

You will be required to answer a select number of Multiple Choice Questions in an observed environment. All the questions will contain source code fragments that you will have to study carefully and ultimately determine the output of each if it were to be executed on the Integrated Development Environment (IDE). In answering the questions, you will be required to use the thinking aloud technique. Moreover, you will be required to explain to the experimenter (interviewer and/or observer) the steps that you will take and why you will take such steps or think in a certain way as you work toward your answers. The experimenter will prod you verbally to keep you talking when you become silent.

4. Are there any possible risks and/or benefits from participating in this study?

There are no known or anticipated risks as a result of participating in the study. Furthermore, participants will not receive any direct benefits for their participation.

5. What if I change my mind during or after the study?

If you decide at any time during the experiment session that you no longer wish to participate in the research activities, you may withdraw your consent without providing any explanation. The information collected from you up to the point when you withdraw will be retained and may be used for the study.

6. What happens to the information I provide?

All information associated with you will be kept in private. Only the researcher and the supervisor will have access to the information. In publishing any results from this study, you will not be identified unless you give me specific permission to do so. I may also share the data with other researchers so that they can check the accuracy of my conclusions. However, this can only be done when I am confident that your confidentiality is fully protected. Any local electronic data will be stored on secured computers where only the researcher can gain access to the data. All physical and electronic records containing information that can identify you will be destroyed one year after publication of the study results.

7. How will the results of the study be published?

Written findings will be published online or in print journals; and written and/or video reporting may be presented at local, provincial, national or international academic conferences. The objective will be to advance an understanding of the cognitive challenges of novice programmers with source code comprehension. Your identity will remain anonymous in all written presentations of data via pseudonyms and the reporting of aggregated results.

8. What if I have questions about this study?

Please feel free to contact the researcher or supervisor if you require further information about the study. If you have concerns or complaints about the conduct of this study, please contact either the supervisor or the Research Ethics Coordinator of the Department of Computer Science and Informatics at the University of the Free State.

Contact details:

- **Researcher:** khomo_khoana@yahoo.com or (+27) 060 620 7710
- **Supervisor:** nell@ufs.ac.za or (+27) 051 401 3591
- **Departmental Research Ethics Coordinator:** BeeldersTR@ufs.ac.za or (+27) 051 401 9320

9. How do I give my consent to participate?

Complete the attached consent form if you understand and agree to take part in this study. Please submit the completed consent form to the researcher. You may keep this information sheet for your own records.

Appendix H – Participant Information Sheet

(Phase 3 - Experts)

Source code comprehension: Uncovering the cognitive challenges of novice programmers

Dear Sir/Madam,

I hereby invite you to participate in a research study on source code comprehension. This study is being conducted as part of the Ph.D. research project of Mr. Pakiso J. Khomokhoana under the supervision of Prof Liezel Nel.

Participation in this research study requires signed consent from participants. Before you complete the consent form, you need to understand why the research study is being conducted, what your participation would require as well as potential benefits and risks. The Research Ethics Committee of the Faculty of Natural and Agricultural Science, University of the Free State has approved this research study. This information sheet and the attached consent form are only part of the process of informed consent.

Please take time to read the following information carefully. Feel free to ask questions if anything is not clear or if you required more details about any aspect of the study.

1. **What is the aim of this study?**

The aim of this study is to use part of the seven-step Decoding the Disciplines (DtDs) framework to investigate how instructors can help CS programming students (especially novices) to improve their SCC skills.

2. **Why have I been invited to participate?**

You have been invited to participate because you have experience in teaching a programming course and/or you are working in the programming industry.

3. What will I be asked to do?

You will be required to participate in a face-to-face interview. The duration of this interview will not be longer than 60 minutes. In this interview, you will be asked to explain in explicit details the steps that you would follow to solve a given source code comprehension related problem. For purposes of capturing all occurrences of the interview proceedings, the interview will be video/audio-taped.

4. Are there any possible risks and/or benefits from participating in this study?

There are no known or anticipated risks as a result of participating in the study. Furthermore, participants will not receive any direct benefits for their participation.

5. What if I change my mind during or after the study?

If you decide at any time during the interview session that you no longer wish to participate in the research activities, you may withdraw your consent without providing any explanation. The information collected from you up to the point when you withdraw will be retained and may be used for the study.

6. What happens to the information I provide?

All information associated with you will be kept in private. Only the researcher and the supervisor will have access to the information. In publishing any results from this study, you will not be identified unless you give me specific permission to do so. I may also share the data with other researchers so that they can check the accuracy of my conclusions. However, this can only be done when I am confident that your confidentiality is fully protected. Any local electronic data will be stored on secured computers where only the researcher can gain access to the data. All physical and electronic records containing information that can identify you will be destroyed one year after publication of the study results.

7. How will the results of the study be published?

Written findings will be published online or in print journals; and written and/or video reporting may be presented at local, provincial, national or international academic conferences. The objective will be to advance an understanding of the cognitive challenges of novice programmers with source code comprehension. Your identity will

remain anonymous in all written and visual presentations of data via pseudonyms and the reporting of aggregated results.

8. What if I have questions about this study?

Please feel free to contact the researcher or supervisor if you require further information about the study. If you have concerns or complaints about the conduct of this study, please contact either the supervisor or the Research Ethics Coordinator of the Department of Computer Science and Informatics at the University of the Free State.

Contact details:

- **Researcher:** khomo_khoana@yahoo.com or (+27) 060 620 7710
- **Supervisor:** nell@ufs.ac.za or (+27) 051 401 3591
- **Departmental Research Ethics Coordinator:** BeeldersTR@ufs.ac.za or (+27) 051 401 9320

9. How do I give my consent to participate?

Complete the attached consent form if you understand and agree to take part in this study. Please submit the completed consent form to the researcher. You may keep this information sheet for your own records.

Appendix I – Participant Consent Form (Phases 1, 2 & 3)

Study title: Source code comprehension: Decoding the cognitive challenges of novice programmers

Researcher: Mr. Pakiso. J. Khomokhoana (*khomo_khoana@yahoo.com*)

Please tick (✔) to indicate you consent to the following:

I have read the Participant Information Sheet and the nature and purpose of the research study has been explained to me. I understand and agree to take part.	Yes ☐	No ☐
I understand the purpose of the research study and my involvement in it.	Yes ☐	No ☐
I have been given sufficient time to consider whether or not to participate in this study.	Yes ☐	No ☐
I am satisfied with the answers I have been given regarding the study and I have a copy of this consent form and information sheet.	Yes ☐	No ☐
I understand that taking part in this study is voluntary and that I may withdraw from participation at any time.	Yes ☐	No ☐
I understand that while information gained during the study may be published, I will not be identified and my personal responses will remain confidential.	Yes ☐	No ☐
I know who to contact if I have any questions about the study in general.	Yes ☐	No ☐
I understand my responsibilities as a study participant.	Yes ☐	No ☐
I wish to receive a summary of the results from the study.	Yes ☐	No ☐

Initials and Surname: ……

Student/Staff number: ……

Signature: ……

Date: ……