Journal of Information Technology Education: Innovations in Practice

An Official Publication of the Informing Science Institute
InformingScience.org

JITEiip.org

Volume 23, 2024

GENERATING A TEMPLATE FOR AN EDUCATIONAL SOFTWARE DEVELOPMENT METHODOLOGY FOR NOVICE COMPUTING UNDERGRADUATES: AN INTEGRATIVE REVIEW

Catherine Higgins*	Technological University Dublin, Dublin, Ireland	catherine.higgins@tudublin.ie
Ciaran O'Leary	Technological University Dublin, Dublin, Ireland	ciaran.oleary@tudublin.ie
Claire McAvinia	Trinity College Dublin, Dublin, Ireland	claire.mcavinia@tcd.ie
Barry J. Ryan	Technological University Dublin, Dublin, Ireland	barry.ryan@tudublin.ie

* Corresponding author

ABSTRACT

Aim/Purpose	The teaching of appropriate problem-solving techniques to novice learners in undergraduate software development education is often poorly defined when compared to the delivery of programming techniques. Given the global need for qualified designers of information technology, the purpose of this research is to produce a foundational template for an educational software development methodology grounded in the established literature. This template can be used by third-level educators and researchers to develop robust educational methodologies to cultivate structured problem solving and software development habits in their students while systematically teaching the intricacies of software creation.
Background	While software development methodologies are a standard approach to structured and traceable problem solving in commercial software development, educational methodologies for inexperienced learners remain a neglected area of research due to their assumption of prior programming knowledge. This research

Accepting Editor Stamatis Papadakis | Received: June 20, 2024 | Revised: August 13, August 23, September 3, 2024 | Accepted: September 9, 2024.
Cite as: Higgins, C., O'Leary, C., McAvinia, C. & Ryan, B. J. (2024). Generating a template for an educational software development methodology for novice computing undergraduates: An integrative review. *Journal of Information Technology Education: Innovations in Practice, 23,* Article 12. https://doi.org/10.28945/5374

	aims to address this deficit by conducting an integrative review to produce a template for such a methodology.
Methodology	An integrative review was conducted on the key components of *Teaching Software Development Education, Problem Solving, Threshold Concepts,* and *Computational Thinking*. Systematic reviews were conducted on *Computational Thinking* and *Software Development Education* by employing the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) process. Narrative reviews were conducted on *Problem Solving* and *Threshold Concepts*.
Contribution	This research provides a comprehensive analysis of problem solving, software development education, computational thinking, and threshold concepts in computing in the context of undergraduate software development education. It also synthesizes review findings from these four areas and combines them to form a research-based foundational template methodology for use by educators and researchers interested in software development undergraduate education.
Findings	This review identifies seven skills and four concepts required by novice learners. The skills include the ability to perform *abstraction, data representation, decomposition, evaluation, mental modeling, pattern recognition,* and *writing algorithms*. The concepts include *state and sequential flow, non-sequential flow control, modularity,* and *object interaction*. The teaching of these skills and concepts is combined into a spiral learning framework and is joined by four development stages to guide software problem solving: understanding the problem, breaking into tasks, designing, coding, testing, and integrating, and *final evaluation and reflection*. This produces the principal finding, which is a research-based foundational template for educational software development methodologies.
Recommendations for Practitioners	Focusing introductory undergraduate computing courses on a programming syllabus without giving adequate support to problem solving may hinder students in their attainment of development skills. Therefore, providing a structured methodology is necessary as it equips students with essential problem-solving skills and ensures they develop good development practices from the start, which is crucial to ensuring undergraduate success in their studies and beyond.
Recommendations for Researchers	The creation of educational software development methodologies with tool support is an under-researched area in undergraduate education. The template produced by this research can serve as a foundational conceptual model for researchers to create concrete tools to better support computing undergraduates.
Impact on Society	Improving the educational value and experience of software development undergraduates is crucial for society once they graduate. They drive innovation and economic growth by creating new technologies, improving efficiency in various industries, and solving complex problems.
Future Research	Future research should concentrate on using the template produced by this research to create a concrete educational methodology adapted to suit a specific programming paradigm, with an associated learning tool that can be used with first-year computing undergraduates.

Keywords	computing education research, educational software development methodology, computational thinking, CS1 education

INTRODUCTION: CONTEXT FOR RESEARCH

The World Economic Forum (2020) has highlighted the importance of digital skills in the future job market, emphasizing the need for a larger workforce proficient in software development. Therefore, increasing the number of software graduates is crucial in an advancing technological world. However, universities globally face significant challenges with student retention and competency, especially in computing courses (Price & Smith, 2014; Zarb et al., 2018). High dropout rates and varying levels of preparedness among students hinder the ability to produce a consistent stream of capable graduates (Petersen et al., 2016). Studies suggest that many students struggle with the rigorous and often abstract nature of computer science courses, leading to high attrition rates (Mahatanankoon & Wolf, 2021). Addressing these issues through improved teaching pedagogies is essential to ensure that more students not only complete their studies but also possess the necessary skills that the software industry requires.

Learning how to develop software means that students have to learn how to problem-solve. Problem solving in the context of providing software solutions requires software developers to have a variety of skills, such as analysing a problem to understand its functional requirements, designing or reusing algorithms as a strategy to fulfil these requirements, learning programming syntax, employing software constructs, and applying testing techniques (Stachel et al., 2013). Therefore, problem solving can be considered fundamental to anyone considering a career as a software developer (Hazzan et al., 2011) and consequently must be a core element of undergraduate courses in software development. This is reflected in the growing interest in the computing education research (CER) community in examining the educational advantages of scaffolding problem planning (Loksa et al., 2016; Prather et al., 2020), particularly at the novice freshman undergraduate level, typically known as CS1 in the literature (Hertz, 2010), as this is where many of the issues with retention, competency, and recruitment occur. However, it has been found in a systematic review conducted by Luxton-Reilly et al. (2018) that the teaching and learning of problem solving is a topic that is less mature than research into developing programming techniques.

When examining potential pedological mechanisms for CS1 education to ensure best practice, it is natural to turn to the software industry and its approach to producing quality software. Commercial software development is typically based on using a software development methodology that provides defined stages for problem solving from initial analysis, design of a solution, implementation, and testing, through to delivery of a quality software solution (Boehm, 2006). Therefore, the design of software development methodologies has been an active topic of research for the software industry since the 1980s (Abbas et al., 2008). However, there is an issue with using commercial methodologies directly in education as they assume its users have prior programming and development knowledge, which renders them incompatible with inexperienced learners. This means that using a commercial software development methodology in an educational setting with novice learners would not be suitable without some adaptation.

In our examination of the literature for existing software development methodologies specifically developed for use in CS1 education, termed in this paper as Educational Software Development Methodologies (ESDM), current and active research into the area were not found, with isolated examples of ESDMs being at least ten years old and not commonly used. This lack of research may be partly due to the fact that software development courses for CS1 students are typically framed by a programming paradigm and language, so methodologies would need to be adaptable to various paradigms. Examples of paradigms are object-oriented paradigms that could use Java as a language of choice and procedural paradigms that use C as their language. In CS1 courses, Siegfried et al. (2021) found that the most common languages used in CS1 courses are Java and Python, which can both be classified as belonging to an object-oriented paradigm. Moreover, an ESDM needs to incorporate a learning curriculum to teach novice learners the essentials of software development. This indicates

that the development of an ESDM is not a trivial exercise and may explain the current gap in active research into these methodologies.

PROBLEM STATEMENT

The literature reports that research into mechanisms for teaching CS1 students how to design and develop solutions to programming problems are less advanced than mechanisms that focus just on teaching programming techniques, despite the need for CS1 students to have those skills. Given the success of software development methodologies in designing and developing solutions in the industry, coupled with the lack of current research into ESDMs for novice learners, this clarified for us that there is a gap in the research into ESDMs and their potential for CS1 learners. This academic paper builds upon the work presented in the corresponding author's PhD thesis on this topic (Higgins, 2021) by synthesizing and extending the insights from that research into a comprehensive and actionable template methodology. While the PhD work and two initial conference proceedings (Higgins et al., 2017a, 2017b) focused on proposing a prototype educational software development methodology and a conceptual framework, this manuscript significantly broadens and deepens that foundation. It collates the findings from that body of work into an integrative review, which coalesces into a foundational template for a methodology where a template in this context is a blueprint designed to guide the development of a concrete methodology by future researchers. The aim of the template is to serve as a research-based starting point that outlines the essential components and stages required, which can be customized to suit various educational contexts for CS1 cohorts. Sharing these findings allows other researchers to build upon this foundation and further explore innovative approaches in this field. The research question for this paper is presented in Table 1.

Table 1. Research question

Number	Research question
RQ1	What are the essential constitutional elements of a foundational template for an Educational Software Development Methodology (ESDM) suitable for CS1 learners?

This paper is organized as follows. In the next section, we present a background literature review that examines the specific issues in undergraduate problem-solving education and the topic of software development methodologies. The subsequent section details the process used to guide the integrative review. This is followed by the findings from the review, then a discussion of these findings, which culminates in a presentation of a foundational template for an educational software development methodology. The paper concludes with the limitations of this research and its conclusions.

LITERATURE REVIEW

Problem solving and analytical thinking skills are students' major weaknesses in undergraduate computing courses (Castro, 2015; Koulouri et al., 2015; Lister et al., 2004; Uysal, 2014; Whalley & Kasto, 2014), with no new evidence being reported in current literature to contradict this view.

In examining how wide-spread the issue with teaching problem solving is at the undergraduate level, a systematic literature review into research trends over 15 years by Luxton-Reilly et al. (2018) found that the process for teaching problem solving to computing undergraduates is unclear in the literature. They contended that it was rarely as defined as the teaching of programming and that tools for problem solving were still at an immature stage of development. A similar review carried out by Silva et al. (2019) into research on teaching algorithms to CS1 students from 2014 to 2018 found that most of the research was dominated by investigations into how to teach programming using various languages and tools. They concluded that sourcing effective pedagogical strategies for teaching CS1 was still a major concern for educators. This is an issue for students as, without a methodological framework, many learners may engage in maladaptive cognitive practices when first learning how to develop software. Examples of such practices include immediately attempting to program a solution

and ignoring the need to develop a plan first; and crafting a solution by reusing existing code whether it is appropriate or not (Huang et al., 2013). These practices can become very ingrained in student practice and can hamper student efforts on their learning journeys (Huang et al., 2013; Lister et al., 2006), which has an impact on failure rates (Bennedsen & Caspersen, 2019; Watson & Li, 2014), and issues with retention (Petersen et al., 2016).

Examining commercial problem-solving methodologies specifically designed for the software development industry has been a robust and very important area of research and growth for the commercial software community since the 1980s (Shama & Shivamanth, 2015). Having a well-defined methodology ensures that software can be developed in a systematic fashion where there is traceability from problem specification right through to the final work product. This means that all work can be measured, improved, and potentially reused. The continuous evolution and reliability of the most commonly used software development methodologies and processes come from a strong evidence-base of use across an international software engineering community over many years. Recent examples of work being published in this area are the adaption of methodologies to suit cloud computing (Panigrahi et al., 2021) and for managing large projects during the COVID-19 lockdowns (Butt et al., 2021).

Two principal methodologies have dominated the software industry over the past three decades. One is the oldest and more generic *Software Development Life Cycle (SDLC)* methodology, which contains the stages of carrying out a preliminary analysis of the problem, defining requirements for solving the solving, designing a solution, implementing the design, testing and continued maintenance of the solution (Royce, 1987). However, in the intervening years, many improvements and amendments have been made to the SDLC methodology. This has included the realisation that due to the complex nature of eliciting user requirements, gathering fixed requirements up-front that then cannot be changed later in the project is unfeasible and can lead to unusable systems. A more effective route is iterative and incremental development, where initial planning generates initial requirements. These requirements are developed through iterative development cycles involving continuous user feedback, and then other features are incrementally added until the entire project is complete (Gilb, 1985). An early example of such a process is the *Unified Process* (Jacobson et al., 1999), which somewhat blurred the lines between the SDLC and other methodologies. It can be argued that all other methodologies and processes have their roots in the SDLC as they use the same essential stages, albeit in different configurations (Shah et al., 2016). Therefore, this suggests that an ESDM should contain stages that reflect the essential SDLC stages.

The second principal methodology is *Agile,* which is arguably the most popular and widely used methodology in current commercial software development (Bustard et al., 2013; Napoleão et al., 2020; Umran Alrubaee et al., 2020). The main characteristics of this methodology originate from iterative and incremental development, which involves breaking development work into small increments that are iteratively developed until they are correct. From an educational perspective, it has been found that the nature of fast, iterative development with Agile makes the inclusion of software development processes based on this methodology (such as XP, Scrum) into undergraduate-level curricula inappropriate for CS1 learners and more appropriate to students in their final or penultimate year of undergraduate study where they have a degree of experience and proficiency in software development (Devedzic, 2011). However, the basic philosophy of iterative and incremental development is viewed as best practice and is appropriate for CS1 learners, so it should be incorporated into an ESDM in an appropriate manner.

One interesting observation emerged when carrying out a review of software development methodologies and processes in the literature. It was found that despite the wealth of established software development processes aimed at experienced developers working on large projects, there are few available for CS1 courses. There is evidence of current research into improving CS1 outcomes (Edwards et al., 2020; Izu & Weerasignhe, 2020; Lishinski et al., 2016; Margulieux et al., 2019). However, while problem solving and developing algorithms are an explicit part of most introductory CS1 courses, the

problem-planning aspects of software development generally appear informally in the literature with little mention of providing students with a suitable software development methodology or process. Table 2 synthesises the educational software development methodologies found in the literature which are appropriate for CS1 learners.

Table 2. ESDMs with a list of their positive and negative characteristics (Higgins, 2021)

Methodology	Positives	Negatives
STREAM - Stubs, Tests, Representation, Evaluation, Attributes, and Methods (Caspersen & Kolling, 2009)	This methodology has a focus on incremental development. It contains six clear stages that allow students to incrementally build up a solution to a problem.	Can only be applied in an object-oriented environment with an objects- first philosophy and is closely linked to Java, as opposed to being a more general methodology that could be used with any development paradigm and programming language.
P³F framework Principles, Patterns, and Process (D. R. Wright, 2011)	This methodology has a focus on design. It uses four elements to allow students to employ expert design strategies to ensure they design efficient and sustainable solutions.	Requires students to understand the nature of design patterns. A topic with which novices would not be familiar.
Programming methodology (Hu et al., 2013)	This methodology focuses on the planning stage of development. It uses a five-stage process to facilitate students in creating plans and goals for their solution to ensure they fully understand the nature of the problem.	Uses a block-based language (similar to Scratch) to develop the plans. This language has its own syntax, which students must master, as well as mastering the programming language that will implement the solution.
(Morgado & Barbosa, 2012) There is no name attributed to this methodology.	This methodology focuses on planning and design. It contains two stages supplied with rules that allow students to fill in template forms to help with their planning. A prototype solution to the problem is supplied by the instructor.	This methodology requires the use of premade prototypes for each solution which makes the process not applicable outside of the classroom.
POPT – Problem-Oriented Programming and Testing (Neto et al., 2013)	This emphasizes the importance of testing in software development and supplies steps for planning and undertaking tests.	Testing is only one aspect of devising a solution in software development, so this is a limited methodology.

From Table 2, all of these methodologies have a specific focus on a particular aspect of development or a specific programming paradigm. For example, the P³F framework focuses on design; POPT focuses on testing; Hu et al.'s (2013) programming methodology can only be used with its block-based language; and Caspersen and Kolling (2009) focus on development in an object-oriented environment. The diversity of all of these concrete implementations of ESDMs, coupled with their age, illustrates the gap in the research into understanding the essential constitutional elements required in ESDMs for CS1 learners, which support all aspects of developing software solutions using any programming paradigm.

RESEARCH METHODS

In the context of designing ESDMs, Zhang and Liu (2012) state that along-side the development stages found in a commercial software development methodology, an educational methodology must also incorporate a knowledge of educational practice, software development, and problem solving. Therefore, it is important to contextualise the environment in which a methodology based on the foundational template would be deployed. This is an undergraduate computing first-year setting where CS1 students may have little or no prior experience designing and programming solutions, so

they need a methodical approach when learning software development in order to allay the prospect of maladaptive cognitive habits being inadvertently engrained in their learning. This means that an ESDM template requires the inclusion of two core features.

- Given students' inexperience in software development, it is imperative that the essential key knowledge and skills required to become competent developers is included as part of the methodological template.
- When solving problems, students must learn how to apply the knowledge and skills in a methodical fashion to encourage good developmental habits. For that reason, and consistent with a commercial software development methodology, an ESDM template requires a set of developmental stages to guide the student developers through the stages of systematic problem solving when producing software solutions.

The research question for this paper was presented in Table 1. This is now expanded into two sub-questions, presented in Table 3, based on understanding the context and demographic for the main question.

Table 3. Research sub-questions

Sub-question number	Research question
RQ1.1	What key knowledge and skills are necessary for CS1 learners to become competent software developers, given their inexperience in the area?
RQ1.2	How can problem-solving developmental stages be structured within an educational software development methodology (ESDM) to ensure that students systematically practice and apply the knowledge and skills while encouraging good developmental habits?

In order to answer these two research sub-questions, four research topics were reviewed which are summarised in Table 4.

Table 4. Summary of research topics and the rationale for including them in the integrative review

Topic	Rationale for review
Teaching CS1	To understand the characteristics, issues, and current best practice in software development education at the introductory undergraduate level (CS1).
Computational Thinking	Given that computational thinking is defined as a set of skills that are used to formulate solutions to problems in an algorithmic form (Mohanty & Bala Das, 2018), this topic was included in order to identify these skills and examine how they could be incorporated into a template for an ESDM.
Threshold Concepts	This research was interested in methodical routes for establishing core constructs that should be taught in CS1. One appropriate mechanism identified in the literature was the theory of Threshold Concepts in identifying such concepts.
Problem Solving	To understand problem solving and its role in guiding the developmental stages when solving problems using an ESDM.

Given this wide suite of topics, an integrative review that enables the synthesis of knowledge gathered using different research approaches in a fragmented field (Cronin & George, 2023) was deemed the most appropriate type of review. Whittemore and Knafl's (2005) five stages of the integrative review process – "problem identification", "literature search", "data evaluation", "data analysis" and "presentation of findings" – formed the guide for the review.

Figure 1 summarises the review process by mapping the five stages of the integrative review onto the work completed. An explanation of the methods used for the literature search into the four topics is presented in the following sections.

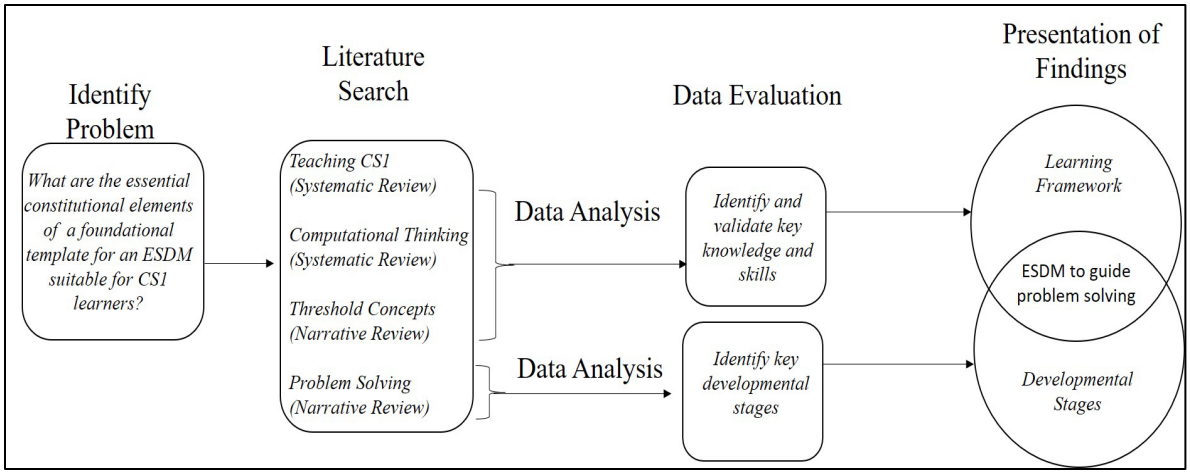

Figure 1. Components of the integrative review process conducted for this research paper

PROBLEM SOLVING

A narrative review was chosen as being most suitable for this stage of the integrative review, given the broad scope of this topic. A wide range of books and research papers, referenced in the results section of this paper, formed the basis of the findings for this narrative review. The findings were synthesised to yield recommendations for the developmental stages of the proposed software development methodology.

TEACHING CS1

Given the extensive research that has been carried out in the areas of teaching software development, it was decided that a systematic literature review of the area would be required to capture the state of software development education in CS1. The aim of this review is to identify the knowledge and skills that an ESDM template would need to incorporate.

The systematic review was guided by the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) standard (Moher et al., 2009), which provides a comprehensive framework for conducting systematic reviews. The PRISMA process was chosen for its rigor and clarity, ensuring that the review is transparent and minimizes bias. The PRISMA guidelines involve four phases: identification, screening, eligibility, and inclusion.

In the identification phase, we searched multiple databases, including ACM Digital Library, ERIC, IEEE Xplore, ScienceDirect, and Taylor & Francis, for relevant literature published between 2000 and 2020. The search terms used were comprehensive, including keywords such as "teaching programming," "teaching of programming," "teaching computer programming," "teaching of computing programming," "teaching coding," "teaching of coding," "teaching algorithms," and "teaching software design." This ensured a broad capture of relevant studies.

During the screening phase, we reviewed the titles and abstracts of the retrieved papers to identify those that met our inclusion criteria. Papers were included if they related to the tuition of software development to CS1 level students. We applied exclusion criteria to eliminate duplicate papers across databases, non-peer-reviewed papers, and papers not specifically aimed at the process of teaching software development to undergraduate first-year students.

In the eligibility phase, we conducted a review of the remaining papers to confirm their relevance and ensure they met all inclusion criteria. This screening process helped us focus on quality studies directly pertinent to our research.

Finally, in the inclusion phase, we compiled the selected studies for detailed analysis, which resulted in 356 such studies. This systematic approach, guided by the PRISMA standard, ensures that our

review is comprehensive and that the conclusions drawn are based on a robust set of evidence. A flowchart of this review process is provided in Figure 2.

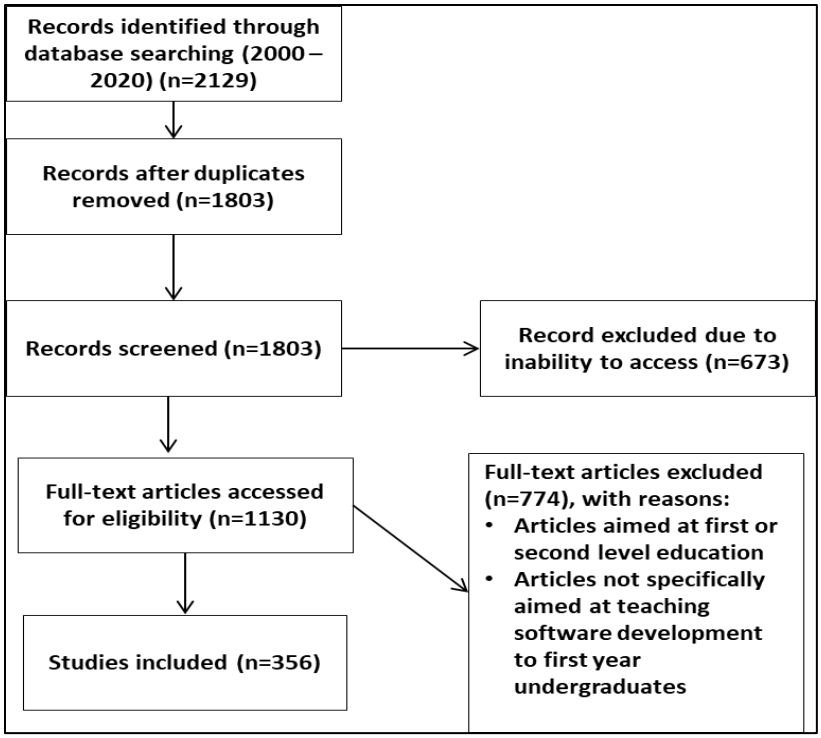

Figure 2. PRISMA (Moher et al., 2009) flow diagram illustrating the steps in the systematic literature review undertaken from 2000–2020 into software development education at the CS1 level

Role of Computational Thinking

Computational thinking (CT) is defined as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent" (Cuny et al., 2010, as cited in Wing, 2011). However, despite extensive research into the area over the past 20 years, it was found that there remain questions on what constitutes a definitive set of its skills (Allan et al., 2010; D. Barr et al., 2011; Denning, 2017; Denning & Tedre, 2019; Grover & Pea, 2013; Weintrop et al., 2016). Consequently, the aim of this review was to provide a set of the most commonly agreed skills for CT and to see if they overlapped with the skills identified in the review of teaching CS1.

Similar to the review conducted on teaching CS1, this research was guided by the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) standard (Moher et al., 2009). This review focused on papers published from 2006, the origin date of active research into computational thinking (CT), to 2020.

The identification phase involved searching multiple databases, including ACM Digital Library, ERIC, IEEE Xplore, ScienceDirect, and Taylor & Francis, using the keyword "computational thinking."

In the screening phase, titles and abstracts of the retrieved papers were reviewed to determine their relevance. The inclusion criteria were that papers must be peer reviewed and suggest a set of skills associated with CT based on a review of earlier work or empirical evidence. Exclusion criteria were applied to eliminate papers that did not suggest a skillset or based their skillset on just one resource.

During the eligibility phase, a review of the remaining papers was conducted to confirm they met all inclusion criteria to ensure that quality studies directly pertinent to our research questions were included.

Finally, in the inclusion phase, the selected studies were compiled for detailed analysis which included 103 such studies. A flowchart of this review process is provided in Figure 3.

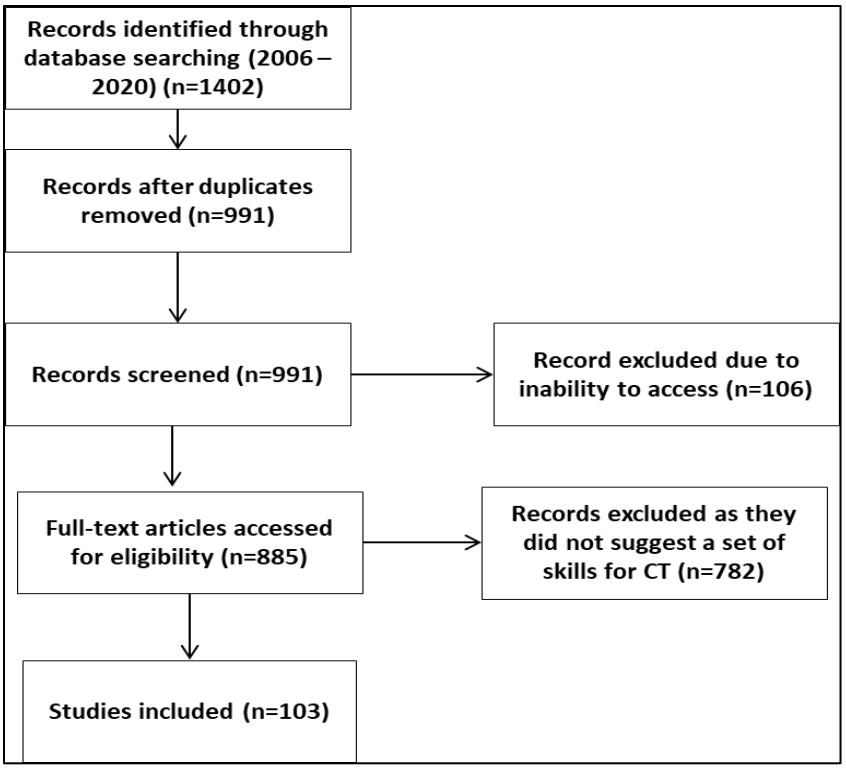

Figure 3. PRISMA (Moher et al., 2009) flow diagram illustrating the steps in the systematic literature review undertaken from 2006–2020 into the identification of computational thinking skills

Role of Threshold Concepts

Threshold concepts (TCs) have been suggested as an approach for charting the learning journey of students in any topic. J. H. F. Meyer and Land (2003, 2005) argue that within a topic there are specific yet difficult concepts that once grasped are not forgotten, and which can transform student learning and understanding. They state that "crossing the threshold" to understanding these concepts is crucial if students are to progress and become proficient in their field of study. This suggests that understanding which TCs are fundamental to a topic would be useful in identifying the core concepts for that topic. Therefore, a broad narrative review was conducted into threshold concepts, and their potential role in identifying key knowledge required by CS1 students. All relevant range of books and research databases were used in this review, referenced in the results and discussion section, which resulted in the most commonly cited threshold concepts for CS1 education being identified.

Results

This section presents the findings from the integrative review to answer the two sub-research questions posed in Table 3. This incorporates an understanding of the key knowledge and skills required by students, coupled with a recommended suite of developmental stages to allow students to practice

their problem solving. The Discussion section of this paper analyses these findings to answer the main research question posed in Table 1.

RQ1.1 Key Knowledge and Skills

In identifying key knowledge and skills for CS1 students, it is first important to understand how such students are typically taught in order to suggest a best practice mechanism for how the knowledge and skills should be taught. Teaching software development has traditionally been instructor-led and instructor-centred (Saulnier et al., 2008). In the typical classroom, this generally translates into traditional lectures supplemented by laboratory work (Schiller, 2009). With this approach, when trying to understand programming constructs, students often use low-level learning strategies to memorize course information. However, such an approach suggests that students do not necessarily understand what they are doing or why they are doing it. When the traditional instructional approach is exclusively used, students are passive recipients of information from the instructor (Prince, 2004) and the underlying pedagogy is mainly behaviourist in nature.

In contrast, the idea of "learning through experience" is an example of active learning. The core element of active learning is encouraging student engagement through activities such as peer instruction and paired programming, which puts the responsibility for learning onto the learner (Greer et al., 2019; Schiller, 2009; Williams & Chinn, 2009). Thevathayan and Hamilton (2015) claimed that passive learning contributes to the creation of incorrect mental models and suggest that meaning and correct models can only be forged though active learning. However, Mayer (2004) and Kirschner et al. (2006) indicated that the general approach of "learning by doing" requires a note of caution as there is little empirical evidence to support this statement in the context of novice learners carrying out pure discovery. Sweller et al. (2011) also argued that novices do not possess the underlying mental models for unsupported experimentation in learning. Therefore, any application of active learning techniques needs to ensure that tuition is properly supported to ensure learning progresses in an effective manner (D. L. Meyer, 2009). This would suggest that an ESDM template should promote active learning though scaffolded problem solving.

A spiral curriculum (see Figure 4), using the spirit of Bruner's (1960) spiral curriculum, is particularly suitable for learning software development due to the nature of development, where each topic is a building block to more complex topics (Armoni, 2014; Jaime et al., 2016).

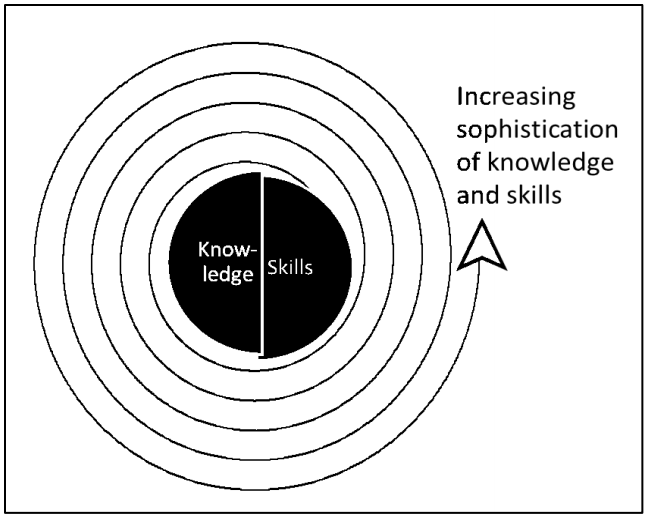

Figure 4. Spiral model for learning, which illustrates the knowledge and skills taught to students act as prerequisites to learning more sophisticated knowledge and skills in a growing spiral

In such a curriculum, learning begins with basic elementary knowledge and associated skills, with new knowledge and skills building on top of previous ones, so previous knowledge and skills are revisited and used with increasing levels of difficulty (Santana & Bittencourt, 2018). Learning how to develop software requires skills to create a working solution, as well as key knowledge, which includes the computational concepts used when employing the skills. Therefore, the attainment of competence as a developer involves learning a subset of knowledge and skills to initially solve basic problems. When a student is being taught a new unit of learning, they will access all of the knowledge and skills that make up that unit but will also need to use the knowledge and skills of previous units as a prerequisite to understanding the current unit.

Identification of key knowledge

Key knowledge consists of the constructs and concepts that students need to understand in order to develop software solutions. These include *programming constructs,* which is the theory behind general computational concepts such as data declaration, iteration, and modularity), and *program syntax,* which is the syntax of the programming language being used to develop software (Qian & Lehman, 2017). This domain does not include the skills involved in the practical application of that knowledge. Identifying this knowledge was facilitated by the systematic review of the characteristics and current practice in teaching CS1. It was further verified by a study of threshold concepts in computing to establish a recommended list of knowledge constructs.

The fundamental programming constructs for introductory software courses are recommended by the ACM, IEEE, and AAAI Joint Task Force on Computing Science Curricula (Kumar et al., 2023). These include an understanding of variables, expressions, and assignments (sequential statements), conditional statements, iterative statements, and modularity. While it can be argued that these key computational constructs are already known and are broadly similar across CS1 courses (Soares et al., 2015), we were interested in methodical routes for establishing key constructs. One potential mechanism identified in the literature is the theory of Threshold Concepts (TCs). TCs are considered to be core concepts in a domain of interest where understanding these concepts is key to transforming the way students understand the domain, allowing them to be successful in their learning (Cousin, 2006; J. H. F. Meyer & Land, 2003). However, while some researchers have argued that the looseness of the interpretation of TCs and the subsequent difficulty in identifying them is worrisome (O'Donnell, 2010; Rowbottom, 2007), others argue that there are certain concepts that will tend to be particularly troublesome and transformative to most students and those concepts are worthy of study (Rountree & Rountree, 2009; Sorva, 2010; A. L. Wright & Gilmore, 2012). This suggested to us that the inclusion of TCs to define the key knowledge would be beneficial in identifying and streamlining the concepts that need to be learned.

To date, there are no consensus TCs for software development (Yeomans et al., 2019), and no current research was found to refute this statement. This is partially due to the relative infancy of TCs as a research area, but it is also due to the difficulty of identifying such concepts (Shinners-Kennedy & Fincher, 2013). Despite this issue, there have been many attempts to clarify the TCs in software development. Much of the original and longest-standing work has been carried out by a number of international researchers from Sweden, Wales, and the US who formed a "Swedish Group" in 2005 to gain consensus on what constituted TCs (Boustedt et al., 2007; Eckerdal et al., 2006; Sanders et al., 2008, 2012; L. Thomas et al., 2014). As a summary of their findings, they suggested **abstraction** (later clarified as data abstraction), **object orientation** (which they recognised as being a very wide area that probably contains other TCs), and **memory/pointers** as candidate concepts.

In attempting to ascertain appropriate TCs for the knowledge element of learning, the review of the literature found that the research on identifying TCs for software development is approximately ten years old. More recent papers with a focus on TCs for software development tend mainly to cite the established work of the Swedish Group (2007-2014) and Sorva (2010), as exemplified by Sim (2017). A synthesis of candidate threshold concepts for software development is presented in Table 5.

Table 5. Summary of candidate threshold concepts for software development as identified in the literature

Researcher	Candidate TCs		
Swedish Group (Boustedt et al., 2007; Eckerdal et al., 2006; Sanders et al., 2008, 2012; L. Thomas et al., 2014)	Using pointers	Understanding Object Orientation	Applying data abstraction
Alston et al. (2015)	Understanding basic programming principles	Understanding and applying abstraction	
Sorva (2010)	Understanding program dynamics	Understanding and applying information hiding (data abstraction)	Understanding Object Orientation
Holloway et al. (2010)	Applying recursion	Understanding Object Orientation	
Shinners-Kennedy (2008)	Understanding program state		
Vagianou (2006)	Understanding program/memory interaction		
Khalife (2006)	Understanding and modelling the notional machine		

In attempting to find commonality in the choice of potential concepts given in Table 5, the concept of *data abstraction* appears three times. This concept is the ability to mentally zoom in and out of a complex data item (such as an aggregate object) to understand it at different levels of detail. We rejected this concept as it is our contention that *data abstraction* is a skill that is acquired through practice rather than a concept. We also contend that understanding the nature of abstraction is encapsulated under both *object interaction* and *program dynamics* as it necessitates understanding how data is conceptually represented in memory during program execution. The concept of *understanding basic programming principles* is rejected as being too vague to be useful to this research. *Applying recursion* is also rejected as it is viewed as being too advanced a topic to be considered for an introductory software development course.

The topic of *object orientation* has been proposed by three researchers in the table. Given that object orientation itself is a large topic rather than a concept, the subtopic of **object interaction** is a proposed threshold concept, as this is a foundational aspect of the use of object orientation in general.

Understanding program dynamics (Sorva, 2010) is a superset of many of the other concepts. For example, *program/memory interaction* by Vagianou (2006), the notion of *state* by Shinners-Kennedy, *pointers* by the Swedish Group, and the *notional machine* by Khalife (2006). Given that program dynamics is a very broad concept, we refined it further as a suite of three threshold concepts that represent program dynamics and which map onto the order of concepts taught as recommended by ACM/IEEE/AAAI Joint Task Force on Computer Science Curricula (Kumar et al., 2023). These concepts are termed in this paper as **State and Sequential Flow**, **Non-Sequential Flow Control**, and **Modularity**, which by their nature incorporate the notion of state, program/memory interaction, and use of the notional machine (Higgins et al., 2017a).

The refined list of threshold concepts and their definitions accepted in this research are now presented and defined in Table 6. From examining the descriptions of the TCs in Table 6, each concept builds on the knowledge learned in the previous concept, which means that there is an order to learning each TC. *State and Sequential Flow* should be the first concept to be taught, followed by *Non-sequential Flow Contro*l, followed by *Modularity*, with *Object Interaction* optionally being the final concept taught

if the development paradigm is object-oriented. This final concept is included as Siegfried et al. (2021) found that approximately 80% of CS1 courses teach in an object-oriented paradigm. However, this concept can be easily removed in an ESDM if not appropriate to a specific cohort. To highlight this order, the TCs have been codified as TC1 to TC4. Having an order of learning has also been reflected in the spiral curriculum at the start of this section.

Table 6. Threshold concepts accepted in this research

TC1: State and Sequential Flow	This stage focuses on the state changes that occur on basic data items (e.g., characters, numbers, strings) when fundamental sequential actions such as printing, assignment, and updates are performed.
TC2: Non-sequential Flow Control	This stage focuses on state changes to data items through the use of non-sequential actions such as iterative and conditional actions and how they both state and flow control through a program.
TC3: Modularity	This stage focuses on partitioning a solution into modules and functions, which has a unique effect on state and flow control.
TC4: Object Interaction	This is an optional stage that is only included in the template if the development paradigm is object orientation, which involves the creation and manipulation of objects.

Identification of skills

This section identifies the skills that are required in order to apply knowledge to successfully problem solve and produce computational solutions to problems. This was facilitated by the systematic review into teaching CS1. In order to verify the skills identified by that review, a separate systematic review of computational thinking (CT) was conducted.

Morgado and Barbosa (2012), citing Whitfield et al. (2007), stated that one specific issue facing computing students is the multiple skills required by learners in creating computational solutions. While the knowledge required by CS1 learners is easier to quantify, the required set of skills is somewhat more opaque. This was supported by an observation from the systematic review into teaching CS1, which found that 71% (n=253) of papers had a primary focus on teaching programming in terms of the constructs of the language or development environment, with the use of relevant problem-solving skills such as problem analysis and design being largely implicit. Even though these papers stressed the importance of problem solving, it was a topic that was less clearly defined. The remaining 29% (n=103) of papers focussed on writing algorithms in the context of teaching specific foundational general algorithms, such as searching or sorting data items. Therefore, the skills for the proposed ESDM were identified by the analysis of systematic reviews into teaching CS1, and separately, into computational thinking.

In conducting the systematic review of teaching CS1, the following set of seven skills emerged from the analysis:

Illustrate evidence of mental modelling of programming constructs and notional machines. When developing computational solutions to problems, the ability to create mental models is a key requirement (Cabo, 2015). Creating mental models includes the ability to visually conceptualise the problem to be solved, to understand general programming constructs, and to understand the state of data and how (and why) that state changes in a solution which is represented as an algorithm or program. Before a solution can be devised for a problem, it is vital that the problem itself is fully understood. In other words, understanding "what" needs to be solved comes before working out "how" to solve it. Being

able to view a problem conceptually as a mental model at various levels of detail involves the use of abstraction as a mechanism for reducing complexity (Koppelman & van Dijk, 2010).

Equally important in mental modelling involves the understanding and application of common programming constructs (e.g., selection, iteration, modularity), which are fundamental to converting a high-level understanding of the problem domain into a computational strategy. It has been documented in the literature that being unable to mentally visualise these concepts is a major obstacle in understanding software (Biju, 2013; Caserta & Zendra, 2011). Indeed, Ma et al. (2011) stated that having incorrect or insufficient mental models of basic constructs is a major reason for poor software construction proficiency. One important mental model required to understand and visualise programming constructs is a valid and consistent model of the machine executing the constructs in an algorithm or program. Formally, this mental model is called a notional machine (Cañas et al., 1994; Du Boulay, 1986; Mendelsohn et al., 1991), which presents concepts at a high conceptual level, so they are more accessible to students. This view was articulated by Dickson et al. (2020), who state that in teaching students, the aim is not to make the student's mental model of how the physical computer behaves as accurate as possible. Instead, the aim is focussed on making the student's mental model of the notional machine as accurate as possible. This viewpoint illustrates the importance of using appropriate notional machines for CS1 students, which are simplified and accessible by a cohort who have no prior mental models of the area. Berry and Kölling (2013) have observed that there are some visualisation tools available to help educators and students illustrate the state of notional machines (e.g., Jeliot, AVLIS Live) but that educators predominately use personalized approaches to presenting notional machines to students. It is universally accepted that to successfully develop software, a central role is played by this notional machine (Bower & Falkner, 2015). However, it was noted in a submission in the *Cambridge Handbook of Computing Education Research* (Fincher et al., 2020; Krishnamurthi & Fisler, 2019) that notional machines do not have a strong presence in either textbooks or curricula for CS1 courses. Therefore, having a visualisation system that helps students explicitly conceptualise their understanding of their designs and solutions would be a valuable mechanism in any proposed ESDM, and would also support the premise of active learning.

Apply Levels of Abstraction. An approach to examining the complex cognitive processes required by students learning how to develop software was suggested by Lister (2011) in his Neo-Piagetian theory of cognitive development. This contains four stages of cognitive development, with students needing to achieve stage 4 – *Formal Operational Reasoning* – if they are to become proficient developers. This fourth stage includes the ability to reason about unfamiliar situations, to use abstractions routinely and systematically, and an ability to engage in meta-cognitive activities. Put succinctly, formal operational reasoning is "what competent programmers do, and what we'd like our students to do" (Corney et al., 2012, p. 79). This skill is required for all aspects of problem solving as it reduces complexity by giving the developer a mechanism for mapping a problem and its solution onto levels of increasing detail. This allows learners to focus on a level without having to understand the details in lower levels. However, the application of abstraction is difficult for students and so needs to be taught explicitly (Koppelman & van Dijk, 2010; Kramer & Hazzan, 2006). Having a visualisation system as part of an ESDM would be useful in helping make abstraction a more explicit activity.

Apply DRY principle. A core principle of software development is known as the DRY (*Don't Repeat Yourself*) principle, which is aimed at reducing duplication of effort and work products though all aspects of development (D. Thomas & Hunt, 2019). The most common application of DRY can be seen with the use of program code reuse through modularity, where individual, useful functions to carry out a task are written (by the user, by other developers, or provided in the software library provided by the programming language); and then these functions can be called and executed in many different applications. DRY is closely associated with Reade's (1989) formal definition of *Separation of Concerns*, which is a design principle that separates software into distinct sections or concerns. Using DRY requires developers to be able to recognise patterns developing as they plan solutions, so they can map these patterns to pre-existing artefacts such as individual designs or programming functions.

This ensures that at least some of the work involved in solving a problem can be reduced to pre-written and verified artefacts that can be used in solutions, both at the design and code levels (Cabezas et al., 2020). Using this form of pattern recognition also allows abstraction to be applied, as artefacts can be used without the user having to be concerned about their inner details. This skill can be applied to problem analysis, designing algorithms and writing program code.

Perform problem analysis and decomposition. This is the ability to understand and articulate the nature of a problem, including its goal and outcome, so it can be decomposed into a series of independent sub-problems or tasks and any number of levels of sub-tasks depending on the complexity of the problem (V. Barr & Stephenson, 2011). In CS1 education, there are two approaches to analysis: top-down and bottom-up (Foster, 2014). The top-down approach (also known as stepwise refinement) was founded in the 1970s (Wirth, 2001) and has been recognized as one of the essential principles in software engineering. In this approach, a problem is broken into a series of high-level tasks. Each task is then refined in yet greater detail, sometimes in many additional sub-task levels, until the entire problem is reduced to base elements that can be programmed. Critics of top-down design state that this approach gives the developer an overall picture of the form the solution will take. They need to understand and design the entire problem first before programming can commence, which can be very time-consuming (Floyd, 2007). In contrast, the bottom-up approach (Dijkstra, 1979, pp. 41-48) starts from a thorough analysis of the problem to identify the low-level tasks that can first be implemented, then combining them together to form larger tasks and continuing the process until the entire problem is solved. This approach to software development is especially useful when there are pre-made solutions for tasks already available. However, unlike the top-down approach, a high-level view of the solution is not available at the start of the process. When low-level modules are being written, developers do not understand the higher-level modules, so it can be more difficult to monitor and manage the project. The object-oriented paradigm is naturally aligned with a bottom-up approach as the design is based around a system of separate classes that must be designed (or reused) and then combined up in a hierarchy to form a full system. Proponents of the bottom-up approach argue that this produces reusable code that saves time later in the process, and a modern iterative and incremental approach to design and coding can take place (as used in Agile processes).

An alternative approach called hybrid design or hybrid development (Ginat & Menashe, 2015) has also been suggested, which is a blend of top-down and bottom-up. A top-down design can be used to decompose a problem into small tasks, and tasks can be slowly implemented and tested using a bottom-up approach. The integration is guided by the organisation of tasks identified at the analysis level so that the sequence of integration steps can be clearly mapped back to different levels of tasks, allowing for the traceability of requirements from specification to program code. Given that a hybrid approach incorporates the best of top-down and bottom-up, it is suggested as the basis for carrying out analysis in the equivalent developmental stage in the ESDM.

Writing algorithms. This is the ability to create a language-independent design for a problem (or a problem broken into separate tasks) that indicates the principal programming constructs to be used to provide a solution. However, despite the recognised importance of algorithmic thinking when designing solutions, it is a topic that is often not taught explicitly in CS1 courses resulting in many students having difficulties when developing solutions (Koulouri et al., 2015; Robins, 2019; Veerasamy et al., 2019). Coffey (2015) highlighted this dilemma by stating that design is often treated very lightly in introductory courses. De Raadt et al. (2009) concurred by saying that such courses focus on the teaching of syntax but do not provide adequate strategies to formulate an analysis of the program domain or construct designs for solutions. This issue with design is not new, as seen in a previous study conducted by Lahtinen et al. (2005), who found that the process of designing a solution proved to be a far more difficult step for learners than programming the solution. One issue with design observed by Garner (2007) was the lack of automatic feedback available to students when writing algorithms, so they often overlook the need for the structured thought processes required in software design and instead move to experimental coding. Having a support tool that can provide feedback on the

correctness of algorithms is important to keep students engaged and to help them understand and correct their mistakes (Hummel, 2006; Safari & Meskini, 2016; Sewell & St George, 2009). However, while there are several visualisation tools reported in the literature to aid with understanding the state of the program execution (e.g., Jeliot (Moreno et al., 2004), Python tutor (Guo, 2013) and Ville (Rajala et al., 2008)), there is a low but growing number of tools to support students in the design of algorithms, e.g., Algorithm Animation System (Mornar et al., 2014); AlgoTouch (Frison, 2015); and Algorithmic Design Language (Jeff & Nguyen, 2018).

Liikkanen and Perttula (2009) note that novices also have specific problems when integrating the decomposed solutions of a suite of sub-problems back into a complete solution, which can lead the novice to apply a trial-and-error strategy, trying out different solutions until one is found that works. This implies that students can stray from designing and coding sub-problems systematically and move into 'hacking' together with a complete solution. D. R. Wright (2012) noted that there was very little research available that examines how new learners make decisions when producing designs. However, it is known that novices will typically reason backward (start from a hypothesis about the solution and work backward towards what is known about the problem). In contrast, experts can reason forward (start from what is known and work toward the unknown) and backward as appropriate (Ericsson & Charness, 1997). To help learners articulate their strategies in a comprehensible way, a design should be represented as an algorithm written using common mechanisms such as pseudocode or flowcharts to avoid the syntax rules of a programming language, with flowcharts helping students visualize execution (Vahid et al., 2019). Using diagrams in design has been shown to improve students' comprehension of computing concepts (Smetsers-Weeda & Smetsers, 2017), with flowcharts (for general development) and activity diagrams (in object-oriented development) commonly used (Armaya'u et al., 2022). Consequently, flowcharts for design are suggested for the ESDM template, with such charts as part of a visualisation system to allow students to explicitly develop their designs, integrate them into a final solution, and help stop them from deviating from their design plans.

Data Identification and Representation. When writing an algorithm or programming a solution, an essential dependent skill is the ability to identify the data that is required in the solution. Once the data is identified, then it must be computationally represented using valid variable declarations, which store a single item of data, or data structures, which store multiple items of data (Arnold et al., 2007; Feaster et al., 2012). However, it is recognised in the literature that the correct identification and declaration of variables has been a consistently difficult topic for students (Du Boulay, 1986; Kohn, 2017) for nearly 40 years. Corroborating this issue, a study conducted by Kaczmarczyk et al. (2010) found that students often had misconceptions regarding the use and memory allocation of variables. Therefore, learning how to identify the required data items as part of algorithm composition is an essential skill related to any ESDM.

Evaluate Solution. Evaluating a solution has two strands. The first strand involves the ability to critique and debug algorithms and programs to eliminate both logical and syntactical errors. The second strand involves the learner being able to reflect on their journey as they navigate solving a problem. A learner's success in any field lies in their ability to engage in metacognition in order to identify any gaps in their knowledge and skills required to succeed in a topic (Sternberg & Sternberg, 2016). Metacognition is 'cognition about cognition'; it refers to the "conscious planning, control and evaluation of one's own thoughts that engage in the learning processes" (Sternberg & Sternberg, 2016, p. 234). Bergin and Reilly (2005) found that students who perform well in software development use more metacognitive management strategies than their peers. As problems become more complex, the need for purposeful reflection and positive feedback becomes even greater (Havenga et al., 2011). Safari and Meskini (2016) suggest that problem-solving heuristics can be taught to students through a metacognitive approach. Using extensive research, they suggest components to teach students how to problem solve, such as thinking deeply about the problem and questioning if the problem can be broken down into tasks; questioning and monitoring their strategy for solving the problem; and

continuously reflecting on their strategy and revising if necessary until a complete, integrated solution is produced. Consequently, teaching students how to articulate their learning journey and engaging in metacognition to self-reflect on their learning should be part of the ESDM template.

A summary of the skills collated from the review into teaching CS1 are presented in Table 7.

Table 7. Summary of skills identified in review of teaching CS1 and a recommendation for their representation in an ESDM

	Skills identified in teaching CS1 review	Recommended representation in ESDM
1.	Illustrate evidence of mental modelling of programming constructs and notional machines	Incorporate a visualisation tool to allow students explicitly model analysis and design outputs when they are planning solutions.
2.	Apply levels of abstraction	Use explicit models in ESDM developmental stages (as part of a visualisation system) to apply to analysis, design, and coding outputs, which will allow students to view these stages at various levels of detail and to connect them into a coherent workflow towards a solution.
3.	Apply DRY principle	Include teaching pattern recognition to reduce redundancy at the analysis, design, and coding stages.
4.	Perform problem analysis and decomposition	Use hybrid development in analysis, design, and coding developmental stages.
5.	Writing algorithms	Use flowcharts in design.
6.	Data Identification and Representation	Data identification and representation should be explicitly incorporated as part of using flowcharts.
7.	Evaluate Solution	Teach students how to debug their solutions and apply metacognition to reflect on their solutions and learning.

In validating the above set of seven skills that emerged from the review of CS1 education, it was important to identify a second body of research that could independently identify skills. One area of active research that appears to have a role in understanding the skills required in software development is computational thinking (CT). It is accepted that to become proficient software developers, students need to develop good computational thinking skills to produce software solutions (Grover, 2019; Liu & He, 2014; Lu & Fletcher, 2009). Denning (2017) states that the research community recognises CT as a set of skills rather than a set of applicable knowledge. This has ensured that CT has become an area of active interest to educators both inside and outside the computer science community (Agbo et al., 2019), with the ACM recognizing CT as one of the fundamental skills desired of all graduates (Kumar et al., 2023).

However, despite the phrase originating with the seminal educational researcher Papert (1996), interest and research in this area only became widespread since the publication of Wing's paper (2006). Therefore, given the relatively recent time period of active research into the topic, there is still a lack of consensus in the literature on exactly what constitutes CT and how it can be defined and assessed (Denning, 2017; Gouws et al., 2013; Grover, 2019; Grover & Pea, 2013; Lockwood & Mooney, 2018; Palts & Pedaste, 2020).

In the context of this research, a systematic review of the literature on CT was conducted to identify the most commonly cited skills. This review resulted in the identification of eight skills, which are presented in a bar chart in Figure 5. The skills are listed and ranked from highest to lowest based on the percentage of times they appear in a skill-set definition for CT in the reviewed papers.

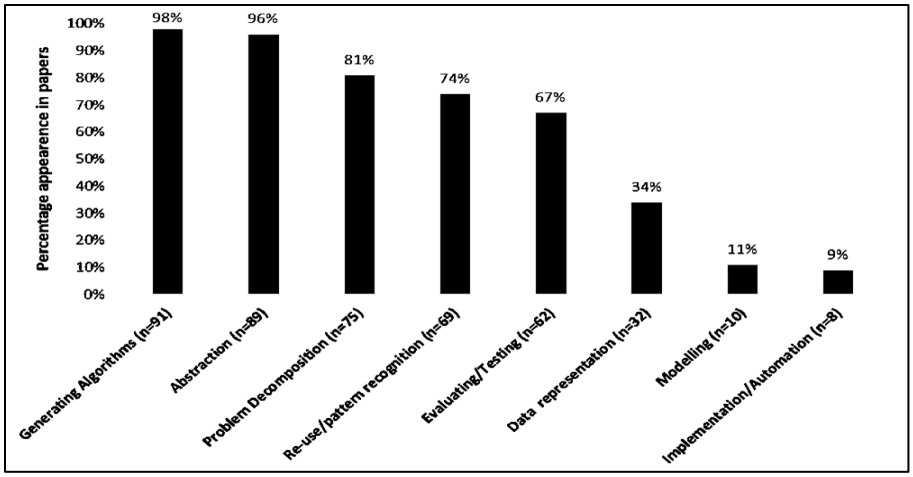

Figure 5. Most commonly listed CT skills produced by a systematic review by the researcher (number of papers n = 93)

One way of corroborating the above list was to compare it to other similar reviews. A comparable systematic review of CT was carried out by Kalelioglu et al. (2016) where they suggested the following skills listed in order of importance: abstraction, algorithmic thinking, problem solving, pattern recognition, design-based thinking, conceptualising, decomposition, automation, analysis, testing & debugging, generalisation, mathematical reasoning, implementing solutions and modelling. This list is a superset of all terms indicated in Figure 5, with our conclusion that the skill *Implementation/Automation* identified from this research project' review and the extra skills indicated by Kalelioglu are not appropriate in this research for the reasons outlined in Table 8.

In comparing the set of seven CT skills identified in Figure 5 (excluding *Implementation/Automation* for the reasons given in Table 8) against the seven skills identified via the review of CS1 in Table 7, it can be seen from Table 9 that the skills map onto each other. The one difference is the absence of integration of algorithms to form higher-level solutions in the definition of *Writing Algorithms* in Table 9. However, given its importance, as highlighted in the review of teaching CS1, the skill of *Writing Algorithms* in this research is assumed to also include the skill of algorithm integration. Therefore, the skills identified as being appropriate for this research are listed alphabetically in Table 10.

Table 8. List of CT skills presented by Kalelioglu et al.'s (2016) review that is rejected in this paper

CT Skill	Reason for exclusion
Problem solving	This is an umbrella term for the skill-set required for CT, and it is too general for it to be useful.
Design-based thinking	The skill of writing algorithms is design-based thinking, as it involves designing solutions before implementation.
Conceptualising	This is too vague a term to be useful, and it can be argued it is an inherent part of all the skills listed in Figure 5.
Mathematical-reasoning	While it has been found that mathematical ability is a predictor of success in introductory computer science modules (Bergin & Reilly, 2005), its lack of explicit inclusion in Figure 5 and its comparatively low importance level (2%) in Kalelioglu's review renders it inadmissible.

CT Skill	Reason for exclusion
Analysis	This term had a myriad of definitions in the literature review, with some authors indicating that it involves understanding the problem domain, and others (including Kalelioglu) suggested that it was a skill that resulted in the decomposition of the problem. For this review, conducting analysis is assumed to be equivalent to problem decomposition, which is defined as representing a breakdown of a problem into required sub-problems that need to be ultimately solved and integrated into a final solution. This equates to requirements elicitation in commercial software development.
Implementation/ Automation	This skill is viewed as the final work product of solving a computational problem, and it is the culmination of the other skills. It is a way to implement a problem solving strategy rather than being an inherent skill of CT itself.
Generalisation	In Kalelioglu's review, this is a term used to represent reuse in solution building, which, in this research project, is part of pattern recognition and does not warrant a separate term.

Table 9. Mapping the suite of skills identified in CS1 review to CT skills

Skills identified in CS1 review	CT skills
Apply levels of abstraction	Abstraction
Identify data	Data representation
Perform problem analysis and decomposition	Decomposition
Evaluate solution	Evaluation of solution
Illustrate evidence of mental modelling of a) programming concepts b) notional machine	Mental modelling
Apply DRY principle	Pattern recognition
Design algorithms for tasks	Writing algorithms

Table 10. Key Skills required by CS1 learners

Skill	Explanation
Abstraction	The ability to reduce the complexity of a problem and its solution by being able to zoom in and out of different levels of detail so focus need only be applied to essential characteristics. A visualisation system which employs modelling is recommended for an ESDM to help students learn and apply this important skill.
Data Representation	The ability to elicit, analyse and represent the data needed to solve a problem. This must be included when writing algorithms and subsequent code.
Decomposition (of problem)	This is the ability to decompose a problem into a series of tasks or sub-problems that need to be solved. It is the ability to analyse a problem to break it into a suite of requirements. This is part of a hybrid development approach recommended for the ESDM template.
Evaluation	This is the ability to evaluate a solution (where a solution contains all of the outputs from problem planning and programming) by testing, debugging, and critiquing it at various levels of abstraction. It also includes the application of metacognitive practices to enable students to critique their solutions and improve their learning.

Skill	Explanation
Mental Modelling	There was confusion in the literature in providing a definition for modelling as, in some instances, it was perceived as being the building of concrete artefacts, and in other places, it was taken to mean mentally modelling constructs. This confusion resulted in its low ranking in Figure 5. However, we view it as being an important skill when it is taken to mean mental modelling in problem solving. It is closely aligned to the ability to apply abstraction, and the template ESDM recommends explicit modelling in a visualisation system as a segue to mental modelling.
Pattern recognition	This is the capability for recognising when all or part of a problem has been solved before so a new design does not need to be developed. This is encapsulated by the Don't Repeat Yourself (DRY) principle (D. Thomas & Hunt, 2019).
Writing Algorithms	This is an ability to design solutions to (decomposed) tasks using computational constructs and integrate them into a higher-level solution. The use of flowcharts is recommended for this skill.

Given that learning to develop software is heavily based on solving problems, the next section focuses on research question RQ1.2 by examining appropriate developmental stages for solving problems in the ESDM.

RQ1.2 Key Developmental Stages

In devising a suite of developmental stages, the first step was to understand the classification of problems that CS1 students are expected to solve. In cognitive psychology, two major types of problems are distinguished: well-defined and ill-defined (Jonassen, 2000). Well-defined problems have one correct interpretation, and their specification provides all the information needed to solve them. In contrast, ill-defined problems have several possible interpretations and often include only fragmentary information. In the context of software development, CS1 students typically start their studies by solving well-defined problems. With experience and increased proficiency, the aim is that they should be able to ultimately move on to solving ill-defined, real-world problems (Mendonça et al., 2009). This means that as part of the curriculum of the ESDM, students should be provided with a range of interesting problems, both well-defined and ill-defined, to ensure they get the necessary practice required to become proficient in development (Kinnunen & Malmi, 2006; Rogerson & Scott, 2010).

The literature was examined as part of the narrative review of problem solving for examples of evidence based, general problem-solving methodologies that could be adapted for software development. For example, Bransford and Stein's (1993) IDEAL is a general problem-solving method that is used to *Identify (the problem), Define (the problem), Explore (options)*, *Act (on a plan),* and *Look (at the consequences)*. Other methodologies include GROW (Alexander & Renshaw, 2005) which is used in the context of cooperate coaching to help workers set goals for themselves (*Goal, Reality, Options, Way Forward*), and the OODA loop model (Boyd, 1996), which is used for decision making and refers to the decision cycle of *Observe, Orient, Decide,* and *Act*.

An appropriate strategy for problem solving that can be applied to a diverse set of educational domains – including software development – was found in Pólya's *How to Solve It* (1957). This text postulated that there are four main stages to problem solving (the context was mathematical problem solving). These four stages are "*understand the problem"*, "*make a plan to solve the problem"*, "*carry out the plan"* and "*reflect on the success of the plan"*. As observed in the introduction to this paper, commercial software development methodologies are based on providing strategies to carry out the basic software development practices of requirements analysis, design, coding (i.e., programming), and testing. Therefore, Pólya's four-stage model can be aligned for use as a foundational strategy for problem solving in software development (Higgins et al., 2017b). Comparable examples were found in the literature, which included Middleton's (2012) strategy that aligned deliverables from the model against the deliverables from a Waterfall process (Royce, 1987) of software development and Thompson's (1996) book *How to Program It*, applied Pólya's stages to functional programming using Haskell.

Consequently, the developmental stages of the proposed ESDM template are based on Pólya's basic problem-solving strategy, which was adapted to align with the commonly agreed stages in commercial software development methodologies but rebrand them to suit novice learners. The four developmental stages of the methodology are illustrated in Figure 6.

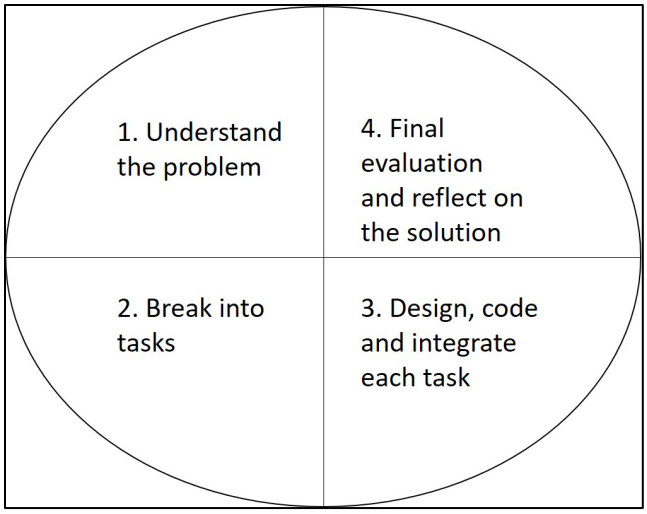

Figure 6. ESDM developmental stages

The numbers associated with each stage in Figure 6 indicate that there is an order to the stages with Stage 1 being the first stage and Stage 4 being the last stage. The developmental stages are also iterative and incremental, as is common practice in commercial software development (Al-Saqqa et al., 2020). In previous sections, having a visualisation system that helps students explicitly conceptualise their understanding of their designs and solutions was highlighted as being a valuable resource. Therefore, it is recommended that a visual support tool that operationalises the four development stages would be advantageous in facilitating students in capturing their work products from each stage.

Stage 1 - Understand the problem

To understand the problem being solved, it is important that the objective of what the problem is trying to achieve and the goal of the problem in terms of its learning outcomes is understood (Gick, 1986). When a student fully understands the aims and expectations of a problem, this can help push and motivate them to learn the new skills required to achieve those expectations (Hattie, 2012).

However, in the context of software development, novices can often rush to implement a solution to a problem before they fully understand the problem because they find it difficult to separate the conceptualisation of potential solutions from a concrete implementation of those solutions (Fornaro et al., 2006; Kokotovich, 2008). This jump to implementation means that design decisions can be quickly made concrete and communicated through program code, with novices then generally being unable or unwilling to change decisions that have been implemented even if that decision is later found to be incorrect. Equally, novices can engage in a cognitive characteristic known as functional fixedness (Duncker & Lees, 1945). This characteristic means that software novices can see examples of computational constructs being used in one scenario and fail to understand their use in other scenarios. Solutions can often be developed very superficially and with little connection to the actual problem to be solved (Chetty & van der Westhuizen, 2015), and this contributes to an attitude in a novice that the understanding and design of the problem is of little importance relative to the importance of implementing the solution. This view is supported by many educators who observe that getting students to take the time to understand and design solutions rather than immediately engage with syntax to generate a solution through trial and error is very difficult (Blanchard et al., 2020;

Parham et al., 2010; D. R. Wright, 2012). Equally, novices will often memorise solutions to problems and will move from problem exercise to exercise with little reflection on the possible connections between problems or the concepts that may inform them (Garner, 2009). Therefore, the aim of this developmental stage is to encourage students to take the time to understand the nature of the problem and its objectives before moving to trying to solve the problem. This is achieved by employing a subset of skills from Table 10. *Abstraction* is used to produce a high-level summarised version of the problem, with *Pattern Recognition* being used to see if any reusable components can be employed in the solution. This stage also involves the *Mental Modelling* of the problem domain at a high level to ensure that the student understands the problem and its goal or expected outcome.

Stage 2 - Break into tasks

For CS1 learners, the most common approach to solving problems is an informal top-down, depth-first decomposition of the problem (Pearce et al., 2015). Therefore, this stage requires the student to use the problem solving heuristic of divide and conquer (Gick, 1986; Wang & Chiew, 2010), which enables a student to use their understanding of the problem to break it into a hierarchy of subproblems (known as tasks) to help reduce the complexity of the problem. It also requires morphological analysis to understand the requirements of the system.

This breakdown of the problem represents an analysis of the problem (*What do I have to do*?) and employs the skill of *Decomposition* with tasks articulating what must be done as opposed to developing a strategy for how it will be done. This stage will use the skills of *Abstraction* and *Mental Modelling*, as viewing a problem conceptually as a mental model at various levels of detail needs abstraction as a mechanism for reducing complexity (Koppelman & van Dijk, 2010). However, Cabo (2015) noted that students' inability to create such a mental model makes it difficult for them to truly understand the problem domain. To support the creation of this mental model and to prevent cognitive overload, there is ample evidence that visualisation systems assist learners in solving problems and understanding developmental concepts (Edmison & Edwards, 2019; Guo, 2013; Sorva et al., 2013; Vrachnos & Jimoyiannis, 2008). Therefore, to make this stage visual, it is suggested that the support tool scaffolds students in brainstorming tasks by using a tool such as a mind map. Mind mapping can be useful in helping learners to brainstorm problems, and specifically in analysing software problems (Li et al., 2015). In this way, the support tool facilitates learners in utilising *Evaluation* to visually trace backward and forward between the summary of the problem in Stage 1 and the output of this stage to ensure consistency between the stages. *Pattern recognition* is also employed here to ensure the DRY principle is being applied if appropriate.

Stage 3 - Design, code, test and integration

This stage focuses on the design, coding, and testing of solutions for tasks devised in stage 2 and their integration into a full solution. It has been reported in the literature that failure to design a solution (or even understand the rationale for design) can lead to novices adopting maladaptive cognitive practices with no problem planning (Loksa et al., 2016; Prather et al., 2020).

Similar to Stage 1, using a support tool facilitates learners to visually utilise *Abstraction* to move between the tasks identified in Stage 2 and their associated designs in order to ensure consistent mapping between the developmental stages.

The skill of *Pattern Matching* will allow users to know if they can reuse a design or coded solution for all or part of the current problem. Each task that requires a solution to be designed will need to employ the problem-solving heuristics of analogy, means-end analysis, reduction, research, and brainstorming (if peer development or group projects are being employed). The skills of *Mental Modelling, Writing Algorithms, Data Representation* and *Evaluation* will be used to devise and evaluate designs and their associated representation as a programmed solution.

Stage 4 - Final evaluation and reflect on the solution

This final stage tests the integrated coded solution and allows the learner to return to previous developmental stages if they discover logical errors during the testing process. This stage also facilitates students to reflect on their learning journey by encouraging them to engage in metacognition so they can regulate their own learning, which is seen as a key factor in predicting learning performance when problem solving (Jacobse & Harskamp, 2012). Gammill (2006) suggested strategies for promoting metacognition include self-questioning (e.g., "*What do I already know about this topic? How have I solved problems like this before*?"), thinking aloud while performing a task and making graphic representations (e.g., concept maps, flow charts, semantic webs) of one's thoughts and knowledge. In software development, Bergin and Reilly (2005) found that students who perform well use more metacognitive management strategies than lower-performing students. In fact, the more complex a problem is the greater the need for metacognitive control, purposeful reflection, and positive feedback (Havenga et al., 2011). Techniques such as the use of questioning prompts and semantic webs could be included as part of the support tool to encourage and develop metacognition.

This concludes the presentation of findings based on RQ1.1 and RQ1.2 (see Table 3). The next section discusses these findings in order to answer RQ1 (see Table 1) and consequently present the proposed foundational template for an educational software development methodology.

DISCUSSION

This research aimed to answer RQ1 (see Table 1) in order to produce a foundational ESDM template suitable for novice undergraduate learners by first addressing two critical aspects encompassed in RQ1.1 and RQ1.2 (see Table 3). Our findings highlight the necessity of both components in fostering capable and knowledgeable students of software development.

RQ1.1 (see Table 3) focused on identifying the key knowledge and skills necessary for CS1 learners. Key knowledge in software development comprises foundational constructs and concepts that students need to grasp in order to develop software solutions effectively. Identifying this knowledge involved a comprehensive review of existing teaching practices in introductory computer science courses (CS1), supplemented by studies on threshold concepts in computing to compile a recommended list of key knowledge constructs. Given that the topic of threshold concepts, in general, refers to the essential knowledge that students must acquire in a domain of interest, we felt it was appropriate to represent the key knowledge for the ESDM template as threshold concepts. This resulted in four threshold concepts labelled TC1 to TC4 (see Table 6): *State and Sequential Flow, Non-Sequential flow, Modularity,* and *Object Interaction,* with the final concept appearing as an optional concept that can be removed if the underlying paradigm being used is not object based. However, given that our review found that the majority of CS1 courses do use an object-oriented paradigm, it is appropriate to include it in this list. It was also seen in the review that this suite of concepts aligns with the ACM/IEEE/AAAI Computing Curricula, which included the teaching of variables, expressions, and assignments (sequential statements), conditional statements, iterative statements, and modularity. Moreover, it was seen in the review that these constructs are broadly recognized across most CS1 courses. These threshold concepts guide the order in which concepts should be taught without being prescriptive regarding how they should be taught or what programming language should be used. Each concept builds upon the previous one, establishing an order of learning from TC1 to TC3 and optionally TC4 for object-oriented paradigms.

Two comprehensive reviews were undertaken to identify the key skills in teaching CS1 and understanding computational thinking. The findings from the review conducted on CT skills closely aligned with the findings from the teaching CS1 review. This resulted in an agreed suite of seven key skills (see Table 10) - *Abstraction, Data Representation, Decomposition (of problem), Evaluation, Mental Modelling, Pattern recognition,* and *Writing Algorithms*. This alignment between the two reviews validates the importance of these skills in developing competent software developers and supports their inclusion

in the ESDM template. These skills allow the knowledge to be applied to solving problems.

In order to understand the relationship that these key knowledge and skills have to each other and how they can be learned and embodied in the template, it is important to devise a learning framework, which is a model that provides scaffolded approaches to help students "form knowledge structures that are accurately and meaningfully organized" while informing "when and how to apply the skills and knowledge they learn" (Ambrose et al., 2010, p. 45).

Measuring the attainment of competence involves learners using a subset of knowledge and skills for a TC to solve appropriate problems, with the development of solutions being recorded to test that the required knowledge and skills have been applied correctly as learning outcomes. As suggested by the review, learning should incorporate problem exercises that start as well-defined and then move to ill-defined as students move from TC1 to TC4. This means that when a student is being taught a new concept, they will access all the knowledge and application of skills with respect to that concept but will also need to use the knowledge and skills of previous concepts as a prerequisite to mastering the current concept. This sequential learning approach aligns with the spiral curriculum model, ensuring a structured progression in understanding software development concepts and associated skills.

An illustration of the learning framework is presented in Figure 7.

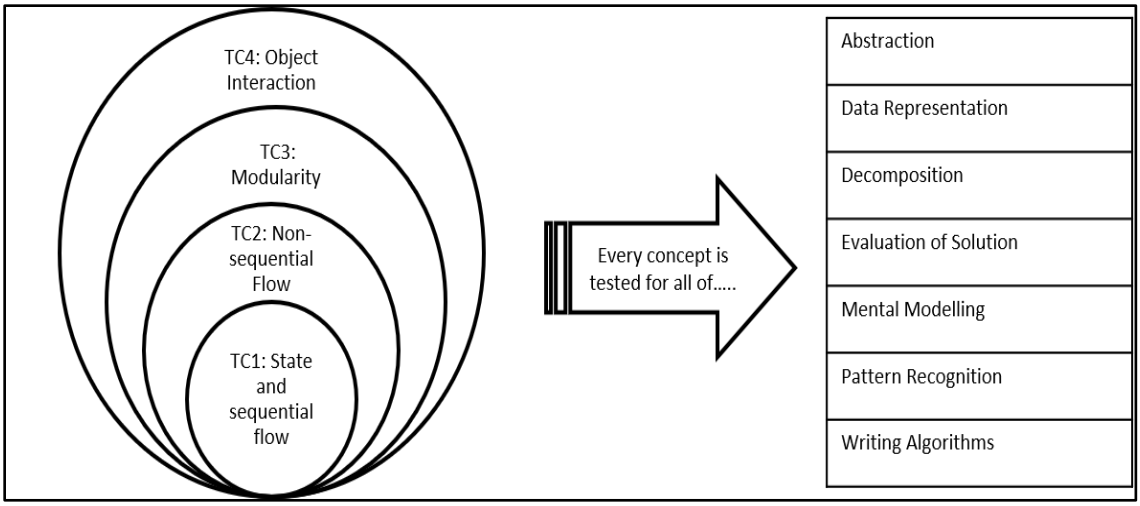

Figure 7. Learning framework consisting of key knowledge and skills
The knowledge is presented as four threshold concepts, with their organisation indicating a spiral curriculum where each concept is an essential part of each subsequent concept. The seven skills, listed alphabetically on the right of the figure, are used to solve problems for each of the four concepts.

In this framework, based on a spiral curriculum, when a student is being taught a new TC, they will be given a sequence of problems appropriate to that TC. They will need to apply all of the seven skills from Figure 7 as part of the ESDM developmental stages in order to solve the problems. This means that each threshold concept will be evaluated via a suite of problems, where evidence of all seven skills being correctly applied must be recorded for each problem. Once all problems for a TC can be completed correctly, a TC is said to be understood and acquired. After mastering a TC, students move to the next TC where once more they cycle through the skills, but they are now also using the knowledge gained from any earlier TC(s).

The choice of specific problems is outside the scope of this research as they would depend on the specific methodology being devised from this ESDM template including knowing the programming paradigm being suggested. However, as part of this review, the importance of contextualizing

learning within real-world scenarios was observed, as it enhances students' ability to transfer classroom knowledge to practical applications. This aligns with the constructivist approach, which advocates for learning experiences that are active, contextual, and reflective. Therefore, providing students with a variety of real-world problems, ranging from well-defined to open-ended (as suggested in the learning framework in the previous sub-section), would help students apply their knowledge in different contexts and develop flexibility in their problem-solving approaches. Moreover, encouraging collaboration and peer review during these stages could also enhance learning outcomes by exposing students to diverse perspectives and solutions.

RQ1.2 focused on the developmental stages required for problem solving with a template ESDM. Our analysis indicates that a methodical approach to problem solving is crucial for instilling good developmental habits and mimicking professional software development practices. The review highlighted the importance of aligning educational practices with industry standards. By integrating principles from established software development methodologies, such as the Software Development Life Cycle (SDLC) and Agile practices, educators can provide students with a more relevant and practical learning experience. However, it is crucial to adapt these methodologies to fit the educational context, ensuring they are suitable for novice learners who may not yet possess the requisite development knowledge.

A staged approach, adapted from Pólya's model and consistent with commercial software development methodologies, was devised for this research. Four stages were suggested, with an initial stage focussed on understanding and defining the problem, followed by a second stage that involved breaking the problem into tasks. The two subsequent stages involve designing and implementing the tasks and, finally, testing and debugging each task while continually reviewing and reflecting on the process and outcomes. This structured progression not only helps students develop systematic problem-solving skills but also reinforces the iterative nature of software development, where continuous improvement and adaptation are key.

Having answered RQ1.1 and RQ1.2, we are now able to answer RQ1, which presents the essential constitutional elements in a foundational ESDM template.

The proposed template serves as a foundational starting point designed to guide researchers and educators in developing concrete methodologies for CS1 learners. It provides a structured learning framework and a set of developmental stages essential for teaching knowledge and skills systematically through scaffolded active learning. The intersection of the learning framework and the developmental stages forms the core of the Educational Software Development Methodology (ESDM) template, where students incrementally solve problems pertinent to each Threshold Concept (TC) using these stages to enhance their understanding and solution-building capabilities. A schematic representation of this foundational template is illustrated in Figure 8.

Instruction is provided in each TC, starting with TC1. As part of the instruction, sample problems and solutions should be provided to students to demonstrate how the knowledge associated with the TC is used in conjunction with the seven skills to solve problems, using the developmental stages to plan and implement solutions. Students will then solve other problems relevant to that TC, supported by the developmental stages, to systematically develop their solutions and to prove competence in that TC. Once all four TCs and their associated problems have been successfully completed, the student is considered to have fulfilled the requirements of a typical CS1 curriculum. Students can then continue to use the developmental stages as the basis for solving other problems outside of CS1. It should be noted that the ESDM template can also be solely used as a software development methodology (i.e., it is not invoking the learning framework), where learners are able to provide their own specification for a problem and use the developmental stages to systematically produce their solution.

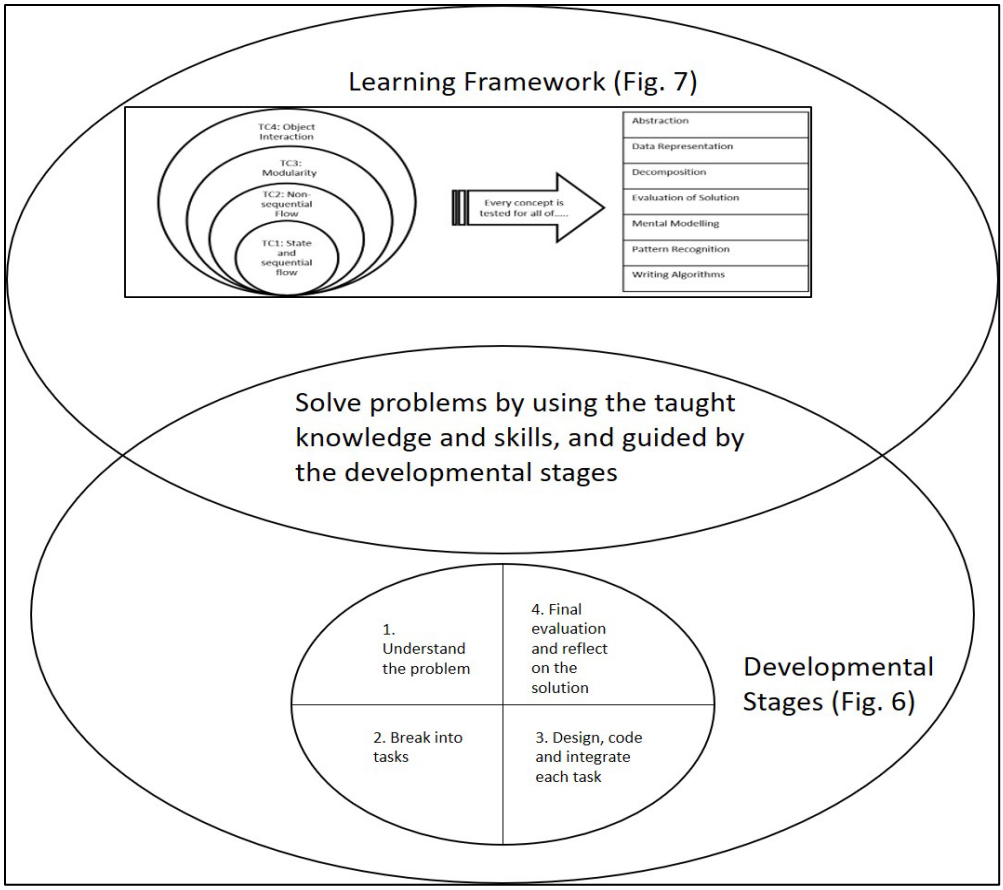

Figure 8. The proposed foundational template for an ESDM which contains a learning framework and four developmental stages which intersect to provide CS1 students with a structured environment to learn how to develop software solutions

It should be noted that while this paper lays out the essential elements of the ESDM, it is beyond its scope to provide concrete examples of its application. The template is intended to be a flexible and adaptable starting point for other researchers to build upon and develop specific methodologies tailored to their educational contexts. Future research should focus on creating detailed methodologies and practical applications of the ESDM template, ensuring that they are robust and adaptable to diverse learning environments. Furthermore, as discussed in the skills identification section, the use of visualization systems is recommended to help students model and conceptualize their problem-solving processes. To this end, developing a software tool that operationalizes the developmental stages of the ESDM would be highly beneficial. Such a tool would enable students to explicitly capture their understanding of problems, break down problems into manageable tasks, design and code these tasks, evaluate their solutions, and track their workflow through the various stages. This tool would not only enhance the learning experience but also provide valuable insights into students' problem-solving approaches, aiding educators in refining and improving the methodology.

LIMITATIONS

This integrative review attempted to be as wide-ranging as possible in reviewing problem solving, CS1 education, computational thinking, and threshold concepts. In order to do this, comprehensive reviews of both a narrative and systematic nature were carried out on material primarily published from 2000 to 2020 on a range of research databases and libraries. However, as extensive as these searches were, it cannot be guaranteed that all relevant literature was reviewed, especially in the

narrative reviews of CS1 education and threshold concepts. Also, the decision just to review CS1 education and exclude research into non-computing majors studying software development or computing studies at first or second-level education may have resulted in some data being omitted.

It is also recognised that the ESDM template that was produced would need refinement by educators who wish to use it in determining the problem sets that would be used with the methodology and in providing a visual support tool. This ESDM, however, does give a comprehensive description of its components and, therefore, is a useful starting point for interested educators and researchers to adopt, adapt, and use in their own practice.

CONCLUSIONS

Despite a wealth of research into the teaching and learning of software development and computer science over the past 30 years, work is still required to identify the right blend of technology and pedagogy to help progress both student success and retention (Price & Smith, 2014; Zarb et al., 2018). Given the importance of undergraduate students forming good developmental habits and competencies (Dorodchi et al., 2019), we were motivated to immerse ourselves in this area as we saw an anomalous situation in existence that concerned the lack of software development methodologies suitable for novice undergraduate education. Consequently, the purpose of this integrative review was to create a template for an educational software development methodology that could be refined and operationalised by researchers and educators.

Devising this template was achieved by addressing two research sub-questions (see Table 3), which encompass the critical aspects of educating novice software developers: understanding the key skills and knowledge required by students, and the structuring of problem-solving developmental stages. To answer these questions, a combination of systematic and narrative literature reviews were undertaken in the following four areas: (1) *teaching CS1*, (2) *problem solving*, (3) *computational thinking*, and (4) *threshold concepts*. The findings from these reviews produced seven skills and four concepts required by novice learners. The skills include the ability to perform *abstraction, data representation, decomposition, evaluation, mental modelling, pattern recognition,* and *writing algorithms*. The concepts included *state and sequential flow, non-sequential flow control, modularity,* and *object interaction* (if the underlying paradigm is object based). Once the key knowledge and skills were identified, the next step was to create a learning framework into which these concepts and skills were integrated. The result was a spiral framework with the four concepts taught in order where problems pertinent to each concept are incrementally solved using all of the seven skills (see Figure 7).

The learning framework was married with four development stages, devised from the research, to guide software problem solving – (1) *understand the problem*; (2) *break into tasks*; (3) *design, code, test, and integrate*; and (4) *final evaluation and reflect on the solution* – in order to provide the foundational template for educational methodologies. This template provides a structured learning framework that incorporates scaffolded instruction and practical application. Integrating problem solving developmental stages with the learning framework provides students with a clear roadmap for applying their skills systematically. Moreover, this approach aligns with the principles of experiential learning, where students learn through reflection on doing. By methodically guiding students through the stages of problem solving, educators can help them develop the habits and skills necessary for professional software development. The learning framework is designed to have a temporary existence that slowly fades as learners become more proficient in software development. This means that the ESDM can continue to be used by students once they get past the CS1 stage of their studies.

While older examples of ESDMs to support different aspects of developing software were found in the literature, no methodology or subsequent process aimed at CS1 learners to support all aspects of developing software solutions in any programming paradigm was found. This makes this proposed template a vital initial step toward creating comprehensive methodologies for teaching CS1 students. It outlines the necessary components and structure but requires further development and

customization to become a fully realized educational tool. It is our belief that the provision of this template methodology provides a basis for other educators and researchers to build on this research and create such tools. To achieve this, the developmental stages are at a sufficiently high level, so their realisation in an operational methodology can be adjusted given the context of the nature and structure of the CS1 course.

However, it should be stated that the development and application of the proposed template into a concrete methodology presents some challenges. First, while the template offers a foundational framework for teaching CS1 students, translating this high-level structure into concrete, actionable methodologies that can be effectively implemented in diverse undergraduate settings is a complex task. Researchers and educators must navigate the variability in student backgrounds and create problem sets suitable for each TC stage and in alignment with their underlying programming paradigm, which necessitates the customization of the template to suit specific contexts. Additionally, creating a visual software tool to operationalize the ESDM, as recommended, involves technical challenges, such as designing intuitive interfaces that can accurately model students' problem solving while providing meaningful feedback. Finally, evaluating the effectiveness of these methodologies and tools in enhancing CS1 education requires robust, long-term empirical studies, which can be resource-intensive and logistically challenging. These challenges underscore the need for collaborative efforts among educators, software developers, and researchers to refine and validate the proposed template in practical settings.

Despite these challenges, it is important to emphasise that the template itself provides a robust and comprehensive foundation. The learning framework, coupled with the developmental stages outlined in the template, offers clear guidance on how to systematically approach the teaching and learning of CS1 constructs. The intersection of these components ensures that researchers have a well-defined pathway to follow, enabling them to adapt the template to various educational contexts with confidence. Moreover, the template's flexibility allows for iterative refinement, meaning that as researchers apply the framework, they can continuously adjust and optimize their methodologies based on empirical feedback and specific classroom dynamics. Thus, while the creation of customized methodologies will require effort and adaptation, the template's inherent structure and clarity serve as a reliable map, guiding researchers through the process of developing effective and context-sensitive educational strategies for CS1 learners. Therefore, researchers and educators are encouraged to use this template as a foundation for developing and testing their own methodologies, contributing to the ongoing improvement of computer science education.

REFERENCES

Abbas, N., Gravell, A. M., & Wills, G. B. (2008). Historical roots of agile methods: Where did "Agile thinking" come from? In P. Abrahamsson, R. Baskerville, K. Conboy, B. Fitzgerald, L. Morgan, & X. Wang (Eds.), *Agile processes in software engineering and extreme programming* (pp. 94-103). Springer. https://doi.org/10.1007/978-3-540-68255-4_10

Agbo, F. J., Oyelere, S. S., Suhonen, J., & Adewumi, S. (2019). A systematic review of computational thinking approach for programming education in higher education institutions. *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (pp. 1-10). Association for Computing Machinery. https://doi.org/10.1145/3364510.3364521

Alexander, G., & Renshaw, B. (2005). *Supercoaching*. Business Books.

Allan, V., Barr, V., Brylow, D., & Hambrusch, S. (2010). Computational thinking in high school courses. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 390-391). Association for Computing Machinery. https://doi.org/10.1145/1734263.1734395

Al-Saqqa, S., Sawalha, S., & Abdel-Nabi, H. (2020). Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies*, *14*(11), 147-170. https://doi.org/10.3991/ijim.v14i11.13269

Alston, P., Walsh, D., & Westhead, G. (2015). Uncovering "threshold concepts" in web development: An instructor perspective. *ACM Transactions on Computing Education*, *15*(1), Article 2. https://doi.org/10.1145/2700513

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

Armaya'u, Z. U., Gumel, M. M., & Tuge, H. S. (2022). Comparing flowchart and swim lane activity diagram for aiding transitioning to object-oriented implementation. *American Journal of Education and Technology*, *1*(2), 99-106. https://doi.org/10.54536/ajet.v1i2.612

Armoni, M. (2014). Spiral thinking: K-12 computer science education as part of holistic computing education. *ACM Inroads*, *5*(2), 31-33. https://doi.org/10.1145/2614512.2614521

Arnold, R., Langheinrich, M., & Hartmann, W. (2007). InfoTraffic: Teaching important concepts of computer science and math through real world examples. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 105-109). Association for Computing Machinery. https://doi.org/10.1145/1227310.1227349

Barr, D., Harrison, J., & Conery, L. (2011). Computational thinking: A digital age skill for everyone. *Learning & Leading with Technology*, *38*(6), 20-23. https://eric.ed.gov/?id=EJ918910

Barr, V., & Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 48-54. https://doi.org/10.1145/1929887.1929905

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming: 12 years later. *ACM Inroads*, *10*(2), 30-36. https://doi.org/10.1145/3324888

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 411-415). Association for Computing Machinery. https://doi.org/10.1145/1047124.1047480

Berry, M., & Kölling, M. (2013). The design and implementation of a notional machine for teaching introductory programming. *Proceedings of the 8th Workshop in Primary and Secondary Computing Education* (pp. 25-28). Association for Computing Machinery. https://doi.org/10.1145/2532748.2532765

Biju, S. M. (2013). Difficulties in understanding object-oriented programming concepts. In K. Elleithy, & T. Sobh (Eds.), *Innovations and advances in computer, information, systems sciences, and engineering* (pp. 319-326). Springer. https://doi.org/10.1007/978-1-4614-3535-8_27

Blanchard, J., Gardner-McCune, C., & Anthony, L. (2020). Dual-modality instruction and learning: A case study in CS1. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 818-824). Association for Computing Machinery. https://doi.org/10.1145/3328778.3366865

Boehm, B. (2006). A view of 20th and 21st century software engineering. *Proceedings of the 28th International Conference on Software Engineering* (pp. 12-29). Association for Computing Machinery. https://doi.org/10.1145/1134285.1134288

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: Do they exist and are they useful? *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 504-508). Association for Computing Machinery. https://doi.org/10.1145/1227310.1227482

Bower, M., & Falkner, K. (2015). Computational thinking, the notional machine, pre-service teachers, and research opportunities. *Australian Computer Science Communications*, *37*(2), 37-46. https://crpit.scem.western-sydney.edu.au/confpapers/CRPITV160Bower.pdf

Boyd, J. R. (1996). The essence of winning and losing. *Unpublished lecture notes*, 12(23), 123-125.

Bransford, J. D., & Stein, B. S. (1993). *The ideal problem solver*. Centre for Teaching and Technology Book Library. https://digitalcommons.georgiasouthern.edu/ct2-library/46

Bruner, J. S. (1960). *The process of education*. Vintage Books. https://doi.org/10.3726/978-1-4539-1735-0/12

Bustard, D., Wilkie, G., & Greer, D. (2013, April). The maturation of Agile software development principles and practice: Observations on successive industrial studies in 2010 and 2012. *Proceedings of the 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, Scottsdale, AZ, USA,* 139-146. IEEE. https://doi.org/10.1109/ecbs.2013.11

Butt, S. A., Misra, S., Anjum, M. W., & Hassan, S. A. (2021). Agile project development issues during COVID-19. In A. Przybyłek, J. Miler, A. Poth, & A. Riel (Eds.), Lean and agile software development (pp. 59-70). Springer. https://doi.org/10.1007/978-3-030-67084-9_4

Cabezas, I., Segovia, R., Caratozzolo, P., & Webb, E. (2020, October). Using software engineering design principles as tools for freshman students learning. *Proceedings of the IEEE Frontiers in Education Conference, Uppsala, Sweden,* 1-5.. https://doi.org/10.1109/fie44824.2020.9274177

Cabo, C. (2015, June). Quantifying student progress through Bloom's taxonomy cognitive categories in computer programming courses. *Paper presented at 2015 ASEE Annual Conference & Exposition, Seattle, Washington.* https://doi.org/10.18260/p.24632

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies*, *40*(5), 795-811. https://doi.org/10.1006/ijhc.1994.1038

Caserta, P., & Zendra, O. (2011). Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualizations and Computer Graphics*, *17*(7), 913-933. https://doi.org/10.1109/tvcg.2010.110

Caspersen, M. E., & Kolling, M. (2009). STREAM: A first programming process. *ACM Transactions on Computing Education*, 9(1), Article 4. https://doi.org/10.1145/1513593.1513597

Castro, F. E. V. G. (2015). Investigating novice programmers' plan composition strategies. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 249-250). Association for Computing Machinery. https://doi.org/10.1145/2787622.2787735

Chetty, J., & van der Westhuizen, D. (2015). Towards a pedagogical design for teaching novice programmers: Design-based research as an empirical determinant for success. *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (pp. 5-12). Association for Computing Machinery. https://doi.org/10.1145/2828959.2828976

Coffey, J. W. (2015). Relationship between design and programming skills in an advanced computer programming class. *Journal of Computer Sciences in Colleges, 30*(5), 39-45.

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. *Proceedings of the Fourteenth Australasian Computing Education Conference* (pp. 77-86). Australian Computer Society.

Cousin, G. (2006). An introduction to threshold concepts. *Planet, 17*(1), 4-5. https://doi.org/10.11120/plan.2006.00170004

Cronin, M. A., & George, E. (2023). The why and how of the integrative review. *Organizational Research Methods, 26*(1), 168-192. https://doi.org/10.1177/1094428120935507

Denning, P. J. (2017). Remaining trouble spots with computational thinking: Addressing unresolved questions concerning computational thinking. *Communications of the ACM, 60*(6), 33-39. https://doi.org/10.1145/2998438

Denning, P. J., & Tedre, M. (2019). *Computational thinking.* MIT Press. https://doi.org/10.4324/9781003102991-1

De Raadt, M., Watson, R., & Toleman, M. (2009). Teaching and assessing programming strategies explicitly. *Proceedings of the 11th Australasian Computing Education Conference* (pp. 45-54). Australian Computer Society.

Devedzic, V. (2011). Teaching agile software development: A case study. *IEEE Transactions on Education, 54*(2), 273-278. https://doi.org/10.1109/te.2010.2052104

Dickson, P. E., Brown, N. C., & Becker, B. A. (2020). Engage against the machine: Rise of the notional machines as effective pedagogical devices. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 159-165). Association for Computing Machinery. https://doi.org/10.1145/3341525.3387404

Dijkstra, E. (1979). Programming considered as a human activity. *Classics in Software Engineering* (pp. 1-9). Yourdon Press. https://doi.org/10.7551/mitpress/11740.001.0001

Dorodchi, M., Al-Hossami, E., Nagahisarchoghaei, M., Diwadkar, R. S., & Benedict, A. (2019, October). Teaching an undergraduate software engineering course using active learning and open source projects. *Proceedings of the IEEE Frontiers in Education Conference, Covington, KY, USA,* 1-5. https://doi.org/10.1109/fie43999.2019.9028517

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

Duncker, K., & Lees, L. S. (1945). On problem solving. *Psychological Monographs, 58*(5), i-113. https://doi.org/10.1037/h0093599

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2006). Putting threshold concepts into context in computer science education. *ACM SIGCSE Bulletin, 38*(3), 103-107. https://doi.org/10.1145/1140124.1140154

Edmison, B., & Edwards, S. H. (2019). Experiences using heat maps to help students find their bugs: Problems and solutions. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 260-266). Association for Computing Machinery. https://doi.org/10.1145/3287324.3287474

Edwards, J., Ditton, J., Trninic, D., Swanson, H., Sullivan, S., & Mano, C. (2020). Syntax exercises in CS1. *Proceedings of the ACM Conference on International Computing Education Research* (pp. 216-226). Association for Computing Machinery. https://doi.org/10.1145/3372782.3406259

Ericsson, K. A., & Charness, N. (1997). Cognitive and developmental factors in expert performance. In P. J. Feltovich, K. M. Ford, & R. R. Hoffman (Eds.), *Expertise in context: Human and machine* (pp. 3–41). American Association for Artificial Intelligence; MIT Press. https://doi.org/10.1017/cbo9780511816796

Feaster, Y., Ali, F., & O Hallstrom, J. (2012). Serious toys: Teaching the binary number system. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (pp. 262–267). Association for Computing Machinery. https://doi.org/10.1145/2325296.2325358

Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., Du Boulay, B., Hauswirth, M., & Petersen, A. (2020). Capturing and characterising notional machines. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 502-503). Association for Computing Machinery. https://doi.org/10.1145/3341525.3394988

Floyd, R. W. (2007). The paradigms of programming. *ACM Turing award lectures*. Association for Computing Machinery. https://doi.org/10.1145/1283920.1283934

Fornaro, R. J., Heil, M. R., & Tharp, A. L. (2006, April). What clients want - what students do: Reflections on ten years of sponsored senior design projects. *Proceedings of the 19th Conference on Software Engineering Education & Training, Turtle Bay, HI, USA,* 226-236. https://doi.org/10.1109/cseet.2006.40

Foster, E. C. (2014). *Software engineering: A methodical approach*. https://doi.org/10.1007/978-1-4842-0847-2_1

Frison, P. (2015). A teaching assistant for algorithm construction. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 9-14). Association for Computing Machinery. https://doi.org/10.1145/2729094.2742588

Gammill, D. M. (2006). Learning the write way. *The Reading Teacher, 59*(8), 754-762. https://doi.org/10.1598/rt.59.8.3

Garner, S. (2007). A program design tool to help novices learn programming. *ICT: Providing Choices for Learners and Learning. Proceedings of ASCILITE, Singapore* (pp. 321-324). https://ascilite.org/conferences/singapore07/procs/garner.pdf

Garner, S. (2009). A quantitative study of a software tool that supports a part-complete solution method on learning outcomes. *Journal of Information Technology Education, 8*, 285-310. https://doi.org/10.28945/698

Gick, M. L. (1986). Problem solving strategies. *Educational Psychologist, 21*(1-2), 99-120. https://doi.org/10.1080/00461520.1986.9653026

Gilb, T. (1985). Evolutionary delivery versus the "waterfall model." *SIGSOFT Software Engineering Notes*, *10*(3), 49-61. https://doi.org/10.1145/1012483.1012490

Ginat, D., & Menashe, E. (2015). SOLO taxonomy for assessing novices' algorithmic design. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 452-457). Association for Computing Machinery. https://doi.org/10.1145/2676723.2677311

Gouws, L. A., Bradshaw, K., & Wentworth, P. (2013). Computational thinking in educational activities: An evaluation of the educational game Light-Bot. *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 10-15). Association for Computing Machinery. https://doi.org/10.1145/2462476.2466518

Greer, T., Hao, Q., Jing, M., & Barnes, B. (2019). On the effects of active learning environments in computing education. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (pp. 267-272). Association for Computing Machinery. https://doi.org/10.1145/3287324.3287345

Grover, S. (2019). Thinking about computational thinking: Lessons from education research. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (p. 1283). Association for Computing Machinery. https://doi.org/10.1145/3287324.3293763

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, *42*(1), 38-43. https://doi.org/10.3102/0013189x12463051

Guo, P. J. (2013). Online Python tutor: Embeddable web-based program visualization for CS education. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (pp. 579-584). Association for Computing Machinery. https://doi.org/10.1145/2445196.2445368

Hattie, J. (2012). *Visible learning for teachers: Maximizing impact on learning*. Routledge. https://doi.org/10.1080/02667363.2012.693677

Havenga, M., Mentz, E., & De Villiers, R. (2011). Thinking processes used by high-performing students in a computer programming task. *The Journal for Transdisciplinary Research in Southern Africa, 7*(1), 25-40. https://doi.org/10.4102/td.v7i1.252

Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to teaching computer science: An activity-based approach*. Springer. https://doi.org/10.1007/978-0-85729-443-2

Hertz, M. (2010). What do "CS1" and "CS2" mean? Investigating differences in the early courses. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 199-203). Association for Computing Machinery. https://doi.org/10.1145/1734263.1734335

Higgins, C. (2021). *The design and evaluation of an educational software development process for first year computing undergraduates* [Doctoral dissertation, Technological University Dublin]. https://arrow.tudublin.ie/sciendoc/257/

Higgins, C., Mtenzi, F., Hanratty, O., & McAvinia, C. (2017a). A conceptual framework for a software development process based on computational thinking. *Proceedings of the 11th International Technology, Education and Development Conference, Valencia, Spain,* 455-464. https://doi.org/10.21125/inted.2017.0244

Higgins, C., Mtenzi, F., O'Leary, C., Hanratty, O., & Mcavinia, C. (2017b). A software development process for freshman undergraduate students. In A. Tatnall, & M. Webb (Eds.), *Tomorrow's learning: Involving everyone. Learning with and about technologies and computing* (pp. 599-608). Springer. https://doi.org/10.1007/978-3-319-74310-3_60

Holloway, M., Alpay, E., & Bull, A. (2010). A quantitative approach to identifying threshold concepts in engineering education. *Journal of Engineering Education 2010: Inspiring the Next Generation of Engineers.* https://openresearch.surrey.ac.uk/esploro/outputs/99516906902346

Hu, M., Winikoff, M., & Cranefield, S. (2013). A process for novice programming using goals and plans. *Proceedings of the Fifteenth Australasian Computing Education Conference* (pp. 3-12). Australian Computer Society. https://www.researchgate.net/publication/266081141_A_Process_for_Novice_Programming_Using_Goals_and_Plans

Huang, T.-C., Shu, Y., Chen, C.-C., & Chen, M.-Y. (2013). The development of an innovative programming teaching framework for modifying students' maladaptive learning pattern. *International Journal of Information and Education Technology, 3*(6), 591-596. https://doi.org/10.7763/ijiet.2013.v3.342

Hummel, H. G. K. (2006). Feedback model to support designers of blended learning courses. *The International Review of Research in Open and Distributed Learning, 7*(3). https://doi.org/10.19173/irrodl.v7i3.379

Izu, C., & Weerasignhe, A. (2020). Assessing CS1 design skills with a string manipulation task. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 432-438). Association for Computing Machinery. https://doi.org/10.1145/3341525.3387382

Jacobse, A. E., & Harskamp, E. G. (2012). Towards efficient measurement of metacognition in mathematical problem solving. *Metacognition and Learning, 7*(2), 133-149. https://doi.org/10.1007/s11409-012-9088-x

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). The unified process. *IEEE Software, 16*(3), 96.

Jaime, A., Blanco, J. M., Domínguez, C., Sánchez, A., Heras, J., & Usandizaga, I. (2016). Spiral and project-based learning with peer assessment in a computer science project management course. *Journal of Science Education and Technology, 25*, 439-449. https://doi.org/10.1007/s10956-016-9604-x

Jeff, B., & Nguyen, K. (2018, December). ADL – Algorithmic design language. Proceedings of the *International Conference on Computational Science and Computational Intelligence, Las Vegas, NV, USA,* 651-654. https://doi.org/10.1109/CSCI46756.2018.00130

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development, 48*(4), 63-85. https://doi.org/10.1007/bf02300500

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 107-111). Association for Computing Machinery. https://doi.org/10.1145/1734263.1734299

Kalelioglu, F., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing, 4*(3), 583-594. http://acikeri-sim.baskent.edu.tr/handle/11727/3831

Khalife, J. T. (2006, June). Threshold for the introduction of programming: Providing learners with a simple computer model. *Proceedings of the 28th International Conference on Information Technology Interfaces*, *Cavtat, Croatia,* 71-76. https://doi.org/10.1109/ITI.2006.1708454

Kinnunen, P., & Malmi, L. (2006). Why students drop out CS1 course? *Proceedings of the Second International Workshop on Computing Education Research* (pp. 97-108). Association for Computing Machinery. https://doi.org/10.1145/1151588.1151604

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist, 41*(2), 75-86. https://doi.org/10.1207/s15326985ep4102_1

Kohn, T. (2017). Variable evaluation: An exploration of novice programmers' understanding and common misconceptions. *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 345-350). Association for Computing Machinery. https://doi.org/10.1145/3017680.3017724

Kokotovich, V. (2008). Problem analysis and thinking tools: An empirical study of non-hierarchical mind mapping. *Design Studies, 29*(1), 49-69. https://doi.org/10.1016/j.destud.2007.09.001

Koppelman, H., & van Dijk, B. (2010). Teaching abstraction in introductory courses. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (pp. 174-178). https://doi.org/10.1145/1822090.1822140

Koulouri, T., Lauria, S., & Macredie, R. D. (2015). Teaching introductory programming: A quantitative evaluation of different approaches. *ACM Transactions on Computing Education, 14*(4), Article 26. https://doi.org/10.1145/2662412

Kramer, J., & Hazzan, O. (2006). The role of abstraction in software engineering. *ACM SIGSOFT Software Engineering Notes, 31*(6), 38-39. https://doi.org/10.1145/1218776.1226833

Krishnamurthi, S., & Fisler, K. (2019). Programming paradigms and beyond. In S. A. Fincher, & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 377-413). Cambridge University Press. https://doi.org/10.1017/9781108654555.014

Kumar, A. N., Raj, R. K., Aly, S. G., Anderson, M. D., Becker, B. A., Blumenthal, R. L., Eaton, E., Epstein, S. L., Goldweber, M., Jalote, P., Lea, D., Oudshoorn, M., Pias, M., Reiser, S., Servin, C., Winters, T., & Xiang, Q. (2023). *Computer Science Curricula 2023*. Association for Computing Machinery. https://doi.org/10.1145/3664191

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin, 37*(3), 14-18. https://doi.org/10.1145/1151954.1067453

Li, C. L., Yang, L. P., & Wang, W. (2015). Application of mind mapping to improve the teaching effect of Java program design course. In H.-C. Liu, W.-P. Sung, & W. Yao (Eds.), *Computing, control, information and education engineering* (p. 451). CRC Press. https://doi.org/10.1201/b18828-101

Liikkanen, L. A., & Perttula, M. (2009). Exploring problem decomposition in conceptual design among novice designers. *Design Studies, 30*(1), 38-59. https://doi.org/10.1016/j.destud.2008.07.003

Lishinski, A., Yadav, A., Enbody, R., & Good, J. (2016). The influence of problem-solving abilities on students' performance on different assessment tasks in CS1. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 329-334). Association for Computing Machinery. https://doi.org/10.1145/2839509.2844596

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In J. Hamer & M. de Raadt (Eds.), *Conferences in research and practice in information technology series* (pp. 9-18). Australian Computer Society. https://opus.lib.uts.edu.au/handle/10453/17580#

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., & Thomas, L. (2004). A multinational 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. *ACM SIGCSE Bulletin, 38*(3), 118-122. https://doi.org/10.1145/1140123.1140157

Liu, B., & He, J. (2014). Teaching mode reform and exploration on the University Computer Basic based on Computational Thinking training in Network Environment. *Proceedings of the 9th International Conference on Computer Science & Education, Vancouver, BC, Canada,* 59-62. https://doi.org/10.1109/iccse.2014.6926430

Lockwood, J., & Mooney, A. (2018). Computational thinking in secondary education: Where does it fit? A systematic literary review. *International Journal of Computer Science Education in Schools*, *2*(1), 41-60. https://doi.org/10.21585/ijcses.v2i1.26

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

Lu, J. J., & Fletcher, G. H. L. (2009). Thinking about computational thinking. *ACM SIGCSE Bulletin, 41*(1), 260-264. https://doi.org/10.1145/1539024.1508959

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., & Szabo, C. (2018). Introductory programming: A systematic literature review. *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 55-106). Association for Computing Machinery. https://doi.org/10.1145/3293881.3295779

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

Mahatanankoon, P., & Wolf, J. (2021). Cognitive learning strategies in an introductory computer programming course. *Information Systems Education Journal, 19*(3), 11-20. https://eric.ed.gov/?id=EJ1301236

Margulieux, L. E., Morrison, B. B., & Decker, A. (2019). Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 548-554). Association for Computing Machinery. https://doi.org/10.1145/3304221.3319756

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American Psychologist, 59*(1), 14-19. https://doi.org/10.1037/0003-066x.59.1.14

Mendelsohn, P., Green, T. R. G., & Brna, P. (1991). Programming languages in education: The search for an easy start. In J. E. Anderson, & J. N. Finlay (Eds.), *Psychology of programming* (pp. 175-200). Elsevier. https://doi.org/10.1016/b978-0-12-350772-3.50016-1

Mendonça, A., de Oliveira, C., Guerrero, D., & Costa, E. (2009, October). Difficulties in solving ill-defined problems: A case study with introductory computer programming students. *Proceedings of the 39th IEEE Frontiers in Education Conference, San Antonio, TX, USA,* 1-6. https://doi.org/10.1109/fie.2009.5350628

Meyer, D. L. (2009). The poverty of constructivism. *Educational Philosophy and Theory, 41*(3), 332-341. https://doi.org/10.1111/j.1469-5812.2008.00457.x

Meyer, J. H. F., & Land, R. (2003). Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practicing within the disciplines. In C. Rust (Ed.), *Improving student learning: Theory and practice – ten years on* (pp. 412-424). OCSLD.

Meyer, J. H. F., & Land, R. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education, 49*(3), 373-388. https://doi.org/10.1007/s10734-004-6779-5

Middleton, D. (2012). Trying to teach problem solving instead of just assigning it: Some practical issues. *Journal of Computing Sciences in Colleges, 27*(5), 60-65. https://dl.acm.org/doi/abs/10.5555/2168874.2168891

Mohanty, R., & Bala Das, S. (2018). A proposed what-why-how (WWH) learning model for students and strengthening learning skills through computational thinking. In P. K. Sa, M. N. Sahoo, M. Murugappan, Y. Wu, & B. Majhi (Eds.), *Progress in intelligent computing techniques: Theory, practice, and applications* (pp. 135-141). Springer. https://doi.org/10.1007/978-981-10-3376-6_15

Moher, D., Liberati, A., Tetzlaff, J., Altman, D. G., & The PRISMA Group. (2009). Reprint – Preferred reporting items for systematic reviews and meta-analyses: The PRISMA statement. *Physical Therapy, 89*(9), 873-880.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. *Proceedings of the Working Conference on Advanced Visual Interfaces* (pp. 373-376). Association for Computing Machinery. https://doi.org/10.1145/989863.989928

Morgado, C., & Barbosa, F. (2012). A structured approach to problem solving in CS1. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (p. 399). Association for Computing Machinery. https://doi.org/10.1145/2325296.2325401

Mornar, J., Granić, A., & Mladenović, S. (2014). System for automatic generation of algorithm visualizations based on pseudocode interpretation. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 27-32). Association for Computing Machinery. https://doi.org/10.1145/2591708.2591743

Napoleão, B. M., Petrillo, F., & Hallé, S. (2020, October). Open source software development process: A systematic review. *Proceedings of the IEEE 24th International Enterprise Distributed Object Computing Conference, Eindhoven, Netherlands,* 135-144. https://doi.org/10.1109/edoc49727.2020.00025

Neto, V. L., Coelho, R., Leite, L., Guerrero, D. S., & Mendonça, A. P. (2013, May). POPT: A problem-oriented programming and testing approach for novice students. *Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA,* 1099-1108. https://doi.org/10.1109/icse.2013.6606660

O'Donnell, R. (2010). A critique of the threshold concept hypothesis and an application in economics. *Working Paper Series 164*, Finance Discipline Group, UTS Business School, University of Technology, Sydney, Australia. https://ideas.repec.org/p/uts/wpaper/164.html

Palts, T., & Pedaste, M. (2020). A model for developing computational thinking skills. *Informatics in Education, 19*(1), 113-128. https://doi.org/10.15388/infedu.2020.06

Panigrahi, C. R., Mall, R., & Pati, B. (2021). Software development methodology for cloud computing and its impact. In Information Resources Management Association (Ed.), *Research anthology on recent trends, tools, and*

implications of computer programming (pp. 151-172). IGI Global. https://doi.org/10.4018/978-1-7998-3016-0.ch008

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning, 1*, 95-123. https://doi.org/10.1007/bf00191473

Parham, J., Gugerty, L., & Stevenson, D. E. (2010). Empirical evidence for the existence and uses of metacognition in computer science problem solving. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 416-420). Association for Computing Machinery. https://doi.org/10.1145/1734263.1734406

Pearce, J. L., Nakazawa, M., & Heggen, S. (2015). Improving problem decomposition ability in CS1 through explicit guided inquiry-based instruction. *Journal of Computing Sciences in Colleges, 31*(2), 135-144. https://dl.acm.org/doi/10.5555/2831432.2831453

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop CS1. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 71-80). Association for Computing Machinery. https://doi.org/10.1145/2999541.2999552

Pólya, G. (1957). *How to solve it* (2nd ed.). Princeton University Press. https://doi.org/10.2307/3609122

Prather, J., Becker, B. A., Craig, M., Denny, P., Loksa, D., & Margulieux, L. (2020). What do we think we think we are doing? Metacognition and self-regulation in programming. *Proceedings of the ACM Conference on International Computing Education Research* (pp. 2-13). Association for Computing Machinery. https://doi.org/10.1145/3372782.3406263

Price, K., & Smith, S. (2014). Improving student performance in CS1. *Journal of Computing Sciences in Colleges, 30*(2), 157-163. https://dl.acm.org/doi/10.5555/2667432.2667454

Prince, M. (2004). Does active learning work? A review of the research. *Journal of Engineering Education, 93*(3), 223-231. https://doi.org/10.1002/j.2168-9830.2004.tb00809.x

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education, 18*(1), Article 1. https://doi.org/10.1145/3077618

Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ViLLE tool. *Journal of Information Technology Education, 7*, 15-32. https://doi.org/10.28945/3237

Reade, C. (1989). *Elements of functional programming*. Addison-Wesley. https://dl.acm.org/doi/abs/10.5555/113909

Robins, A. V. (2019). Novice programmers and introductory programming. In S. A. Fincher, & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 327-376). Cambridge University Press. https://doi.org/10.1017/9781108654555.013

Rogerson, C., & Scott, E. (2010). The fear factor: How it affects students learning to program in a tertiary environment. *Journal of Information Technology Education Research, 9*, 147-171. https://doi.org/10.28945/1183

Rountree, J., & Rountree, N. (2009). Issues regarding threshold concepts in computer science. *Proceedings of the Eleventh Australasian Conference on Computing Education* (pp. 139-146). Association for Computing Machinery. https://doi.org/10.5555/1862712.1862733

Rowbottom, D. P. (2007). Demystifying threshold concepts. *Journal of Philosophy of Education, 41*(2), 263-270. https://doi.org/10.1111/j.1467-9752.2007.00554.x

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. *Proceedings of the 9th International Conference on Software Engineering* (pp. 328-338). Association for Computing Machinery. https://dl.acm.org/doi/10.5555/41765.41801

Safari, Y., & Meskini, H. (2016). The effect of metacognitive instruction on problem solving skills in Iranian students of health sciences. *Global Journal of Health Science, 8*(1), 150-156. https://doi.org/10.5539/gjhs.v8n1p150

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L., & Zander, C. (2008). Student understanding of object-oriented programming as expressed in concept maps. *Proceedings of the 39th*

SIGCSE Technical Symposium on Computer Science Education (pp. 332-336). Association for Computing Machinery. https://doi.org/10.1145/1352135.1352251

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L., & Zander, C. (2012). Threshold concepts and threshold skills in computing. *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (pp. 23-30). Association for Computing Machinery. https://doi.org/10.1145/2361276.2361283

Santana, B. L., & Bittencourt, R. A. (2018, October). Increasing motivation of CS1 non-majors through an approach contextualized by games and media. *Proceedings of the IEEE Frontiers in Education Conference, San Jose, CA, USA,* 1-9. https://doi.org/10.1109/fie.2018.8659011

Saulnier, B. M., Landry, J. P., Longenecker, H. E., Jr., & Wagner, T. A. (2008). From teaching to learning: Learner-centered teaching and assessment in information systems education. *Journal of Information Systems Education, 19*(2), 169-174. https://jise.org/volume19/n2/JISEv19n2p169.html

Schiller, S. Z. (2009). Practicing learner-centered teaching: Pedagogical design and assessment of a second life project. *Journal of Information Systems Education, 20*(3), 369-381. https://jise.org/Volume20/n3/JISEv20n3p369.html

Sewell, A., & St George, A. (2009). Developing efficacy beliefs in the classroom. *The Journal of Educational Enquiry, 1*(2), 58-71. https://ojs.unisa.edu.au/index.php/EDEQ/article/view/576

Shah, U. S., Jinwala, D. C., & Patel, S. J. (2016). An excursion to software development life cycle models: An old to ever-growing models. *ACM SIGSOFT Software Engineering Notes*, *41*(1), 1-6. https://doi.org/10.1145/2853073.2853080

Shama, P. S., & Shivamanth, A. (2015). A review of agile software development methodologies. *International Journal of Advanced Studies in Computers, Science and Engineering, 4*(11), 1-6.

Shinners-Kennedy, D. (2008). The everydayness of threshold concepts: State as an example from computer science. In R. Land, J. H. F. Meyer, & J. Smith (Eds.), *Threshold concepts within the disciplines* (pp. 119-128). Brill. https://doi.org/10.1163/9789460911477_010

Shinners-Kennedy, D., & Fincher, S. A. (2013). Identifying threshold concepts: From dead end to a new direction. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (pp. 9-18). Association for Computing Machinery. https://doi.org/10.1145/2493394.2493396

Siegfried, R. M., Herbert-Berger, K. G., Leune, K., & Siegfried, J. P. (2021, August). Trends of commonly used programming languages in CS1 and CS2 learning. *Proceedings of the 16th International Conference on Computer Science & Education, Lancaster, United Kingdom,* 407-412. https://doi.org/10.1109/ICCSE51940.2021.9569444

Silva, D. B., de Lima Aguiar, R., Dvconlo, D. S., & Silla, C. N. (2019, October). Recent studies about teaching algorithms (CS1) and data structures (CS2) for computer science students. *Proceedings of the IEEE Frontiers in Education Conference, Covington, KY, USA,* 1-8. https://doi.org/10.1109/fie43999.2019.9028702

Sim, T. Y. (2017, November). Online supported learning and threshold concepts in novice programming. Proceedings of the *IEEE Conference on e-Learning, e-Management and e-Services, Miri, Malaysia,* 85-90. https://doi.org/10.1109/ic3e.2017.8409243

Smetsers-Weeda, R., & Smetsers, S. (2017). Problem solving and algorithmic development with flowcharts. *Proceedings of the 12th Workshop on Primary and Secondary Computing Education* (pp. 25-34). Association for Computing Machinery. https://doi.org/10.1145/3137065.3137080

Soares, A., Martin, N. L., & Fonseca, F. (2015). Teaching introductory programming with game design and problem-based learning. *Issues in Information Systems, 16*(3), 128-137. https://doi.org/10.48009/3_iis_2015_128-137

Sorva, J. (2010). Reflections on threshold concepts in computer programming and beyond. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 21-30). Association for Computing Machinery. https://doi.org/10.1145/1930464.1930467

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education, 13*(4), Article 15. https://doi.org/10.1145/2490822

Stachel, J., Marghitu, D., Brahim, T. B., Sims, R., Reynolds, L., & Czelusniak, V. (2013). Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. *Journal of Integrated Design and Process Science, 17*(1), 37-54. https://doi.org/10.3233/jid-2013-0004

Sternberg, R. J., & Sternberg, K. (2016). *Cognitive psychology* (7th ed.). Wadsworth Publishing.

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive load theory*. Springer. https://doi.org/10.1007/978-1-4419-8126-4

Thevathayan, C., & Hamilton, M. (2015). Supporting diverse novice programming cohorts through flexible and incremental visual constructivist pathways. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education* (pp. 296-301). Association for Computing Machinery. https://doi.org/10.1145/2729094.2742609

Thomas, D., & Hunt, A. (2019). *The pragmatic programmer* (2nd ed.). Addison-Wesley.

Thomas, L., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., & Zander, C. (2014). *A broader threshold: Including skills as well as concepts in computing education.* Paper presented at the Paper presented at Fourth Bienniel Conferencer on Threshold Concepts: From personal practice to communities of practice, United Kingdom of Great Britain and Northern Ireland. https://hdl.handle.net/2160/13528

Thompson, S. (1996). *How to program it.* Computing Laboratory, University of Kent. https://www.cs.kent.ac.uk/people/staff/sjt/Haskell_craft/HowToProgIt.html

Umran Alrubaee, A., Cetinkaya, D., Liebchen, G., & Dogan, H. (2020). A process model for component-based model-driven software development. *Information*, *11*(6), 302. https://doi.org/10.3390/info11060302

Uysal, M. P. (2014). Improving first computer programming experiences: The case of adapting a web-supported and well-structured problem-solving method to a traditional course. *Contemporary Educational Technology*, *5*(3), 198-217. https://doi.org/10.30935/cedtech/6125

Vagianou, E. (2006). Program working storage: A beginner's model. *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling* (pp. 69-76). Association for Computing Machinery. https://doi.org/10.1145/1315803.1315816

Vahid, F., Edgcomb, A., Lysecky, R., & Rajasekhar, Y. (2019, October). New web-based learning content for core programming concepts using Coral. *Proceedings of the IEEE Frontiers in Education Conference, Covington, KY, USA*, 1-5. https://doi.org/10.1109/fie43999.2019.9028529

Veerasamy, A. K., D'Souza, D., Lindén, R., & Laakso, M. J. (2019). Relationship between perceived problem-solving skills and academic performance of novice learners in introductory programming courses. *Journal of Computer Assisted Learning*, *35*(2), 246-255. https://doi.org/10.1111/jcal.12326

Vrachnos, E., & Jimoyiannis, A. (2008, July). DAVE: A dynamic algorithm visualization environment for novice learners. *Proceedings of the Eighth IEEE International Conference on Advanced Learning Technologies, Covington, KY, USA*, 319-323. https://doi.org/10.1109/icalt.2008.148

Wang, Y., & Chiew, V. (2010). On the cognitive process of human problem solving. *Cognitive Systems Research*, *11*(1), 81-92. https://doi.org/10.1016/j.cogsys.2008.08.003

Watson, C., & Li, F. W. B. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 39-44). Association for Computing Machinery. https://doi.org/10.1145/2591708.2591749

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, *25*, 127-147. https://doi.org/10.1007/s10956-015-9581-5

Whalley, J., & Kasto, N. (2014, January). How difficult are novice code writing tasks? A software metrics approach. *Proceedings of the Sixteenth Australasian Computing Education Conference, Auckland, New Zealand*, 105-112. https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV148Whalley.pdf

Whitfield, A. K., Blakeway, S., Herterich, G. E., & Beaumont, C. (2007). Programming, disciplines and methods adopted at Liverpool Hope University. *Innovation in Teaching and Learning in Information and Computer Sciences*, *6*(4), 145-168. https://doi.org/10.11120/ital.2007.06040145

Whittemore, R., & Knafl, K. (2005). The integrative review: Updated methodology. *Journal of Advanced Nursing*, *52*, 546-553. https://doi.org/10.1111/j.1365-2648.2005.03621.x

Williams, J., & Chinn, S. J. (2009). Using Web 2.0 to support the active learning experience. *Journal of Information Systems Education*, *20*(2), 165-174. https://jise.org/volume20/n2/JISEv20n2p165.html

Wing, J. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33-35. https://doi.org/10.1145/1118178.1118215

Wing, J. (2011). Research notebook: Computational thinking – What and why? *The Link*: *The Magazine of the Carnegie Mellon University School of Computer Science*. https://people.cs.vt.edu/~kafura/CS6604/Papers/CT-What-And-Why.pdf

Wirth, N. (2001). Program development by stepwise refinement. In M. Broy & E. Denert (Eds.), *Pioneers and their contributions to software engineering* (pp. 545-569). Springer. https://doi.org/10.1007/978-3-642-48354-7_23

World Economic Forum. (2020). *The future of jobs report 2020*. https://www.weforum.org/reports/the-future-of-jobs-report-2020

Wright, A. L., & Gilmore, A. (2012). Threshold concepts and conceptions: Student learning in introductory management courses. *Journal of Management Education*, *36*(5), 614-635. https://doi.org/10.1177/1052562911429446

Wright, D. R. (2011). *Principles, patterns, and process: A framework for learning to make software design decisions* [Doctoral dissertation, North Carolina State University]. http://www.lib.ncsu.edu/resolver/1840.16/7447

Wright, D. R. (2012, June). Inoculating novice software designers with expert design strategies. *Proceedings of the American Society for Engineering Education Conference* (pp. 25.784.1 - 25.784.25). https://doi.org/10.18260/1-2--21541

Yeomans, L., Zschaler, S., & Coate, K. (2019). Transformative and troublesome? Students' and professional programmers' perspectives on difficult concepts in programming. *ACM Transactions on Computing Education*, *19*(3), 1-27. https://doi.org/10.1145/3283071

Zarb, M., Abandoh-Sam, J. A., Alshaigy, B., Bouvier, D., Glassey, R., Hughes, J., & Riedesel, C. (2018). An international investigation into student concerns regarding transition into higher education. *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 344-345). Association for Computing Machinery. https://doi.org/10.1145/3293881.3295780

Zhang, H., & Liu, H. (2012, August). Educational software process improvement model and strategy. *Proceedings of the International Conference on Computer Science and Information Processing, Xi'an, Shaanxi,* 945-947. https://doi.org/10.1109/csip.2012.6309011

AUTHORS

Dr Catherine Higgins has over 25 years of experience working in third-level education as a lecturer in software development. Currently, she is working in the Faculty of Business at the Technological University Dublin. She has also worked in the computing industry as a developer and in software quality assurance. Her principal research interest is in researching ways of improving competence and success rates in computer science education, particularly for first-year computing undergraduates.

Dr Ciarán O'Leary is Head of Teaching and Learning in the Faculty of Computing, Digital and Data and a lecturer in Computer Science at the Technological University Dublin. As a researcher, Ciarán is primarily interested in how people use digital technology in education and elsewhere, as well as how designers expect people to use digital technology. Other interests include education for sustainable development in the context of future-focused Computing curricula and the embedding of undergraduate research in STEM programmes.

Dr Claire McAvinia is an academic developer at Trinity College Dublin, providing expertise in learning, teaching, and assessment in higher education, including digital education. Her research interests are in open education, education for sustainable development, and post-digital learning spaces in higher education. She previously received a Teaching Hero Award from Ireland's National Forum for the Enhancement of Teaching and Learning and is a Fellow of both the UK's Staff and Educational Development Association (SEDA) and Advance HE.

Dr Barry J. Ryan is a biochemistry lecturer at Technological University Dublin and is currently on secondment, leading the development of the university's Educational Model. He is passionate about the practical implementation of research-informed teaching and supporting others in their personal development in this area. He is concurrently a Senior Fellow of the Higher Education Academy, a National Forum Teaching and Learning Research Fellow, and a chartered science teacher.