Accepted manuscript, to appear in:
Proceedings of the 31st ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2026)

Diagnosing Debugging: The ECDM Framework for Designing Diagnostic Cases in Simulation-Based Teacher Training

Viviane Rehor
Heike Wachter
Technical University of Munich
Computing Education Research Group
Munich, Germany

Annabel Wolf
Christian Hartmann
Maria Bannert
Technical University of Munich
Chair of Teaching and Learning with Digital Media
Munich, Germany

Tilman Michaeli
Technical University of Munich
Computing Education Research Group
Munich, Germany

Abstract

Debugging is a central yet challenging activity for novice programmers, and diagnosing students' debugging difficulties places high demands on teachers under time pressure. Research shows that especially novice teachers often struggle to diagnose such situations in a precise and meaningful way. Although simulation-based approaches have proven effective for fostering diagnostic skills in other domains, they remain underexplored in computer science (CS) teacher education, also lacking theory-driven structures for designing diagnostically rich debugging cases. To this end this paper introduces the ECDM Framework, a conceptual framework for constructing and analyzing realistic diagnostic cases for debugging situations. It integrates four core dimensions, **E**rror, **C**ause, **D**ebugging process, and **M**otivational–affective trajectory, to capture essential aspects of authentic debugging situations while allowing controlled complexity. The paper illustrates how the framework can be applied and discusses its potential to support teachers' diagnostic skills in debugging.

CCS Concepts

• **Social and professional topics → Computing education**.

Keywords

Debugging, teacher education, diagnostic skills, simulation-based learning

ACM Reference Format:
Viviane Rehor, Heike Wachter, Annabel Wolf, Christian Hartmann, Maria Bannert, and Tilman Michaeli. 2026. Diagnosing Debugging: The ECDM Framework for Designing Diagnostic Cases in Simulation-Based Teacher Training. In *Proceedings of the 31st ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2026), July 10–15, 2026, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3803400.3809348

1 Introduction

Debugging is an indispensable aspect of programming, yet it is one of the most difficult activities for novices [35, 48]. Beginners often lack the experience to manage the complexity of identifying and fixing errors and rely on support by teachers [29]. In such situations, teachers have to rapidly *diagnose* why a student is unable to resolve a specific bug. This is a substantial challenge for teachers [46], in particular for novice teachers with less classroom experience.

Diagnosing is a core professional practice of teachers. Therefore, there is a broad consensus across disciplines that diagnostic skills should be explicitly fostered in teacher training [6]. This requires structured opportunities to practice. Simulation-based approaches have proven effective for this purpose, for example in domains such as mathematics [7, 39] or physics [23]. To this end, simulations represent professional practice, such as classroom situations or student–teacher interactions, for which students can practice diagnosing. Central to any such simulation are the underlying cases. In this context, a *case* is defined as a specific instantiation of a diagnostic situation. It is constructed by configuring distinct student and problem characteristics to create a coherent diagnostic task, represented either through recordings of real lessons or as scripted, purposefully designed scenarios. While unscripted cases offer high authenticity, scripted cases enable diagnostically relevant aspects to be selectively highlighted, reduced, or systematically manipulated. This allows repeated practice with controlled complexity and adjustable pacing, supporting focused diagnostic learning without the time pressure, high emotional stakes, cognitive demand and unpredictability of live classroom settings [9, 32]. Despite this potential, simulation-based approaches remain largely unexplored in CS education with only a few first studies in this direction [36, 42, 47, 51].

To adequately diagnose debugging situations, teachers must assess observable aspects and draw informed inferences from them [15]. Developing cases therefore requires a systematic examination of the essential aspects necessary for diagnosing. Existing studies provide insights into *what* errors students make (typical programming problems [1, 2, 17, 27]), *why* these errors arise during programming (causesdifficulties [28, 37, 41]), and *why* students struggle to resolve them independently (debugging processes and behaviors [21, 33] as well as motivational-affective factors). However, these strands of work have rarely been synthesized with the explicit goal of identifying diagnostically relevant, observable aspects that can inform the systematic design of debugging cases.

In this article, we address this gap by introducing the ECDM Framework, which structures empirically grounded aspects of debugging situations into a coherent model to support the creation of authentic diagnostic cases for teacher training, applicable for both K-12 and higher education teacher training.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ITiCSE 2026, Madrid, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2634-7/2026/07
https://doi.org/10.1145/3803400.3809348

2 Theoretical Background and Related Work

2.1 Diagnostic Skills

Diagnostic skills are widely recognized as a core component of effective teaching and teachers professional competence. *Diagnosing* refers to the identification and interpretation of students' abilities, misconceptions, learning needs, and motivational states in order to inform appropriate instructional decisions and interventions [45]. Diagnostic situations in classroom practice are characterized by high informational complexity, varying degrees of typicality, and substantial demands on professional agency, including time pressure, social interaction, and self-regulation [9]. Diagnosis is further complicated by the fact that many relevant aspects are not directly observable but must be inferred from *observable cues* through the application of teachers' professional knowledge. These demands are particularly challenging for novice teachers [6]. Effective diagnosis requires an interplay of content knowledge (CK), pedagogical knowledge (PK), and pedagogical content knowledge (PCK), which together enable teachers to recognize diagnostically relevant cues and select appropriate instructional responses [22].

Simulation-based approaches to foster diagnostic skills. Subject specific studies (e.g. [7, 23, 36, 39, 51]) as well as a comprehensive meta-analysis [6] show that scaffolded simulation-based learning can lead to substantial gains in diagnostic skills across teachers with varying prior knowledge. Central to any simulation is an underlying case and a representation in a chosen modality (e.g., text-based, video) for the particular simulation environment: a concrete, exemplary professional situation that captures essential aspects of practice. Take a scripted scene of two students struggling with a specific bug and requiring teacher support. Through *representational scaffolding,* simulations systematically reduce and control complexity by selecting, emphasizing, or omitting situational aspects, thereby lowering cognitive load and directing attention to diagnostically relevant observable cues. With increasing experience, additional aspects can be reintroduced to progressively approximate the full complexity of authentic classroom practice [9, 13, 43]. Effective simulation-based diagnostic training therefore crucially depends on the availability of well-designed cases that systematically represent diagnostically relevant aspects of practice.

2.2 Debugging and Diagnosing Debugging Situations

Debugging is not a linear task but a demanding, iterative, and hypothesis-driven process. It requires coordinating multiple skills, including understanding the problem domain, knowing programming concepts, tracing program logic, and locating and fixing errors [21, 33, 40]. Beginners often lack the experience needed to manage this complexity [35]. As a consequence, students frequently rely on teachers when they encounter debugging problems in classroom settings. Supporting students during debugging therefore constitutes a major professional challenge for CS teachers [29].

Classroom-based studies on student–teacher interactions, particularly in debugging physical computing systems, document a range of instructional strategies, such as asking guiding questions, articulating the problem, emphasizing systematic processes, or responding to students' frustration and joy [16]. While these studies provide rich **descriptions of teachers' observable actions**, they place less emphasis on the diagnostic reasoning processes through which teachers interpret students' difficulties and select these actions.

A small number of studies have **examined teachers' diagnostic and pedagogical reasoning** using elicited tasks and representations outside real classroom settings. For example, Tsan et al. asked how teachers would help their students in debugging in a professional development context and found that teachers often focused on providing solutions rather than engaging with students' underlying reasoning [42]. Yadav et al. employed scripted video-based simulations of students' understanding of programming constructs to study how CS teachers respond. Wachter and Michaeli used scripted videos of debugging situations to analyze how teachers would diagnose and intervene [46, 51]. While these approaches demonstrate the potential of simulations for **examining** diagnostic reasoning, they do not offer much transparency regarding how their cases are constructed or which diagnostically relevant aspects are selected and structured.

Finally, approaches explicitly aimed at **fostering diagnostic skills** in CS remain scarce. One emerging example is BlockTalk, a generative AI-based tool that enables teachers to practice conversational debugging in simulated interactions [14]. Similarly, Wachter and Michaeli report an exploratory pilot study using scripted video simulations to support novice teachers' diagnostic skills, while emphasizing the need for broader and more systematic investigations [47].

Overall, the literature underscores the promise of simulations but offers limited guidance on the systematic design of cases for simulation-based training. This highlights the need for systematic frameworks that integrate empirical knowledge about learners, debugging processes, and diagnostic demands.

3 Analyzing Debugging Situations from a Diagnostic Perspective

Constructing authentic and instructionally valuable cases requires identifying the core aspects that define a classroom debugging situation. Although debugging is a multi-layered process, not all of its aspects are equally observable or diagnostically relevant for teachers. In this section, we therefore analyze prior research on novice programming and debugging from a diagnostic perspective to identify the observable aspects that inform instructional decisions. Based on this synthesis, we identified four interconnected dimensions that are essential for diagnosis: What kind of error was made, why was it made, why a student is not able to solve it and how motivation influences the whole process. The following subsections ground each dimension in existing research and establish its relevance for teachers' diagnostic reasoning.

3.1 What error was made?

From a design perspective, the error (or bug; used interchangeably [33]) type constitutes the central anchor of any debugging scenario. Error types are diagnostically relevant because they shape how students search for, interpret, and attempt to fix problems, and thus influence which instructional interventions are appropriate [5, 29]. Research shows that the error type determines whether

the problem is signaled explicitly (e.g., via compiler messages or errors during execution in syntax and semantic errors) or remains implicit (e.g., via incorrect output in logic errors) which influences the observability of the error [8]. Across studies, **syntax errors** are typically the most frequent and easiest to fix, whereas for **logic errors** the source of the error is harder to identify [1, 2, 17, 27, 30]. In contrast, research in K–12 classrooms shows that even compile-time errors pose major challenges for novices [29]. Recent work therefore argues for moving beyond error frequency toward errors that genuinely cause student struggle, as such errors are more likely to lead to frustration without learning gains [3]. Therefore, to create a case the error type has to be explicitly defined, as this fundamentally constrains the student's potential actions and the diagnostic cues available to the teacher.

3.2 Why was the error made?

While the error defines the observable problem, the cause constitutes the hidden reasons for the creation of the bug. To create a coherent case, the design must determine why the error occurred, as identical errors can stem from fundamentally different student difficulties. Only by identifying these underlying reasons can teachers support students in overcoming and understanding the error. Early influential work by Spohrer and Soloway differentiated whether errors stemmed from misconceptions about programming language constructs or from plan composition problems, noting that the former were less widespread than commonly assumed [41]. [34] further demonstrated that many recurring errors originate from language-independent conceptual bugs summarized as the belief that the computer is "intelligent" and "just understands" what the programmer intends (even if it is not explicitly written in code) [34]. Later work synthesized these and other findings into a set of diagnostically relevant *student difficulties* as error causes: **syntactic**, **conceptual**, and **strategic difficulties** [37]. Most advanced being strategic difficulties, which are responsible for missing programming strategy during problem decomposition, design, testing, and debugging, especially for novel and complex problems [28]. Importantly, Veerasamy explicitly distinguishing **slips and lapses** from knowledge- and rule-based errors, highlighting that not all debugging problems indicate missing understanding [44]. This distinction is vital for scripting the student's reaction to teacher interventions: while **syntactic**, **conceptual** or **strategic difficulties** require explanatory or scaffolding interventions, in the case of a **slip** the student might self-correct upon a simple prompt.

3.3 Why can't the error be solved?

While error and cause define the static state of the problem, the debugging behavior dictates the dynamic evolution of the case. It determines how the simulated student interacts with the error over time, heavily influencing the whole script for the case. For example whether students apply strategies such as tracing, pattern matching (which relies heavily on experience) or targeted testing, whereas struggling students rely on unsystematic tinkering or superficial code reading [10, 11, 31, 49].

Additionally, many students do not fail to resolve bugs due to missing knowledge or motivation, but because they get stuck in the debugging process itself. Research consistently describes an ideal-typical debugging process as an iterative, hypothesis-driven process that can be summarized in four core **Debugging Process Steps (DPS)** [21, 33, 40]:

- DPS 1 Observation of the Failure–noticing unexpected program behavior or output,
- DPS 2 Formulation of Hypotheses–generating explanations for the observed failure,
- DPS 3 Identification and Verification–testing hypotheses to locate the faulty code (if verification fails **DPS 2**) ,
- DPS 4 Fixing and Verification of the Fix–applying and validating a correction (if verification fails **DPS 2**) .

Across educational contexts, novices frequently omit or incompletely execute key steps of this cycle, particularly hypothesis formation **(2)**, identifying or localizing the bug which includes hypothesis testing **(3)**, and verification of the fix **(4)**, which often results in unproductive trial-and-error behavior [33, 40].

Case design for diagnosing debugging situations therefore requires a scripted debugging behavior for the teacher to recognize both the quality of the debugging process and which strategies students use, misuse, or neglect.

3.4 How does motivation shape debugging?

Finally, to create an authentic representation of a classroom, a case must account for the motivational–affective factors influencing the situation being integrated in the script for the case. Those factors such as self-efficacy, goal orientation, and self-regulation play a central role in how students engage with programming and debugging [24]. Self-efficacy, in particular, is a strong predictor of academic outcomes and has been shown to relate to programming performance and debugging success [50]. Importantly, self-efficacy is dynamic and is shaped by students' emotional experiences during debugging, especially repeated failure or success [20, 26]. Debugging is not a purely cognitive task, it is an affectively demanding activity often accompanied by frustration, anxiety, or fluctuating confidence [12, 19].

Although these factors have a significant impact on students' engagement and perseverance, motivational–affective aspects of debugging remain underrepresented in computing education research [24]. From a diagnostic perspective, this implies that teachers must attend not only to cognitive or conceptual difficulties, but also to students' motivational states and confidence. Diagnosing these aspects is essential for selecting interventions that reduce frustration, support persistence, and foster positive feedback loops in which successful debugging strengthens self-efficacy and future engagement [26, 29, 40].

4 The ECDM Framework

Building on the diagnostic aspects identified in section 3, we now propose the ECDM Framework as a tool for case design consisting of four dimensions: **E**rror, **C**ause, **D**ebugging process, and **M**otivational–affective trajectory. While the previous section analyzed main aspects of debugging situation based on empirical research, this section operationalizes these findings into concrete categories for case construction. The framework functions as a

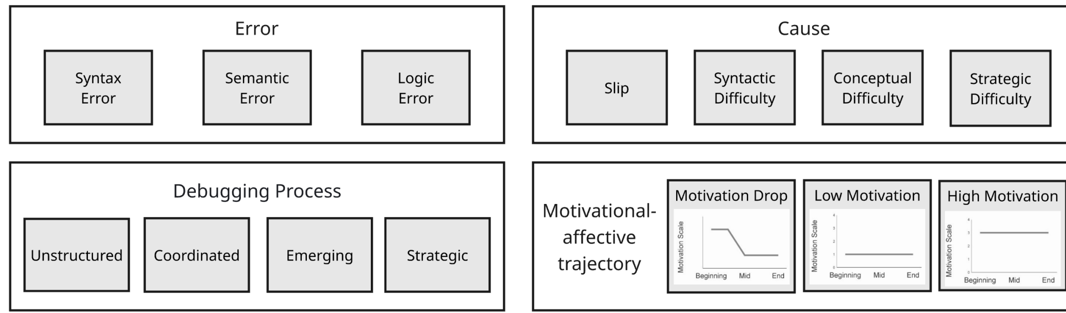

Figure 1: The ECDM framework.

modular system (see figure 1). The structure allows case designers to systematically "configure" a realistic scenario by selecting one category from each dimension, choose an instantiation of that category and realize a case representation through a chosen modality (e.g., text-based, video) corresponding to the particular simulation environment which includes observable cues. These cues include source code, student reasoning dialog, behavior showing motivational-affective states and debugging behavior. By repeating this process with different configurations and different instantiations, the ECDM framework supports the creation of a wide range of authentic cases teachers may encounter in practice. In the following subsections, we describe these specific categories and give examples for their instantiation.

4.1 Error

The core of any case is the specific error. This dimension describes what kind of error is in the code and is an objective dimension that can not be observed from the students behavior. Since the range of possible errors is vast, the framework uses the error types with similar observability based on the nature of system feedback they trigger as categories: **Syntax**, **semantic**, and **logic errors**, widely used in introductory programming education and empirical research [1, 2, 17, 27, 38]. **Syntax errors** result from violations of a programming language's formal grammar (e.g., missing semicolons, mismatched braces) and prevent compilation, often allowing bug localization at line level. **Semantic errors** occur in syntactically correct code when language constructs are misused or misunderstood, manifesting as compile-time errors or runtime failures (e.g., undeclared identifiers, incorrect operators, wrong parameter usage). **Logic errors** compile and execute but produce incorrect behavior (e.g., faulty loop conditions), making them particularly difficult for novices to detect and reason about.

4.2 Cause

In a specific case we need to show why a student produced a particular error, we adopt the well-established distinction proposed by Qian and Lehman and extend it by explicitly including slips as a separate category [37, 44]. Together, **slips**, **syntactic difficulties**, **conceptual difficulties**, and **strategic difficulties** provide a comprehensive yet non-overly granular account of novice programming difficulties, grounded in computing education and cognitive psychology and supported by empirical work [37, 44]. **Slips** are unintended performance errors caused by momentary lapses in attention or memory and occur despite the student having the necessary knowledge. **Syntactic difficulties** arise when students know what they want to express but lack the formal language knowledge to produce syntactically correct code. **Conceptual difficulties** arise from misunderstandings of programming constructs, principles, or the semantics of operations and involve incomplete or incorrect mental models of programming constructs. **Strategic difficulties** refer to problems in planning, task understanding, and integrating syntactic and conceptual knowledge into coherent solution strategies during program construction. Importantly, our four dimensions reflect a temporal distinction: error causes explain why the erroneous action occurred during programming, whereas difficulties when trying to find and fix the bug are treated separately. Although prior work has sometimes subsumed debugging under strategic difficulties [28, 37], we exclude debugging-related difficulties here due to the distinct competencies required for debugging compared to programming [1, 10] and address them in the debugging-skill dimension in section 4.3.

Importantly, the **cause** dimension is closely related to the **error** dimension but not a direct 1:1 mapping. While many combinations of error and cause are theoretically possible, some are more likely than others. For example, an instantiation of a logic error would be a method call where the return value is not handled. In that case the final calculation would not show the expected result. The error may arise from a conceptual difficulty, such as not understanding that a method's return value must be explicitly handled to have an influence on the result, but it may also stem from a slip, where the student simply forgets to assign the return value to a variable despite understanding the concept. Some other combinations might be considerably less plausible. Keeping error and cause conceptually separate allows the framework to represent diagnostically meaningful variation: the same observable error may reflect different underlying causes and therefore require different instructional responses. By selecting empirically established causes that recur across learners, the framework enables the systematic generation and analysis of typical debugging situations that reflect instantiations of different categories of cognitive origins of errors.

4.3 Debugging Process

The next aspect that needs to be integrated in a case is the student's debugging behavior. This dimension describes how the student

approaches the problem and is chosen independently from the other categories: A student with any level of debugging skill can encounter any type of error or underlying cause. To categorize this behavior for the case design, we distinguish the **quality of the debugging process**. To this end, we describe qualitative ranges of the debugging process, spanning from **Unstructured** to **Strategic**. We deliberately use descriptive names rather than numerical levels to emphasize that these are *interpretive groupings*, not fixed stages of development. The distinction between these quality levels is necessarily fluid, and no clear-cut boundaries can be assumed in authentic classroom situations. The choice to describe four levels is therefore pragmatic, serving primarily to support varied case generation for simulation-based training. Each quality level is instantiated by choosing from optional indicators derived from literature (e.g. [4, 10, 11, 18, 31, 48, 49]) that serve as guidance.

An **unstructured** debugging process can be seen through omitting of key steps of the ideal-typical debugging process: students often fail to generate hypotheses (DPS 2), struggle to identify the relevant code location (DPS 3), and rely on unsystematic trial-and-error or tinkering without verifying whether the issue has been resolved (DPS 4). An **emerging** debugging process is characterized by students understanding the task and noticing that something is wrong, but still fail to articulate or test a concrete hypothesis (DPS 2). Attempting to isolate the problem remains unstructured (e.g., commenting out random code), and testing is typically limited to single examples without considering edge cases (DPS 4). A **coordinated** debugging process involves successful hypothesis formation (DPS 2) and the use of more systematic strategies, such as consulting error messages, IDE features, or basic tracing, yet students may still struggle to precisely locate the faulty code segment or to consistently verify hypotheses (DPS 3), often tracing linearly from the start of the program. Lastly, a **strategic** debugging process, is characterized by a fully coordinated and reflective process: students form and revise hypotheses (DPS 2), systematically locate the source of the bug (DPS 3), avoid trial-and-error fixes, trace execution flow, often backwards from the observed failure, and effectively use IDE tools and error messages, followed by careful verification through testing with multiple and varied test cases to ensure correctness (DPS 4).

4.4 Motivational-affective trajectory

The final element of a case is to determine the student's motivational-affective trajectory during the debugging session. As discussed in section 3.4, motivational-affective components strongly influence students' debugging performance, their willingness to persist, and their broader interest in programming while still being an independent dimension. This means that any selection of a motivational-affective trajectory can be made for all possible combinations of the other three dimensions. Research shows that emotional states can fluctuate significantly during debugging: Frustration caused by an impasse may be followed by joy and increased confidence after successfully resolving an error. Confident and motivated students might even keep their positive mindset through the whole process, in disregard of not solving the problem (**High Motivation**). On the other hand, initial motivation can be followed by increasing frustration when aimlessly tinkering and not finding a solution (**Motivational Drop**) [19]. Sometimes a state of anxiety due to low self-efficacy from the beginning and the fear of making mistakes can even persist through the whole process (**Low Motivation**) [25].

Although motivation has been noted for its conceptual diffuseness and is hard to define, its resulting engagement can be measured via persistence, effort, help-seeking and continued participation [24]. Such indicators constitute the building blocks used to construct the three motivational-affective trajectories. Integrating these trajectories is crucial for creating training cases since teachers need to recognize affective patterns that may hinder or facilitate debugging, enabling more targeted and supportive interventions.

5 Application of the ECDM Framework

To illustrate how the ECDM framework can be applied in practice, we describe the workflow of creating a concrete case for a K-12 setting: We select a concrete instantiation for each of the four dimensions—error, cause, debugging process, and motivational-affective trajectory—using the framework's categories as a structural guideline. While the framework allows starting with any dimension, inherent interdependencies exist. For instance, the particular error type limits the range of plausible causes and debugging processes that can be realistically depicted. This involves defining a concrete coding example, an explanation for the origin of the error, indicators of a particular debugging process quality, and a motivational–affective trajectory. In the following, we apply this workflow to construct a concrete case for a K-12 setting and demonstrate a possible representation for a video-based simulation environment.

Example Case. We start our case by choosing the error, a frequently reported **semantic** novice error in Java programming: Invoking a method that is not defined in the current scope [27] – providing a realistic and empirically grounded starting point. Given this error instantiation, certain causes become more plausible than others. While a **slip** (e.g. forgetting to define the method) could in principle lead to a similar bug, we deliberately select a **conceptual difficulty** as the underlying cause. In this case, the students assume that a method such as `min()` should exist because it is meaningful in everyday language. This reflects an incomplete understanding of built-in language features versus user-defined methods. We decide for the students to exhibit a **coordinated debugging process** which will be manifested through a nearly ideal-typical process, making use of IDE features (a strategy seen in experts [11]), and reading the error message (expert like behavior [48]). Additionally, we will have the students show a **high motivation** throughout the case. We acknowledge that inferring a student's internal state is a complex task, thus observable motivational cues should initially be represented through clear indicators and can later be extended to more subtle cues as diagnostic expertise increases. This specific combination creates a diagnostically challenging situation: The teacher must recognize that despite the students' systematic and motivated approach to find and fix the error, they cannot succeed due to a specific underlying conceptual difficulty.

From Case to Simulation. Once a case has been instantiated, it can be represented in various formats, such as text-based scenarios, conversational agents, or videos – depending on the intended

simulation environment. To this end, the selected case characteristics must be made observable through carefully designed cues that allow teachers to infer the diagnostically relevant aspects. For our example, we will sketch the script for a video-based simulation depicting students engaged in a debugging task, accompanied by a screen capture of their code. This dual representation enables the simultaneous observation of students' verbal interactions, debugging behaviors, and program execution, thereby supporting authentic diagnostic reasoning.

We showcase two motivated students, using a conversational cue like: *This seems doable, I think we can solve it* in a enthusiastic fashion *[motivational cue]*, in an introductory CS classroom learning about Java and confronted with the task: *Compute the minimum of value* $a = 5$ *and value* $b = 10$ *using a method in the given class* `MinCalculator` *and print the result.* In their first attempt, the students write the following program *[error cue]*, convinced that `min` is a built-in Java method.

```
public class MinCalculator {
    public void MinCalculator() {
        int a = 5;
        int b = 10;
        int min = 0;
        min = min(a, b);
        System.out.println("The minimum is: " + min);
    }
}
```

The compiler recognizes the syntactic structure of the program, but the symbols are not meaningful in context because the method `min` is not declared (`Undeclared method: min(int,int)`). When the IDE highlights `min` with a red underline, the students first observe the failure *[debugging process cue (dpc): DPS 1]*. They then formulate an initial hypothesis *[dpc: DPS 2]*, suspecting a spelling issue and reasoning that exact lettering is crucial in programming *[cause cue: The students express the belief that the method must exist]*. Supported by the IDE, and the students attempt to fix and verify the hypothesis *[dpc: DPS 3/4]* by changing `min` to `Min` and recompiling the program. However, this does not resolve the error. Using the coding environment, the students undo the unsuccessful change via the IDE, demonstrating flexible tool use and strategic behavior characteristic of more advanced debugging *[dpc]*. Remaining motivated and verbalizing it with statements like: *I think we're on the right track [motivational cue]*, they consider alternatives *[dpc]* and formulate a new hypothesis *[dpc: DPS 2]*: *Perhaps* `min` *has to be defined earlier. Let's add the variable!* They modify the code accordingly by declaring `int min = 0;` and reattempt compilation. Once again, fixing and verification *[dpc: DPS 3/ 4]* are applied, yet the error persists. At this point, the students ask the teacher for help.

Crucially, the underlying case remains modality-independent, supporting various representations. Translating a case into a functional simulation, however, remains a complex, modality-dependent challenge. Rather than authoring the scenario itself, the ECDM framework structures the case generation to ensure every representation is grounded in established debugging research.

6 Discussion and Conclusion

Diagnosing debugging is more than just identifying a bug. It is a complex professional skill that requires dedicated practice, for which simulations offer great potential in teacher training. With the ECDM framework, we propose a conceptual basis for constructing diagnostic cases for debugging situations, addressing the current lack of theory-driven foundations for simulation case design in this domain.

By bringing together error, cause, debugging process, and motivational-affective trajectory, the framework structures the design of individual cases and ensures that diagnostically relevant aspects are represented. Motivational-affective aspects, in particular, are reported to be neglected by teachers in their diagnosis [46], emphasizing their importance in learning to diagnose.

Beyond individual cases, the ECDM framework enables the systematic generation of comprehensive debugging sets. While the potential for combinatorial complexity is high, the framework's purpose is not to cover every possible permutation, but to ensure that selected cases are theoretically grounded and comparable. By highlighting key dimensions, the framework allows for the strategic, rather than exhaustive, variation of factors to create distinct training scenarios across a wide range of authentic debugging situations.

While the ECDM framework is based on empirical research on debugging situations, the framework itself and its usefulness for case generation have not yet been empirically validated. To address this, in future work, we will investigate the authenticity, typicality, complexity, and comparability of cases created with the framework, as well as their suitability for fostering diagnostic skills. Authenticity, in particular, appears critical, as prior research indicates the importance of realistic and transferable cases in simulation-based learning of diagnostic skills [7].

Beyond case design, the ECDM framework also opens directions for future research on teachers' diagnostic skills. We argue that it provides a structured basis for investigating how and when different diagnostically relevant aspects–derived from the framework–are attended to and utilized by teachers during diagnostic activities. For example, relationships between error types or underlying causes and diagnostic outcomes can be examined in relation to teachers' prior CK and PCK. Additionally, though our current focus is not on providing instructional interventions to teachers, the framework could serve as a foundation to asses how different cases require different teacher interventions. In this way, the ECDM framework not only supports simulation-based training but also offers a foundation for advancing empirical research on teachers' diagnostic skills in CS education.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – TRR 419/1 – 542251251. The authors used ChatGPT for language revision and take full responsibility for the final content.

References

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *SIGCSE Bull.* 37, 3 (2005), 84–88.

[2] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of SIGCSE'15*. ACM, 522–527.

[3] Nabeel Alzahrani and Frank Vahid. 2021. Common Logic Errors for Programming Learners: A Three-decade Literature Survey. In *Proceedings of ASEE Virtual'21*. ASEE Conferences.

[4] Laura Bofferding, Sezai Kocabas, Mahtob Aqazade, Ana-Maria Haiduc, and Lizhen Chen. 2022. *The effect of play and worked examples on first and third graders' creating and debugging of programming algorithms*. ACM, 19–29.

[5] Sharon McCoy Carver and David Klahr. 1986. Assessing Children's Logo Debugging Skills with a Formal Model. *Journal of Educational Computing Research* 2, 4 (1986), 487–525.

[6] Olga Chernikova, Nicole Heitzmann, Maximilian Christian Fink, Venance Timothy, Tina Seidel, Frank Fischer, and DFG Research group COSIMA. 2020. Facilitating diagnostic competences in higher education—a meta-analysis in medical and teacher education. *Educational Psychology Review* 32, 1 (2020), 157–196.

[7] Elias Codreanu, Daniel Sommerhoff, Sina Huber, Stefan Ufer, and Tina Seidel. 2020. Between Authenticity and Cognitive Demand: Finding a Balance in Designing a Video-Based Simulation in the Context of Mathematics Teacher Education. *Teaching and Teacher Education* 95 (2020), 103146.

[8] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proceedings of ACE'18*. ACM, 83–89.

[9] Frank Fischer et al. 2022. Representational Scaffolding in Digital Simulations – Learning Professional Practices in Higher Education. *Information and Learning Sciences* 123, 11/12 (2022), 645–665.

[10] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[11] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging From the Student Perspective. *IEEE Transactions on Education* (2010), 390–396.

[12] Jamie Gorson, Kathryn Cunningham, Marcelo Worsley, and Eleanor O'Rourke. 2022. Using Electrodermal Activity Measurements to Understand Student Emotions While Programming. In *Proceedings of ICER'22*. ACM, 105–119.

[13] Pamela Grossman, Christa Compton, Danielle Igra, Matthew Ronfeldt, Emily Shahan, and Peter W. Williamson. 2009. Teaching Practice: A Cross-Professional Perspective. *Teachers College Record* 111, 9 (2009), 2055–2100.

[14] Paulina Haduong, Brian Yu, Miranda Shen, Bobby Gerami, and Karen Brennan. 2025. BlockTalk: GenAI Simulations for Conversational Debugging Practice. In *Proceedings of ICLS 2025*. ISLS, 2852–2854.

[15] Nicole Heitzmann et al. 2019. Facilitating Diagnostic Competences in Simulations in Higher Education A Framework and a Research Agenda. *Frontline Learning Research* (2019), 1–24.

[16] Colin Hennessy Elliott, Alexandra Gendreau Chakarov, Jeffrey B. Bush, Jessie Nixon, and Mimi Recker. 2023. Toward a Debugging Pedagogy: Helping Students Learn to Get Unstuck with Physical Computing Systems. *Information and Learning Sciences* 124, 1/2 (2023), 1–24.

[17] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull.* 35, 1 (2003), 153–156.

[18] Irvin R. Katz and John R. Anderson. 1987. Debugging: An Analysis of Bug-location Strategies. *Human–Computer Interaction* 3, 4 (1987), 351–399.

[19] Päivi Kinnunen and Beth Simon. 2010. Experiencing Programming Assignments in CS1: The Emotional Toll. In *Proceedings of ICER'10*. ACM, 77–85.

[20] Päivi Kinnunen and Beth Simon. 2011. CS Majors' Self-Efficacy Perceptions in CS1: Results in Light of Social Cognitive Theory. In *Proceedings of ICER'11*. ACM, 19–26.

[21] David Klahr and Sharon McCoy Carver. 1988. Cognitive objectives in a LOGO debugging curriculum. *Cognitive Psychology* (1988), 362–404.

[22] Maria Kramer, Christian Förtsch, William J. Boone, Tina Seidel, and Birgit J. Neuhaus. 2021. Investigating Pre-Service Biology Teachers' Diagnostic Competences: Relationships between Professional Knowledge, Diagnostic Activities, and Diagnostic Accuracy. *Education Sciences* 11, 3 (2021), 89.

[23] Ingrid Krumphals and Markus Feser. 2025. Fostering pre-service physics teachers' diagnostic skills and readiness through video vignettes and micro-teaching sessions: An exploratory single- case study. *Journal of Physics Conference Series* 2950 (2025), 012042.

[24] Alex Lishinski and Aman Yadav. 2019. Motivation, Attitudes, and Dispositions. In *The Cambridge Handbook of Computing Education Research* (1 ed.). Cambridge University Press, 801–826.

[25] Alex Lishinski, Aman Yadav, and Richard Enbody. 2017. Students' Emotional Reactions to Programming Projects in Introduction to Programming: Measurement Approach and Influence on Learning Outcomes. In *Proceedings ICER'17*. ACM, 30–38.

[26] Adam V. Maltese, Alex Simpson, and Amy Anderson. 2018. Failing to Learn: The Impact of Failures during Making Activities. *Thinking Skills and Creativity* 30 (2018), 116–124.

[27] Davin McCall and Michael Kolling. 2014. Meaningful Categorisation of Novice Programmer Errors. In *Proceedings of FIE'14*. IEEE, 1–8.

[28] Tanya J. McGill and Simone E. Volet. 1997. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297.

[29] Tilman Michaeli and Ralf Romeike. 2019. Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. In *Proceedings of EDUCON'19)*. 1030–1038.

[30] Chan Mow. 2012. Analyses of student programming errors in Java programming courses. *Journal of Emerging Trends in Computing and Information Sciences* 3, 5 (2012), 739–749.

[31] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proceedings of SIGCSE'08*. ACM, 163–167.

[32] David Nicol. 2021. The power of internal feedback: exploiting natural comparison processes. *Assessment & Evaluation in Higher Education* 46, 5 (2021), 756–778.

[33] Meghan M. Parkinson, Seppe Hermans, David Gijbels, and Daniel L. Dinsmore. 2024. Exploring Debugging Processes and Regulation Strategies during Collaborative Coding Tasks among Elementary and Secondary Students. *Computer Science Education* 34, 4 (2024), 617–644.

[34] Roy D. Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (1986), 25–36.

[35] D. N. Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., 213–229.

[36] Ursula Pieper and Jan Vahrenhold. 2020. Critical Incidents in K-12 Computer Science Classrooms - Towards Vignettes for Computer Science Teacher Training. In *Proceedings of SIGCSE'20*. ACM, 978–984.

[37] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1 (2017), 1–24.

[38] Davorka Radaković, Faculty of Sciences University of Novi Sad (Serbia), and Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. 2024. Common Errors in High School Novice Programming. *IPSI Transactions on Internet Research* 20, 1 (2024), 47–59.

[39] Daniel Sommerhoff, Elias Codreanu, Michael Nickl, Stefan Ufer, and Tina Seidel. 2023. Pre-Service Teachers' Learning of Diagnostic Skills in a Video-Based Simulation: Effects of Conceptual vs. Interconnecting Prompts on Judgment Accuracy and the Diagnostic Process. *Learning and Instruction* 83 (2023), 101689.

[40] Elena Spoerer and Tilman Michaeli. 2025. Investigating Debugging Processes: A Scoping Review. In *Proceedings of Koli Calling '25*. ACM.

[41] James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[42] Jennifer Tsan, David Weintrop, and Diana Franklin. 2022. An Analysis of Middle Grade Teachers' Debugging Pedagogical Content Knowledge. In *Proceedings of ITiCSE'22*. ACM, 533–539.

[43] Jeroen J. G. Van Merrienboer, Richard Clark, and Marcel Croock. 2002. Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development* 50 (2002), 39–61.

[44] Ashok Kumar Veerasamy. 2016. Identifying Novice Student Programming Misconceptions and Errors From Summative Assessments. *Journal of Educational Technology Systems* 45 (2016), 50–73.

[45] Franziska Vogt and Marion Rogalla. 2009. Developing Adaptive Teaching Competency through coaching. *Teaching and Teacher Education* 25 (2009), 1051–1060.

[46] Heike Wachter and Tilman Michaeli. 2024. Analyzing Teachers' Diagnostic and Intervention Processes in Debugging Using Video Vignettes. In *Proceedings of ISSEP'24*. Vol. 15228. Springer Nature Switzerland, 167–179.

[47] Heike Wachter and Tilman Michaeli. 2025. Fostering Diagnostic Skills: Using Video Vignettes in Computer Science Teacher Education. *HDI* (2025).

[48] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice Reflections on Debugging. In *Proceedings of SIGCSE'21*. ACM, 73–79.

[49] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A Think-Aloud Study of Novice Debugging. *ACM Trans. Comput. Educ.* 23, 2, Article 28 (2023), 38 pages.

[50] Susan Wiedenbeck. 2005. Factors Affecting the Success of Non-Majors in Learning to Program. In *Proceedings of ICER'05*. ACM, 13–24.

[51] Aman Yadav, Marc Berges, Phil Sands, and Jon Good. 2016. Measuring Computer Science Pedagogical Content Knowledge: An Exploratory Analysis of Teaching Vignettes to Measure Teacher Knowledge. In *Proceedings of WiPSCE'16*. ACM, 92–95.