Exploring the Relationship between Debugging Self-Efficacy and CASE Tools for Novice Troubleshooting

Cole Gilbert
Games Academy
Falmouth University
Penryn, Cornwall, UK
cg233192@falmouth.ac.uk

Brian McDonald
Games Academy
Falmouth University
Penryn, Cornwall, UK
brian.mcdonald@falmouth.ac.uk

Michael James Scott
Screen, Technology & Performance
Falmouth University
Penryn, Cornwall, UK
michael.scott@falmouth.ac.uk

ABSTRACT

Novice software developers encounter pitfalls which evoke learning experiences that are more often frustrating than enlightening. Such experiences dampen their debugging self-efficacy, impacting their attainment and retention. Structured debugging using computer-aided software engineering (CASE) tools could help students overcome these obstacles. Unfortunately, students find such tools challenging to use because they tend to cater to the needs of experts rather than novices. This paper examines the relationship between debugging self-efficacy and CASE tools. The study challenged 66 undergraduate computing students to complete a small-scale troubleshooting task in C#, allocating them to one of three groups: those using a simplified tool for novices, others using an off-the-shelf tool, and those using no tool. Analysis shows significant differences between the groups ($p = .02$, $\eta_p^2 = .32$). Using an off-the-shelf tool or no tool decreases debugging self-efficacy. There was no change in debugging self-efficacy when using the simplified tool. These findings suggest that educators should exercise caution when using off-the-shelf tools due to their impact on students' debugging self-efficacy. Simplification appears to mitigate the negative effect but does not seem to offer any improvement.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Applied computing** → *Psychology*; • **Software and its engineering** → *Software maintenance tools*; • **Human-centered computing** → Empirical studies in HCI.

KEYWORDS

computer-aided software engineering, novice programmers, self-efficacy, explicit guidance, debugging, experiment

ACM Reference Format:
Cole Gilbert, Brian McDonald, and Michael James Scott. 2024. Exploring the Relationship between Debugging Self-Efficacy and CASE Tools for Novice Troubleshooting. In *Proceedings of Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research (UKICER'24)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

UKICER'24, September 5–6, 2024, Manchester, UK

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Debugging is a key hurdle to overcome when learning computer programming. Not just because of the defeating feeling of knowing that code doesn't work but also because of the need to interpret cryptic errors and intimidating warnings to fix them. These stymie troubleshooting at a time when mental models of how software works are fragile [33], being inaccurate or incomplete [47]. This struggle has a complex relationship with self-efficacy, being the "conviction that one can successfully execute [a] behaviour required to produce [a desired] outcome" [3, p. 193]. Various factors interact with self-efficacy in unexpected ways [23], but it seems to worsen throughout introductory study [36, 43]. As such, many computing educators explore self-efficacy [34, 58] and its links to attainment [45], retention [11], and equality [1].

Tools that structure programming approaches, such as computer-aided software engineering (CASE), potentially offer a solution. However, there is a temptation to dive straight into professional tools due to a desire for 'authenticity' and 'industry-readiness' [17]. Few such tools seem to cater to the learnability needs of novices [21, 32, 37, 42, 48]. Instead, they are often designed for the needs of professionals [7, 59]. Techapalokul & Tilevich [54] explore this gap, investigating the recurring struggles of novices and implying an opportunity towards quality-oriented learning through tool-supported practice.

This paper examines the relationship between using CASE tools and debugging self-efficacy during novice troubleshooting. Focusing on plugins for Visual Studio, it compares no tool and an off-the-shelf tool to a prototype CASE tool simplified for a CS1 context. The tool strives beyond supporting students in identifying problematic code, working towards instructing students about quality-enhancing practices. It also provides explicit guidance on problem-solving [26], which situates programming as a six-stage process in which mental representations of problems are translated into solutions and encoded, with monitoring and visualisation to illustrate each stage [31]: reinterpret problem prompt; search of analogous problems; search for solutions; consider a potential solution; implement a solution; and evaluate the implemented solution. Further to this, help on debugging techniques is drawn from a manual for beginners [16] with prompting according to knowledge types identified in the debugging framework presented by Li *et al.* [29]. These resources help to situate and convey lessons on explicit problem-solving procedures and debugging techniques within a live development context. Shaping the learning experience *in-situ* ought to cultivate self-efficacy. However, the effectiveness of such design considerations in a CASE tool concerning debugging self-efficacy is not yet known.

2 LITERATURE REVIEW

2.1 Background

Self-efficacy is a general construct related to the conviction one has for meeting some criteria in completing a task [3]. However, due to its specificity to a particular task, it is usually operationally defined in a domain-specific way. Scott and Ghinea [51] suggest: "debugging self-efficacy captures learners' cognitive self-assessment of whether or not they are confident in their ability to write and debug simple programs" [p.126].

Though debugging self-efficacy has received less attention in the literature, computing educators have explored many forms of self-efficacy. These forms consistently correlate with several educational outcomes. Ramalingam, LaBelle, and Wiedenbeck [45] revealed the link between self-efficacy and attainment in the introductory programming context. More recent studies replicate this finding [57]. It is also associated with retention [28, 41], with several studies highlighting its role in retaining diverse cohorts, particularly for gender and ethnicity [20, 30, 39]. As such, self-efficacy is an important psychological construct with links to attainment, retention, and equality in computing.

The operation of self-efficacy constructs is complex, with even the students who produce good software sometimes developing low self-efficacies [23]. Theoretic models suggest reciprocating effects mediated by factors such as the design of learning and social environments, control-value appraisals, activity and outcome emotions, and competence-orientation [51]. It is essential to consider these aspects of a learning experience as they sometimes have opposing influences on self-efficacy, which can vary in their influence at different stages of learning.

Ultimately, appraisals drive change in self-efficacy. Such appraisals are not only based on perception of outcome but also experience. As such, pedagogies that strive to enhance debugging self-efficacy ought to implement strategies that facilitate positive appraisals of the self. Bandura [4, 6] suggests these include vicarious experiences, observing successful strategies that one believes they can enact (i.e., role models and exemplars); mastery experiences, successfully enacting a plan in a relevant context (i.e., performing well or feeling competent); emotional and physiological experiences, where affects become an "energising facilitator of performance" [5, p.5] rather than barriers (i.e., avoiding debilitating feelings); and social persuasion, feedback pertaining that instils conviction in one's capacity for success (i.e., encouragement and constructive critique).

Implementing these strategies in the debugging context is not trivial. *Passive* and *ad-hoc* approaches to teaching debugging are sub-optimal because they leave students vulnerable to unproductive troubleshooting methods and, therefore, the potential to make negative self-appraisals when they get stuck or become overwhelmed. Such occurrences are not uncommon as, typically, students encounter many difficulties when they start learning programming [50]. A common pitfall is that novice programmers sometimes write code as they go without considering a plan or taking a disciplined approach [53]. It isn't easy to envision how to solve a problem and then construct a computer program that enacts the solution [25]. However, neglecting these facets of programming is problematic. When already faced with an abundance of challenges and misconceptions that interfere with learning programming [44] as well as the feedback from the development environment, which seems to present more of a barrier than an aide [59], it becomes unsurprising that so many novices find themselves troubleshooting and debugging without direction [58].

Furthermore, as industry demands become increasingly complex, professionals must build computer systems in a disciplined way to ensure maintainability and stability. Fostering good practices early in the student journey is critical: "Without appropriate educational intervention, poor programming practices can persist and negatively affect the introductory learners" [54, p. 8]. Early intervention ensures that bad habits don't become entrenched and hinder later professional development. Studies support this view, showing the utility of educating novices on software quality fundamentals [55].

So, instruction is necessary to help learners avoid scenarios where they are unlikely to make positive self-appraisals or may stifle their future adoption of industry practice. However, a lecture in isolation is unlikely to change such behaviour radically. Instead, the learning design requires scaffolding in a way that cultivates deliberate practice [50]. One approach is to embed the relevant lessons into the tool. Situating learning like this shapes behaviour by directing attention to practices that minimise defects.

Explicit guidance [26] is a "human-executable procedure for accomplishing a programming task" [p.2]. Experimental evidence illustrates the benefits of such procedures to programming self-efficacy [31] and specifically on helping learners enact structured debugging [38]. As an instructional strategy, it offers affordances that align with methods for improving debugging self-efficacy [46]—providing a step-by-step structure to a programming task alongside contextualising guidance when students are encountering difficulty.

2.2 Related Work

Educators are already embedding debugging tutorials into integrated development environments [22], and there are even game-based approaches which show promise [8, 27]. There are few tools, however, that focus on quality acculturation and debugging self-efficacy in tandem. A review of 101 tools suggests that their feedback is often constrained to problem identification rather than remedial process [21]. Keuning *et al.* [21] argue there is a need for research and development into tools that advise students *how* to fix problems and to suggest *next steps* to improve quality. Other researchers seem to agree [29, 32], but take a more expansive and holistic perspective that also includes direction on problem-solving fundamentals [49] and coverage of a more appropriate and encompassing range of knowledge domains [29].

Intelligent and automated tutoring systems in the programming context present a considerable technical challenge. It is an active area of research, with many recent advances in the field of automatic program repair via tools such as CLARA [10]. Within a given problem domain, such systems could eventually match and interpolate between student code and candidate solutions, guiding students towards favourable, higher-quality code. Code generators, such as Codex and CoPilot, potentially offer means to increase the flexibility of the technology. This form of automated program repair shows a lot of promise [56]. However, they have many drawbacks

in learning contexts [14]. The distance from a working solution could be so far that the suggested repairs are inaccurate [56], and there are suspicions that the target code these generators produce could be defective [2].

Other solutions tend to orient around enriching feedback [40], with recent work focusing on generating hints [56] or describing strategies [26]. These often follow a mixed-initiative approach that coordinates human and computer action. However, they tend to produce high-volume feedback that could overwhelm a beginner. Among these are tools that help identify and address code smells [24]. These are violations of software design principles [15]. As some errors and warnings arise more frequently than others [13, 35], it is possible to classify them in a way that could indicate common violations or use graph matching to suggest repairs [10]. Furthermore, Hermans & Aivaloglou [19] suggest several code smells, particularly 'long method' and 'duplication of code', that arise in many projects completed by novices. Hence, using code smells as a metaphor might help establish a dialogue with novices and help them avoid common quality pitfalls.

It is also worth highlighting the widespread use of plugins in integrated development environments. Though easy to set up, many such plugins contrast with educational goals. Such "professional [tools] do not fulfil students' didactic needs, since they report abstract error messages that are targeted to expert programmers" [59, p.1]. As such, many novices may be unable to leverage plugins effectively and may even find them uninterpretable.

3 RESEARCH QUESTION & HYPOTHESES

The research question this paper strives to address is: to what extent does using a CASE tool influence the debugging self-efficacy of programming students during troubleshooting? By incorporating features that structure the learning experience to help promote positive self-appraisals whilst striving to minimise negative self-appraisals, this paper poses the following hypotheses: (H_1) there will be a difference in debugging self-efficacy before and after the debugging exercise; and (H_2) there will be a relationship between the tool used in the exercise (simplified tool for education; off-the-shelf tool; or no tool) and debugging self-efficacy after the exercise, controlling for debugging self-efficacy before the exercise.

4 CASE TOOLS

The first author implemented a prototype CASE tool[1] for this study, using agile methods [52] and a human-centred approach. This involved small-scale user testing through each sprint of development. Once they had produced a minimum viable product and assured its quality, they packaged it as a Visual Studio plugin. The tool is an extension for programming in C#, conducting static analysis, displaying messages in the Visual Studio interface, and appending to compiler output. Shown in Figure 1, the tool automatically computes standard software metrics when any source files in the project are saved, including maintainability index, class coupling and cyclomatic complexity. A supplementary static analysis also searches for code smells using a matching algorithm. Scores are stored in the project directory in an `XML` document and parsed using `LINQ`.

[1] https://github.com/Falmouth-Games-Academy/simplified-case-for-novices/

As metrics drop below a pre-defined threshold, the tool displays a plain text warning in the console at the bottom of the window. This warning encourages developers to revise their code alongside a list of which specific functions are at fault, a brief explanation, and suggestions. It provides advice until the code metric scores meet or exceed the pre-determined values. At this point, it reports no issues. An 'OK' status message is displayed in subsequent runs to reassure users that the tool is functioning. The tool does not restrict the user's ability to write code should they choose to ignore messages in the console.

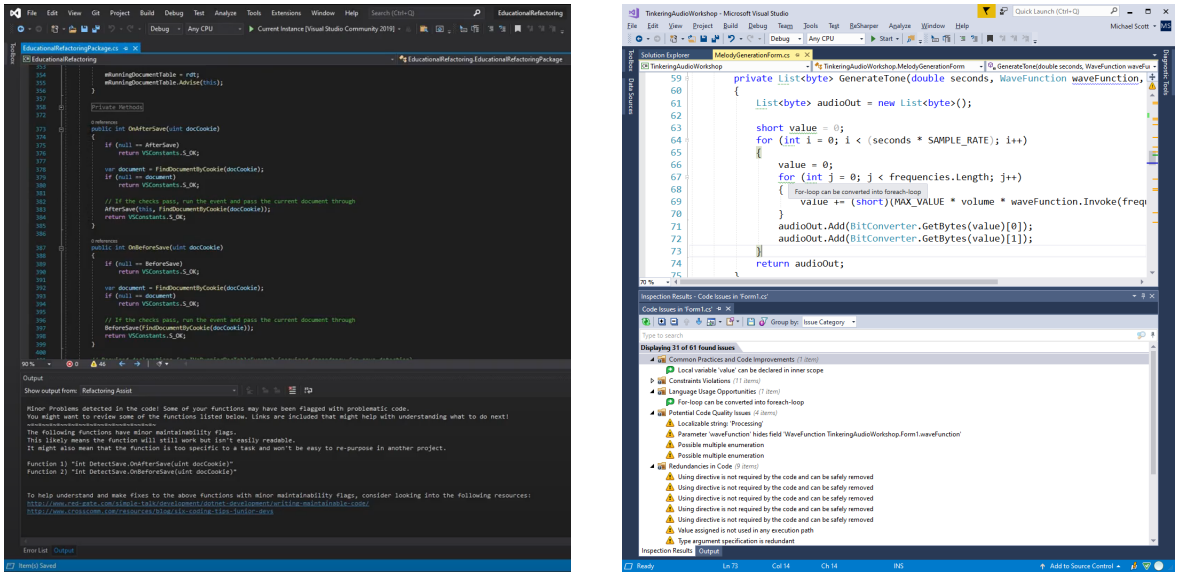
This research compares the prototype CASE tool to *ReSharper*, a professional tool designed to enhance the development process. Its features include code analysis, formatting, refactoring capabilities, cleanup, and boilerplate code generation. It also provides error detection, intelligent code suggestions, and search tools. It prominently displays a complex hierarchy window which shows warnings alongside test output across a project. Together, these features assure the consistency and readability of code while promising to boost productivity by automating repetitive tasks.

What distinguishes the prototype from this off-the-shelf tool is the following: (i) the tool has a simplified interface to aid novice learners, there are no complex menus to navigate or options that use terminology that novices may have issues understanding; (ii) it illustrates and walks learners through a six-stage process [31] to discourage them from coding without a plan—reinterpret problem, search for analogous problems, search for solutions, evaluate a potential solution, implement a solution, and evaluate the implementation—with an outline and a link to a short explanation of the currently selected stage; (iii) the filters change what is displayed depending on the currently active stage, such that prompts highlight errors in the implementation stage and highlight potential refactors in the evaluation stage, and so on, providing a more structured approach than off-the-shelf tools which tend to offer these features at all stages of development; and (iv) feedback messages have a more straightforward form that strives to be easily interpreted by novices (adapted from a debugging manual designed for CS1 students [16]). Furthermore, the tool can expand the feedback to provide links to learning resources specific to the detected code structural issue. Selecting a link opens a browser window directly to supplementary resources. These third-party resources on the web impart lessons on how to avoid a particular pitfall detected by the tool.

5 EVALUATION METHODOLOGY

5.1 Experimental Design & Exercise

The researchers followed an experimental methodology using quantitative data to address the research question. It followed a between-participant design due to concern about practice and preference effects. Each participant used only one tool to avoid such bias. The experimental design consisted of a parallel-group double-anonymous randomised trial, incorporating balanced allocation between three groups: those using the novel, more straightforward tool (experimental group), those using an off-the-shelf tool (alternative group), and those using no tool (control group). The experiment occurred on the university campus under the observation of research team members using the workstations



(a) Prototype Tool (Custom Visual Studio Extension)

(b) Off-the-Shelf Tool (ReSharper)

Figure 1: CASE Tools Used in the study

available in the main computer studio. Following a briefing and ethics declaration, participants completed the consent form and took a pre-test measure of debugging self-efficacy before the experiment. They were randomly allocated to a group and completed the exercise using the relevant tool. Participants also took a post-test measure of debugging self-efficacy after they had completed the exercises or if one hour had elapsed.

Participants completed a series of four small-scale tasks[2] which represented a typical practical workshop. The research team selected tasks based on beginner-level materials on string manipulation from the CS1 curriculum at their institution: repeating strings, reversing strings, inverting casing, and removing string characters. The course team had adapted these tasks from existing coding tests available on Codewars [12]. Learners reflected on their performance and optionally wrote notes as a final step.

5.2 Measurement

The independent variable is the tool used, determined by group allocation. The dependent variable is debugging self-efficacy. Debugging self-efficacy is measured using the relevant subset of items from Scott & Ghinea's measurement instrument [51]. The authors adapted two items in the scale to, respectively, better reflect the C# context and the troubleshooting tasks:

(1) I am confident that I can understand C# exceptions (e.g., NullReferenceException)
(2) I am confident I can solve simple problems with my programs
(3) I am confident I can implement a method from a description of a problem or algorithm
(4) I am confident I can debug a program that manipulates text according to rules

[2] https://raw.githubusercontent.com/Falmouth-Games-Academy/simplified-case-for-novices/main/ParticipantTestFolder/README.md

This instrument scores debugging self-efficacy using a five-point Likert scale, with the composite value being the mean of responses across the four questions. A high number (i.e., five) indicated positive debugging self-efficacy. The questions in the pre-test and post-test were the same to facilitate comparison. The researchers used Microsoft Office Forms to gather data per data protection requirements the local research ethics committee set.

5.3 Recruitment

The authors recruited students from the CS1 module at their institution. They promoted the study via institutional email, a computer-supported collaborative work platform, notices on the virtual learning environment, and a course-related social media group. Following ethical requirements, the study was voluntary, and participants provided informed consent.

5.4 Participants

A power analysis was conducted to determine a viable sample size for detecting a 'large' effect (see [9]). So, the authors recruited 66 cohort members for the experiment (40% of the cohort). They randomly allocated 22 students to each group. Participants were first-stage undergraduate students enrolled in computing courses in 2021-22 and 2022-23. They typically required at least 112 points on the University and College Admission System (UCAS) to enrol in these courses and had a pre-university mathematics entry requirement. The sampling frame consisted of all those students who had completed the first 15 of 30 weeks that formed the CS1 sequence. This frame ensured that the participants were novice programmers at a stage where they were familiar with programming syntax in C# but were unlikely to have mastered debugging or been familiar with CASE tools. All participants were volunteers and gave informed consent. Approximately 70%

identified as male, 18% as female, and 6% as another gender identity, with the rest (6%) preferring not to state. The mean age was 20.7 years, with a majority (82%) within the 18–21 age bracket, with the rest older. Most (87%) identified as white, and nearly 30% declared a disability. Contrasting these descriptive statistics from the sample with the known population for the two local CS1 cohorts (i.e., 86% identifying as male, mean age 19.6 years, 88% identifying as white, 30% declaring a disability) revealed no statistically significant discrepancies. However, it's notable that the general profile of the student intake during the pandemic and post-pandemic period included a slightly higher proportion of mature students.

6 RESULTS & ANALYSIS

Data were analysed using RStudio version 1.3.959. Statistical significance is determined using conventions (i.e., $p < .05$). Checks for normality, homogeneity of variance, and homoscedasticity verified that the assumptions for the paired t-test and the Analysis of Covariance (ANCOVA) held.

Paired t-tests compare pre-test and post-test debugging self-efficacy to evaluate the first hypothesis. These are shown in Table 1. The results indicate that the group using the more straightforward tool did not change in debugging self-efficacy ($p = .364, d = 0.55$). Similarly, the results indicate no change for the group using no tool ($p = .198, d = -1.3$). However, the group using the off-the-shelf tool experienced a statistically significant decline in debugging self-efficacy ($p = .020, d = -2.56$). Hattie [18] suggests that effect sizes where $|d| > .40$ are educationally relevant. As such, the data partially supports H_1.

An ANCOVA evaluates the second hypothesis. The results are shown in Table 2. They suggest a statistically significant difference in the mean change in debugging self-efficacy between the tools used whilst adjusting for pre-test levels of debugging self-efficacy, $F(2, 21) = 4.664 and p = 0.02$. The model has an adjusted R^2 of .49, predicting 49% of the variance. The total effect size for the model is $\eta_p^2 = .555$, but the effect size for the experimental grouping was $\eta_p^2 = .318$; thus, the tool used explains 32% of the variance, with prior levels of debugging self-efficacy explaining the remainder. As such, the data supports H_2.

To explore H_2 in greater depth, given three conditions, *post-hoc* tests compared the conditions. Due to the need to make multiple comparisons, a Bonferroni correction was applied. The results show that the simplified tool was statistically significantly different to the off-the-shelf tool ($p = .010$) and marginally different to the no tool ($p = .055$). The estimated marginal means are shown in Table 3. The estimated marginal mean z-score for the more straightforward tool is positive, while for the other groups, it is negative. It is worth noting the magnitude is relatively small, less than half of the total cohort standard deviation, but with noticeable mean differences between the groups.

A box plot is shown in Figure 2. It illustrates the difference in debugging self-efficacy between the pre-test and post-test, subtracting the pre-test score from the post-test score and standardising it to a z-score. Using the mean of the entire dataset (at $z = 0$) shows the relative differences between the group using the simplified tool, the off-the-shelf tool, and no tool. It also shows notable differences in variance. Those using the novel, more straightforward tool have a much smaller interquartile range, with outliers at each distribution end. The interquartile ranges of those using the off-the-shelf tool or no tool were more extensive but skewed, with medians below zero indicating the reduction of debugging self-efficacy. Generally, those using the off-the-shelf tool developed lower debugging self-efficacy.

Table 1: Paired T-Test Comparing Pre and Post-Debugging Self-Efficacy (Z-Score) by group

Group	$\Delta_{\bar{x}}$	Δ_s	t	df	p	d
Simplified Tool	.276	1.032	.927	21	.364	0.55
Off-the-Shelf Tool	-.663	.558	-3.144	21	.004	-2.56
No Tool	-.364	.613	-1.329	21	.198	-1.33

Pre-test is a covariate in the model, estimated using $z = 0$

Table 2: Analysis of Covariance (ANCOVA)

Factor	df	Sum Sq	Mean Sq	F	p	η_p^2
Group	2	195.69	97.85	4.664	.022	.318
Pre-Test	1	243.42	243.42	6.193	.003	.367
Residuals		419.6	20.98			

Table 3: Estimated Marginal Mean of Post-Test Debugging Self-Efficacy (Z-Score) by group

Group	$\bar{x}$	Std Error	CI_{lower}	CI_{upper}
Simplified Tool	.447	.208	0.013	1.803
Off-the-Shelf Tool	-.522	.270	-1.086	0.042
No Tool	-.342	.324	-1.017	.333

Pre-test is a covariate in the model, estimated using $z = 0$

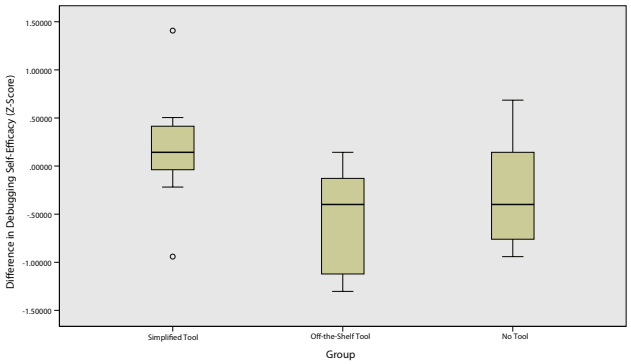


Figure 2: Box Plot Comparing Differences in Pre and Post Scores on Debugging Self-Efficacy Across the Tool Use Groups

7 FINDINGS

The data shows a statistically significant difference in how CASE tools interact with debugging self-efficacy when completing a short troubleshooting exercise. The data shows that tailoring a CASE tool for novices seems to help preserve existing debugging self-efficacy during a brief activity. However, the data reinforces the

notion that debugging self-efficacy *declines* considerably when using an off-the-shelf tool designed for professionals. This finding accords with previous findings that educators should be cautious when introducing their students to tools used by professionals [59]. While relevant and authentic, they can dampen self-efficacy when they interfere with rather than aid a troubleshooting process. This phenomenon will likely arise when learners misunderstand the tools' vocabulary or lack a mental model of what the tool is doing.

An observation during the experiment was that there were stark differences in the way students interacted with the CASE tools. Many participants ignored the tool, regardless of which they had access to. Others expressed their displeasure with tips and recommendations, seeming to question their validity. From this, the authors speculate that this may be a function of prior experience or prior debugging self-efficacy, such that those with high self-efficacy want to break away from support structures whilst those with low self-efficacy want to brute-force tackle challenges without novelties or distractions. It would be worthwhile to use qualitative methods to investigate this phenomenon further. It would also be helpful to embed analytics to verify who engages with tools and whether that influences their success or any change in debugging self-efficacy.

The relationship appears to be rather complex. Though the prototype did not illustrate the anticipated gain between the pre-test and post-test measurements, there are two exciting observations. Firstly, the group using the experimental prototype had relatively low variance. Secondly, the confidence interval for the estimated marginal means for the experimental group using the simplified tool was positive (as shown in Table 3). These observations might suggest that most of those with prior debugging self-efficacy above the mean would see a modest increase. As such, the tool may have prevented learners from making negative self-appraisals. However, it may not have helped those with lower self-efficacies from making positive self-appraisals, perhaps as reliance on the tool diminished mastery attributions. Further study to explore the way these attributions are made based on prior debugging self-efficacy is needed to verify this interpretation.

A limitation of this work is the generalisation of these results to other contexts. The authors conducted the study within a single institution and compared only two tools. As such, they sought to maximise internal validity rather than external validity. Additionally, the method is cross-sectional, representing a potential change vector from a single learning experience. Learning is longitudinal, so further work examining a range of tools across longer periods would yield more insights into how debugging self-efficacy changes throughout students' journeys.

8 CONCLUSION

Debugging is a considerable stumbling block for novice programmers, and CASE tools have the potential to help. They can guide novices and help them overcome obstacles. However, this study reinforces that some tools can deleteriously impact novice programmers' debugging self-efficacy. So much so that it is worse than using no tool at all. These findings accord with previous work showing that novices find such tools cryptic and unwieldy. Though the effect sizes are concerning, showing significant negative impacts beyond those typically seen in short-form exercises.

In comparison, a prototype tool designed for education preserved debugging self-efficacy. Whilst there was no statistically significant change, there are indications that those with mean levels of prior debugging self-efficacy benefited in a small way, and the overall variance in change score was lower. However, simplification was insufficient to yield an improvement in debugging self-efficacy. The findings suggest that educators should carefully consider using professional tools in contexts where they might undermine the self-efficacy of their students. They also suggest that educational tool designers should consider how self-efficacy might shape the way learners engage with their tools.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2019. Early programming education and career orientation: the effects of gender, self-efficacy, motivation and stereotypes. In *Proceedings of the 50th ACM technical symposium on computer science education*. ACM, New York, NY, 679–685. http://doi.org/10.1145/3287324.3287358

[2] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2023. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 24 pages. http://doi.org/10.1007/s10664-023-10380-1

[3] Albert Bandura. 1977. Self-efficacy: toward a unifying theory of behavioral change. *Psychological review* 84, 2 (1977), 191. http://doi.org/10.1037/0033-295X.84.2.191

[4] Albert Bandura. 1982. Self-efficacy mechanism in human agency. *American psychologist* 37, 2 (1982), 122. http://doi.org/10.1037/0003-066X.37.2.122

[5] Albert Bandura. 1995. Exercise of personal and collective efficacy in changing societies. *Self-efficacy in changing societies* 15 (1995), 334. http://doi.org/10.1017/cbo9780511527692.003

[6] Albert Bandura, Nancy E Adams, and Janice Beyer. 1977. Cognitive processes mediating behavioral change. *Journal of personality and social psychology* 35, 3 (1977), 125. http://doi.org/10.1037/0022-3514.35.3.125

[7] Kim B. Bruce. 2018. Five Big Open Questions in Computing Education. *ACM Inroads* 9, 4 (2018), 77–80. http://doi.org/10.1145/3230697

[8] Chiung-Fang Chiu and Hsing-Yi Huang. 2015. Guided debugging practices of game based programming for novice programmers. *International Journal of Information and Education Technology* 5, 5 (2015), 343. http://doi.org/10.7763/IJIET.2015.V5.527

[9] Jacob Cohen. 1992. A power primer. *Psychological Bulletin* 112, 1 (1992), 155–159. http://doi.org/10.1037//0033-2909.112.1.155

[10] Maheen Riaz Contractor and Carlos R Rivero. 2022. Improving Program Matching to Automatically Repair Introductory Programs. In *International Conference on Intelligent Tutoring Systems*. Springer, Cham, Switzerland, 323–335. http://doi.org/10.1007/978-3-031-09680-8_30

[11] S Joseph DeWitz, M Lynn Woolsey, and W Bruce Walsh. 2009. College student retention: An exploration of the relationship between self-efficacy beliefs and purpose in life among college students. *Journal of college student development* 50, 1 (2009), 19–34. http://doi.org/10.1353/csd.0.0049

[12] Nathan Doctor and Jake Hoffner. 2022. *Codewars*. Andela. www.codewars.com

[13] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects across Introductory Programming Topics. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. ACM, New York, NY, 941–947. http://doi.org/10.1145/3478431.3499415

[14] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. ACM, New York, NY, 10–19. http://doi.org/10.1145/3511861.3511863

[15] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, MA. https://dl.acm.org/doi/book/10.5555/311424

[16] Rita Garcia, Chieh-Ju Liao, and Ariane Pearce. 2022. Read the Debug Manual: A Debugging Manual for CS1 Students. In *2022 IEEE Frontiers in Education Conference (FIE)*. IEEE, New York, NY, 1–7. http://doi.org/10.1109/FIE56618.2022.9962675

[17] Mark Guzdial and Allison Elliott Tew. 2006. Imagineering inauthentic legitimate peripheral participation: an instructional design approach for motivating computing education. In *Proceedings of the second international workshop on Computing education research*. ACM, New York, NY, 51–58. http://doi.org/10.1145/1151588.1151597

[18] John Hattie. 2008. *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Routledge, Milton, UK. http://doi.org/10.4324/9780203887332

[19] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th*

International Conference on Program Comprehension (ICPC). IEEE, New York, NY, 1–10. http://doi.org/10.1109/ICPC.2016.7503706

[20] Mica A Hutchison, Deborah K Follman, Melissa Sumpter, and George M Bodner. 2006. Factors influencing the self-efficacy beliefs of first-year engineering students. *Journal of Engineering Education* 95, 1 (2006), 39–47. http://doi.org/10.1002/j.2168-9830.2006.tb00876.x

[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43. http://doi.org/10.1145/3231711

[22] Olli Kiljunen. 2021. Teaching Students to Fix Programming Errors with Tutorials Embedded in an IDE. In *21st Koli Calling International Conference on Computing Education Research*. ACM, New York, NY, 32. http://doi.org/10.1145/3488042.3489969

[23] Päivi Kinnunen and Beth Simon. 2012. My program is ok–am I? Computing freshmen's experiences of doing programming assignments. *Computer Science Education* 22, 1 (2012), 1–28. http://doi.org/10.1080/08993408.2012.655091

[24] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610. http://doi.org/10.1016/j.jss.2020.110610

[25] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. *SIGCSE Bull.* 37, 3 (jun 2005), 14–18. http://doi.org/10.1145/1151954.1067453

[26] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449.

[27] Michael J. Lee. 2014. Gidget: An online debugging game for learning and engagement in computing education. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, 193–194. http://doi.org/10.1109/VLHCC.2014.6883051

[28] Robert W Lent, Steven D Brown, and Kevin C Larkin. 1986. Self-efficacy in the prediction of academic performance and perceived career options. *Journal of counseling psychology* 33, 3 (1986), 265. http://doi.org/10.1037/0022-0167.33.3.265

[29] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a Framework for Teaching Debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '19)*. ACM, New York, NY, 79–86. http://doi.org/10.1145/3286960.3286970

[30] Guan-Yu Lin. 2016. Self-efficacy beliefs and their sources in undergraduate computing disciplines: An examination of gender and persistence. *Journal of Educational Computing Research* 53, 4 (2016), 540–561. https://doi.org/10.1177/0735633156084

[31] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, CA) *(CHI '16)*. ACM, New York, NY, 1449–1461. http://doi.org/10.1145/2858036.2858252

[32] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 55–106. http://doi.org/10.1145/3293881.3295779

[33] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2011. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education* 21, 1 (2011), 57–80. http://doi.org/10.1080/08993408.2011.554722

[34] Lauri Malmi, Judy Sheard, Päivi Kinnunen, and Jane Sinclair. 2020. Theories and Models of Emotions, Attitudes, and Self-efficacy in the Context of Programming Education. In *Proceedings of the ACM Conference on International Computing Education Research*. ACM, New York, NY, 36–47. http://doi.org/10.1145/3372782.3406279

[35] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, New York, NY, 1–8. http://doi.org/10.1109/FIE.2014.7044420

[36] Dawn McKinney and Leo F Denton. 2004. Houston, we have a problem: there's a leak in the CS1 affective oxygen tank. *ACM SIGCSE Bulletin* 36, 1 (2004), 236–239. http://doi.org/10.1145/971300.971386

[37] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (2019), 77–90. http://doi.org/10.1109/TE.2018.2864133

[38] Tilman Michaeli and Ralf Romeike. 2019. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*. ACM, New York, NY, 1–7. http://doi.org/10.1145/3361721.3361724

[39] An Nguyen and Colleen M Lewis. 2020. Competitive enrollment policies in computing departments negatively predict first-year students' sense of belonging, self-efficacy, and perception of department. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 685–691. http://doi.org/10.1145/3328778.3366805

[40] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40. http://doi.org/10.1145/3513140

[41] Frank Pajares and Dale H Schunk. 2001. Self-beliefs and school success: Self-efficacy, self-concept, and school achievement. In *Self Perception*, Richards Riding and Stephen Rayner (Eds.). Greenwood Publishing Group, Westport, CT, Chapter 11, 239–266. http://books.google.co.uk/books?id=IEbPEAAAQBAJ&pg=PA239

[42] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*. ACM, New York, NY, 204–223. http://doi.org/10.1145/1345443.1345441

[43] Markeya S. Peteranetz, Leen-Kiat Soh, Duane F. Shell, and Abraham E. Flanigan. 2021. Motivation and Self-Regulated Learning in Computer Science: Lessons Learned From a Multiyear Program of Classroom Research. *IEEE Transactions on Education* 64, 3 (2021), 317–326. http://doi.org/10.1109/TE.2021.3049721

[44] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24. http://doi.org/10.1145/3077618

[45] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. ACM, New York, NY, 171–175. http://doi.org/10.1145/1007996.1008042

[46] Guity Ravai, Ludmila Nunes, and Ronald Erdei. 2017. The Introduction of Informal Cooperative Learning into our Programming Laboratories. In *18th Annual Midwest Scholarship of Teaching and Learning (SoTL) Conference* (South Bend, IN). Indiana University, South Bend, IN, 21 pages. http://core.ac.uk/download/pdf/83144757.pdf

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

[48] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172. http://doi.org/10.1076/csed.13.2.137.14200

[49] Juan Pablo Saenz and Luigi De Russis. 2022. On How Novices Approach Programming Exercises Before and During Coding. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, New York, NY, 1–6. http://doi.org/10.1145/3491101.3519655

[50] Michael James Scott and Gheorghita Ghinea. 2013. Educating programmers: A reflection on barriers to deliberate practice. In *Proceedings of the 2nd Annual HEA STEM Conference* (Birmingham, UK). Higher Education Academy, Cork, Ireland, 1–6. http://doi.org/10.48550/arxiv.1311.0390

[51] Michael James Scott and Gheorghita Ghinea. 2014. Measuring enrichment: the assembly and validation of an instrument to assess student self-beliefs in CS1. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ACM, New York, NY, 123–130. http://doi.org/10.1145/2632320.2632350

[52] James Shore, Diana Larsen, Gitte Klitgaard, and Shane Warden. 2021. *The Art of Agile Development*. O'Reilly Media, Sebastopol, CA. http://dl.acm.org/doi/book/10.5555/1407480

[53] Elliot Soloway and James C Spohrer. 2013. *Studying the novice programmer*. Psychology Press, New York, NY. http://doi.org/10.4324/9781315808321

[54] Peeratham Techapalokul and Eli Tilevich. 2017. *Understanding recurring software quality problems of novice programmers*. Technical Report. Virginia Polytechnic Institute. http://hdl.handle.net/10919/78337

[55] Eli Tilevich, Peeratham Techapalokul, and Simin Hall. 2020. Teaching the culture of quality from the ground up: Novice-tailored quality improvement for scratch programmers. In *2020 ASEE Annual Conference & Exposition*. ASEE, Washington, DC, 16 pages. http://doi.org/10.18260/1-2--35283

[56] Daniel Toll, Anna Wingkvist, and Morgan Ericsson. 2020. Current state and next steps on automated hints for students learning to code. In *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, New York, NY, 1–5. http://doi.org/10.1109/fie44824.2020.9274053

[57] Chun-Yen Tsai. 2019. Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior* 95 (2019), 224–232. http://doi.org/10.1016/j.chb.2018.11.038

[58] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 73–79. http://doi.org/10.1145/3408877.3432374

[59] Stelios Xinogalos and Maya Satratzemi. 2004. Introducing novices to programming: a review of teaching approaches and educational tools. In *2nd International EISTA Conference*. IIIS, Winter Garden, FL, 60–65. https://www.researchgate.net/profile/Maya-Satratzemi/publication/27380823