An approach for detecting student perceptions of the programming experience from interaction log data

Jamie Gorson, Nicholas LaGrassa, Cindy Hsinyu Hu, Elise Lee,
Ava Marie Robinson, and Eleanor O'Rourke

Northwestern University, Evanston IL, USA
{jgorson, nick.lagrassa, cindyhu2023, eliselee, avarobinson2021}
@u.northwestern.edu, eorourke@northwestern.edu

Abstract. Student perceptions of programming can impact their experiences in introductory computer science (CS) courses. For example, some students negatively assess their own ability in response to moments that are natural parts of expert practice, such as using online resources or getting syntax errors. Systems that automatically detect these moments from interaction log data could help us study these moments and intervene when the occur. However, while researchers have analyzed programming log data, few systems detect pre-defined moments, particularly those based on student perceptions. We contribute a new approach and system for detecting programming moments that students perceive as important from interaction log data. We conducted retrospective interviews with 41 CS students in which they identified moments that can prompt negative self-assessments. Then we created a qualitative codebook of the behavioral patterns indicative of each moment, and used this knowledge to build an expert system. We evaluated our system with log data collected from an additional 33 CS students. Our results are promising, with F1 scores ranging from 66% to 98%. We believe that this approach can be applied in many domains to understand and detect student perceptions of learning experiences.

Keywords: CS education · Detection systems · Self-efficacy · Self-assessment

1 Introduction

While programming skills are increasingly important for 21st century learners, many students struggle in introductory computer science (CS) courses [24, 32]. Recent studies suggest that this struggle may be exacerbated by students' self-perceptions; students often believe that they do not belong [54, 30, 48], are not capable of succeeding [34, 19, 13], or are performing poorly in CS [21, 26, 31, 32]. In this paper, we focus on one aspect of student self-perceptions: negative self-assessments. In our previous work, we found that students frequently assess their own programming ability, using moments that occur during the programming process as signals of whether they are performing well [22, 21]. However, many

of the moments that students see as negative performance indicators are natural parts of expert practice; for example, many students believe that spending time planning and struggling with syntax errors are signs of low ability [28, 41, 52]. Students who negatively self-assess more frequently also tend to have lower self-efficacy [22], which can influence persistence and performance in CS [31, 37].

While our previous survey studies have established the prevalence of negative self-assessments in CS [22, 21], we have a limited understanding of the programming moments that prompt negative self-assessments. If we could detect these moments as they arise during the programming process, we would be able to study them more directly. Furthermore, if such a detection system were automated, we could study these moments using a significantly larger sample of data, as manually detecting them is labor-intensive. An automated detection system would also enable the development of real-time feedback interventions, which provide messages to students at key moments. This type of intervention has been shown to be effective in mediating student perceptions in other contexts [39, 11] and can scale to meet the increasing demand in CS.

Interaction log data collected from programming environments may be useful for automatically detecting self-assessment moments, since researchers have successfully leveraged this type of data to analyze student programming process [9, 8, 7], predict student performance [23, 9, 1, 38, 42], and build automated feedback interventions [14, 35]. However, most of these prior systems use bottom-up methods to identify behavioral patterns in interaction data, rather than using top-down approaches to detect pre-defined programming moments like struggling with syntax errors, an example of the self-assessment moments. Systems that use top-down approaches, such as cognitive tutors [3, 2, 10], generally require models of expert practice. However, researchers are not experts in understanding how students perceive the programming process, and thus we would need to elicit this knowledge from students to create such a model in this domain.

To address these challenges, we contribute an approach called *retrospective-enabled perception recognition* for designing systems that detect student perceptions of the programming process. In this approach, the designer uses retrospective interviews [16] to elicit student perceptions of programming moments, and then builds a qualitative codebook that describes the behavioral patterns indicative of each moment. This codebook is used to inform the design of an expert system. We used our approach to design an automated detector for eight self-assessment moments based on retrospective interviews with 41 CS students. We evaluated the performance of our system using data collected from an additional 33 students, comparing the automatically detected moments to those manually labelled by the authors. Our results are promising, with F1 scores ranging from 66% to 98%. We also present an analysis of our systems' incorrect decisions, enabled by the transparency of the expert system approach. Our detection system has the potential to facilitate future studies of self-assessment moments and support interventions that provide real-time feedback. Our findings also suggest that the *retrospective-enabled perception recognition* approach can be used to design detection systems for student perceptions in other contexts in the future.

2 Background

2.1 Student self-perceptions in CS

Computing education researchers have found that students often have negative perceptions about themselves and their experiences in CS. For example, when student perceptions of a programming session do not align with their expectations, students sometimes have negative emotional reactions even after successfully solving problems [25]. Additionally, students who have community-oriented goals often perceive that they can not meet these goals in computing careers [29]. Studies also show that many students perceive that they do not belong in computer science, often because they belong to a group that is underrepresented in the field [36, 15, 48, 18, 53]. These negative perceptions have been shown to correlate with students' self-efficacy [4, 25, 26, 54], or the belief in one's ability to accomplish a task or achieve mastery in a specific domain [5, 6, 47]. Self-efficacy has a direct impact on student learning outcomes [45] and often correlates with student performance in CS courses [55, 33, 43].

In this paper, we focus on students' perceptions of programming experiences that prompt them to negatively assess their ability. CS1 students assess their programming ability frequently [21, 26], and often think they are performing poorly when they encounter programming moments essential to the programming process [21, 22]. Our recent survey study with 214 CS students from three universities identified 13 programming moments which cause many students to negatively self-assess, even though the moments are also natural parts of expert practice [28, 41, 52]. For example, some students report that they feel like the are performing poorly when they use online documentation to look up syntax, stop to think about their solution, and spend time planning [22]. We also found that students who negatively self-assess in response to more of these moments tend to have lower self-efficacy. However, we do not know how these moments arise or when they occur in students' programming process.

2.2 Analyzing programming interaction log data

Researchers have explored many methods for interpreting log data collected from programming environments. This interaction data is used for two primary purposes: to produce new knowledge from a bottom-up analysis of student interactions, and to perform top-down detection of programming moments.

Many researchers take data-driven approaches to study the student programming process [8, 7, 51, 20] and to evaluate or predict student performance [23, 9, 1, 38, 42, 55]. Initially, most of this work analyzed compilation logs [23, 51], but more recently, researchers have leveraged machine learning techniques to identify patterns in interaction log data. For example, Blikstein et al. clustered students based on their problem-solving pathways to study how they progressed through programming assignments [9]. Berland et al. also used clustering techniques to study tinkering and how programming behaviors change across stages of the problem [7]. These studies used a bottom-up approach, analyzing data to find

patterns organically rather than building hand-architected models to identify preconceived moments of interest.

Some researchers have used interaction log data and expert knowledge of the programming process to identify pre-defined moments through top-down approaches. Expert systems, a common technique, reason about student interactions based on models of expert decision-making processes. For example, cognitive tutors like the LISP tutor [44] use expert systems to provide relevant feedback. Marwan et al. used a similar approach to analyze program states to identify milestones in student progress while solving problems [35]. Koskal et al., however, demonstrated how challenging it can be to build systems that detect pre-defined programming moments [27]. The authors set out to develop a system to automatically detect the stages in the *design recipe* [17], a scaffolded process for solving programming problems. However, they found that the fuzzy design recipe stages were hard to automatically detect from low-level log data due to the wide variation in student behaviors during each stage [27].

While previous studies show promise in deriving indicators of student behavior and process from low-level data, existing approaches do not yet explore how to use log data to automatically detect moments based on student perceptions. Expert systems are designed to model expert knowledge, but researchers are not experts in understanding how students perceive the programming process. As a result, we need an approach for eliciting this knowledge from students. We contribute a new approach for designing systems that use interaction log data to detect programming moments that students perceive as meaningful.

3 Retrospective-enabled perception recognition

The main contribution of this paper is our approach for detecting student perceptions of the programming experience from interaction log data. In this section, we describe our new approach and present the methods we used to build a system to detect moments when students may negatively self-assess while programming.

To enable our system, we designed extensions to collect interaction log data from two programs: jGRASP [12] (an IDE often used in introductory Java courses) and Chrome (a commonly used web browser). We chose these two programs because they account for a large portion of student interactions with the computer while programming. Each extension collects time-series data in a JSON format for a number of user actions and events, which allows us to keep track of student behavior and the state of the IDE. Our jGRASP extension, built in collaboration with the jGRASP development team, captures all keystrokes, cursor movements, console messages, and interactions with buttons and windows. Our Chrome tool captures all navigation on websites, including the URLs and scrolling behavior while viewing a page. During the data collection process, we iterated on the events and actions collected by the extensions as we learned more about the behaviors associated with each moment. For example, after looking at the data, we realized that student scrolling patterns revealed important information about their behavior, so we added this to our extensions.

3.1 Phase 1: Retrospective interviews

We conducted retrospective interviews during Phase 1 to capture student perceptions of the programming experience. We recruited 41 participants from a large public university in the United States. At the time of the study, all participants enrolled in a second-semester introductory CS course (CS2), a requirement for CS majors, were eligible to participate. We recruited students with emails sent by the professor of the course. The study took place virtually through Zoom. Students provided consent to participate and were compensated for their time.

The goal of the interview was to gather examples of self-assessment moments naturally occurring during programming sessions, along with participants' perceptions of those moments. When a participant joined the Zoom call, the researcher installed the Chrome and jGRASP extensions on the student's computer. Then the researcher provided a short review of how to use jGRASP to ensure a baseline level of familiarity with the development environment. We asked the student to work on one of three similar programming problems while sharing their screen, and told them to work on the problem like they would a homework assignment. During this part of the interview, the researcher turned the student's video and microphone off and did not interrupt them to reduce the effect of the lab environment on their behavior as much as possible.

After 30 minutes of programming, we conducted a retrospective interview [16]. We gave the student a list describing a subset of the self-assessment moments from Gorson & O'Rourke [22] (see examples in Table 1). We chose to only include the moments that occur during the programming process, like *changing approaches*, and not general reflections, like *spending a long time on a problem*, because we were more likely to be able to determine when they will happen. Finally, the student and researcher watched a screen recording of the programming session and the student identified each time one of those moments occurred. Below in Figure 1, we provide an example of the self-assessment moments that were labelled in the retrospective interview for one participant.

Table 1. Negative self-assessment moments detected by our expert system.

Moments *and detailed descriptions*
Using resources to look up syntax *from the web or other sources*
Using resources to research an approach *from the web or other sources*
Changing approaches *to try a new approach for solving the programming problem*
Writing a plan *in the comments or notes to outline future programming steps*
Getting simple errors *which are usually compiler errors due to oversights or typos*
Getting Java errors *which are usually runtime errors due to conceptual mistakes*
Struggling with errors *while trying to fix or debug the errors*
Stopping to think *while implementing a solution*

Fig. 1. The self-assessment moments that occured in one participant interview.

3.2 Phase 2: Qualitative analysis

The goal of Phase 2 was to develop a qualitative codebook that the researchers could use to identify negative self-assessment moments independently, without additional knowledge of student perceptions. Identifying moments such as using resources may appear straightforward, however students' perceptions of these moments are quite nuanced. For example, in our prior work students reported different reactions when using resources to look up syntax versus using resources to research how to solve the problem [21, 22]. While it is relatively easy to determine when a student is viewing a website or a course resource, determining the purpose of its use is more difficult. In addition, it is critical to identify each use of a resource, because a student who references the same resource multiple times will have a different experience than a student who uses multiple resources for different purposes. We therefore use a detailed qualitative codebook to capture the nuances discovered through the retrospective interview process

To develop this codebook, we qualitatively analyzed the retrospective interviews. After conducting the first 20 interviews, we compiled a list of all student-labeled moments. From that list, we distilled a set of representative behaviors for each moment and wrote an initial draft of the codebook. The codebook includes a high-level definition of each moment and a set of heuristics that describe the behavioral patterns indicative of each moment. We then re-watched the first twenty interviews and iterated on the behavioral descriptions for each moment until two researchers could accurately and consistently label all of the moments.

As an example, we describe how we identify *struggling with errors* using our codebook. We defined three levels of behavioral indicators for this moment: strong, medium, and weak. If a student exhibits a strong indicator, such as running code in an attempt to fix a bug three times in a row without succeeding, we would label this as *struggling with errors*. If there is no strong indicator, but there are two medium indicators, such as using resources after getting an error, we would also label this as *struggling with errors*. Finally, while weak indicators, such as a slower pace of typing, are not enough to label the moment on their own, the researchers use them to strengthen their confidence in the decisions.

3.3 Phase 3: Codebook verification

In Phase 3, we first tested the codebook using data from an additional 21 interviews. After each new interview, two authors watched the screen recording of the programming session and used the codebook to label the self-assessment moments. Then, the researchers compared their decisions to the participant's labels in the retrospective interview as member-checks of the labelling scheme [49]. When there were misalignments between a participant's labels and the researchers' labels that could not be explained by the participant misusing or missing a label, the researchers adjusted the description of that moment to incorporate the newly observed behavior. This iterative process continued until the researchers did not need to make changes for five consecutive interviews in which the moment was present. At that point, we considered the codebook for

that moment to have reached saturation [46, 40]. Of the 12 moments that we asked student to label during the retrospective interview, we were able to reach saturation for eight (see Table 1). Most of the moments for which we did not reach saturation occurred at the beginning of the programming session, such as *writing a plan before implementation*. At this point, students generally interact less with the computer, making it more difficult to identify these moments.

3.4 Phase 4: Implementation of the detection system

In Phase 4, we built an expert system to detect self-assessment moments using the heuristics in our qualitative codebook. Our system has two stages: data transformation and decision-making. In the data transformation stage, we parse through each event captured in the interaction log data, recreating the programming session and recording around 100 human-authored metrics into a knowledge base. Together the metrics provide a comprehensive snapshot of the state of the programming process. For example, one metric captures the number of lines that a student pastes from a resource into their code. In the decision-making stage, we analyze the metrics at each log event to determine if any of the self-assessment moments occurred. We use two different styles of heuristic algorithms, either if-then rules when there is less ambiguity in the decision-making process (e.g. *getting simple errors*), or fuzzy logic [56] when many metrics need to be considered in parallel (e.g. *using resources to look up syntax*). For example, we use fuzzy logic to increase our confidence that a student is using a resource to look up syntax if they paste either one or two lines of code from the resource.

As a concrete example, consider the strong indicator for the *struggling with errors* moment, when a student runs the code in an attempt to fix a bug three times in a row. One metric for this indicator calculates whether the student is working on the same error across multiple compilations. This metric keeps track of the number of the errors in the console and the names of the errors. After each compile, we use this information (along with some additional details about code edits) to evaluate if the student is still working on the same bug.

We chose an expert system because retrospective data is time-intensive to collect. It is impractical to collect enough student-labeled data to serve as ground-truth for machine learning algorithms. Additionally, data-driven approaches often produce features that are not human-interpretable, making it difficult to understand their decisions and limitations. With an expert system, we can trace the decision process and ensure that the system is making logical choices.

4 Evaluation of the system

4.1 Methods

We evaluated our system by comparing the automatically detected moments to those manually labelled by the authors. While researchers can make mistakes in labeling, this data is the most reliable item of comparison, as participant-labelled

data is often inconsistent due to differing interpretations of the moments and participant attention spans. We collected data from programming sessions with 33 additional students from the same university and CS2 course as our initial interviews. The setting and procedure were the same, with the exception of the retrospective interview, which was excluded. To establish the reliability of the researcher-assigned labels, two authors independently labelled the same seven interviews, or 21% of this data set, achieving 82% agreement. Those authors then authors independently labelled the remainder of the data.

One challenge in evaluating this system is establishing a way to compare moment timing between the researchers and the machine. When manually labelling the moments, the researchers picked a timeslot from non-overlapping ten-second windows (e.g., 0-10, 10-20). When comparing the system's results to the researcher-labelled set, we used an additional fifteen-second buffer on both sides of the ten-second window because the start time of a moment can be difficult to determine and might fall on the border of a window. We marked a machine detection as correct if the timestamp assigned to a label was within this forty-second window. We used a slightly larger buffer to more accurately represent two of the moments. For *changing approaches*, we used a two-minute buffer instead of a fifteen-second buffer because this moment often takes places over a few minutes, and we did not have a way to consistently identify matching start times. For *struggling with errors*, the researchers identified the start and end time for the error cycle in which the participant struggled. We deemed a system-identified label as correct if the system chose any time within the error-cycle boundaries. While both of these windows are larger, they reflect the context of these moments and the system's ability to identify these moments accurately.

After running our system on the log data from our evaluation data set, we further analyzed its performance by looking at each false positive and false negative result. The authors reviewed each case and categorized the reason for the false detection by watching the screen recording of the moment and consulting the codebook. During this process, we identified a number of instances when the researchers mislabeled moments, and also noted the limitations of our system.

4.2 Findings

Our results in Table 2 show that we had very high F1 scores for some moments, such as *getting simple errors*, and lower but still reasonable F1 scores for others, such as *writing a plan*. While precision and recall are both important, high precision matters most for interventions to ensure that real-time messages are delivered in response to true moments, and recall is most important for studies to ensure that relevant moments are not missed. The data also shows that the moments arise at varying levels of frequency; *getting simple errors* and *stopping to think* were most frequent, while *writing a plan* and *using resources to research an approach* only occurred occasionally. Our system tended to perform worse for less frequent moments, likely because our codebook and system were developed using fewer observations. However, the frequency of a moment does not necessarily indicate its importance. While we do not yet know how each moment

Table 2. Results from our evaluation of the detection system

Moment	Precision	Recall	F1 Score	Count	Human Errors
Using resources to look up syntax	82.0%	86.1%	84.0%	128	2
Using resources to research an approach	66.7%	66.7%	66.7%	21	1
Changing approaches	73.1%	73.1%	73.1%	26	8
Writing a plan	60.0%	75.0%	66.7%	15	0
Getting simple errors	99.1%	97.7%	98.4%	213	13
Getting Java errors	90.3%	90.3%	90.3%	31	2
Struggling with errors	69.2%	90.0%	78.3%	26	5
Stopping to think	79.1%	75.3%	77.2%	159	15

influences student self-efficacy, some of the less frequent moments may have a stronger impact on student experiences than the more frequent ones.

One benefit of our approach is that our system's decisions are transparent and can be assessed using our qualitative codebook. This enabled us to conduct an analysis on our system's false positives and false negatives. First, our analysis revealed many human errors in labeling, showing how challenging it is for humans to accurately label this type of data and highlighting the value of an automated system. Our analysis also revealed trends that provide direction for improving the system. For example, 10% of the system's incorrect decisions occured because the researcher and system disagreed about the timing of a moment. When we designed the codebook, we focused on describing the heuristics to determine whether a moment occurred, rather than the exact start time for every moment. As a result, our system had less information to help it choose start times. Many of these moments occur over a period of multiple minutes, and therefore detection within a wider range of times could be acceptable. In the future, we would suggest either developing heuristics for determining start times during the qualitative analysis or changing the evaluation to allow the system to select any time point during the moment, as we did for *struggling with errors*.

Our analysis of the system's incorrect decisions also revealed that particular metrics were difficult to encode. For example, our system was not always able to determine when a student had resolved a particular error, which is crucial to detecting the *struggling with errors* moment. This can be quite complex, as students exhibit a wide variety of behaviors when debugging. Another challenge we encountered is that our system does not always have enough information to determine the student's purpose for using resources when it knows a *using resources* moment occurred, resulting in a lower recall for *using resources to research an approach*. Even though our metrics generally provided enough guidance for the researcher, without human intuition or contextual understanding, the system was less accurate in interpreting the variety of ways that students use resources. With more development time, we could increase the accuracy of detection for both of these moments, but it would require significant effort to fully model all potential behaviors. While it is likely not possible to fully capture the variance in student behavior in our models, our relatively high detection accuracy and our concrete ideas for improvement show that this is a viable approach.

5 Conclusions

In this paper, we present a new approach for designing systems that detect student perceptions of the programming process, called *retrospective-enabled perception recognition*. We apply this approach to develop an expert system to detect programming moments that prompt students to negatively self-assess, building on expertise gained through retrospective interviews with 41 CS2 students. We evaluated our system with programming session data collected from an additional 33 CS2 students, finding that our system achieve F1 scores ranging from 66% to 98% for the eight self-assessment moments.

While we are encouraged by our system's performance, this work has a number of limitations. First, our evaluation relies on researcher-assigned labels. While we verified the labeling process through a formal qualitative analysis, researcher labels may not perfectly represent student perceptions. Additionally, while we believe the *retrospective-enabled perception recognition* approach can be applied to other problems, more research is needed to understand how our expert system generalizes. We developed and tested our system with students from just one course and university, and our observations of student programming sessions occurred in a lab setting. Furthermore, students worked on a limited set of problems in one programming language. As a result, additional work is needed to understand whether our system will generalize to a more naturalistic setting, more diverse problems, and other programming languages.

While our results are promising, our models could likely be improved with additional techniques for interpreting interaction log data. For example, natural language processing could help our system understand the semantics of comments and web-page content, which the researchers used when labelling the moments. Additionally, the success of this expert system suggests that this problem may be a good fit for machine teaching, an approach that empowers experts to guide machines in learning to make decisions [50]. Future work should explore whether the knowledge of student perceptions derived from the retrospective interviews can inform a machine teaching algorithm and increase our ability to detect student perceptions accurately.

Through *retrospective-enabled perception recognition*, we contribute a new approach for combining qualitative methods and expert system design to detect learning moments that students perceive as meaningful, which could generalize to a number of problems and contexts. Furthermore, our system for detecting negative self-assessment moments has the potential to enable new studies and interventions that were not previously possible. In future work, we hope to use this system to study the relationship between student behaviors, perceptions of the programming process, and self-assessments. We also hope to develop real-time feedback interventions to help students re-frame self-assessment moments and improve self-efficacy.

6 Acknowledgements

This work was supported by NSF Grant IIS-1755628. Thank you to Delta Lab.

References

1. Ahadi, A., Lister, R., Haapala, H., Vihavainen, A.: Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance. In: Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15. pp. 121–130. ACM Press, Omaha, Nebraska, USA (2015). https://doi.org/10.1145/2787622.2787717, http://dl.acm.org/citation.cfm?doid=2787622.2787717
2. Anderson, J.R., Conrad, F.G., Corbett, A.T.: Skill acquisition and the LISP tutor. Cognitive Science **13**(4), 467–505 (1989), publisher: Elsevier
3. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: Lessons learned. The journal of the learning sciences **4**(2), 167–207 (1995)
4. Bandura, A.: Self-efficacy mechanism in human agency. American psychologist **37**(2), 122 (1982)
5. Bandura, A.: Self-efficacy: The exercise of control. Macmillan (1997)
6. Bandura, A.: Self-efficacy. The Corsini encyclopedia of psychology pp. 1–3 (2010), publisher: Wiley Online Library
7. Berland, M., Martin, T., Benton, T., Petrick Smith, C., Davis, D.: Using learning analytics to understand the learning pathways of novice programmers. Journal of the Learning Sciences **22**(4), 564–599 (2013)
8. Blikstein, P.: Using learning analytics to assess students' behavior in open-ended programming tasks. In: Proceedings of the 1st international conference on learning analytics and knowledge. pp. 110–116. ACM (2011), http://dl.acm.org/citation.cfm?id=2090132
9. Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., Koller, D.: Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. Journal of the Learning Sciences **23**(4), 561–599 (2014)
10. Corbett, A.: Cognitive computer tutors: Solving the two-sigma problem. In: International Conference on User Modeling. pp. 137–147. Springer (2001)
11. Corbett, A.T., Anderson, J.R.: Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 245–252. ACM (2001)
12. Cross, J.H., Hendrix, D., Umphress, D.A.: JGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. In: 34th Annual Frontiers in Education, 2004. FIE 2004. pp. 1466–1467. IEEE Computer Society (2004)
13. Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., Saffrey, P.: Manipulating mindset to positively influence introductory programming performance. In: Proceedings of the 41st ACM technical symposium on Computer science education. pp. 431–435. ACM (2010), http://dl.acm.org/citation.cfm?id=1734409
14. Edwards, S., Li, Z.: Towards progress indicators for measuring student programming effort during solution development. In: Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling '16. pp. 31–40. ACM Press, Koli, Finland (2016). https://doi.org/10.1145/2999541.2999561, http://dl.acm.org/citation.cfm?doid=2999541.2999561
15. Ehrlinger, J., Dunning, D.: How chronic self-views influence (and potentially mislead) estimates of performance. Journal of personality and social psychology **84**(1), 5 (2003), publisher: American Psychological Association

16. Ericsson, K.A., Simon, H.A.: Protocol analysis: Verbal reports as Data. MIT Press, Cambridge, MA (1984)
17. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to design programs. MIT press Cambridge (2001)
18. Fisher, A., Margolis, J.: Unlocking the clubhouse: the Carnegie Mellon experience. ACM SIGCSE Bulletin **34**(2), 79–83 (2002), http://dl.acm.org/citation.cfm?id=543836
19. Flanigan, A.E., Peteranetz, M.S., Shell, D.F., Soh, L.K.: Exploring Changes in Computer Science Students' Implicit Theories of Intelligence Across the Semester. pp. 161–168. ACM Press (2015). https://doi.org/10.1145/2787622.2787722, http://dl.acm.org/citation.cfm?doid=2787622.2787722
20. Fuchs, M., Heckner, M., Raab, F., Wolff, C.: Monitoring students' mobile app coding behavior data analysis based on IDE and browser interaction logs. In: 2014 IEEE Global Engineering Education Conference (EDUCON). pp. 892–899. IEEE, Istanbul (Apr 2014). https://doi.org/10.1109/EDUCON.2014.6826202, http://ieeexplore.ieee.org/document/6826202/
21. Gorson, J., O'Rourke, E.: How Do Students Talk About Intelligence? An Investigation of Motivation, Self-efficacy, and Mindsets in Computer Science. In: Proceedings of the 2019 ACM Conference on International Computing Education Research - ICER '19. pp. 21–29. ACM Press, Toronto ON, Canada (2019). https://doi.org/10.1145/3291279.3339413, http://dl.acm.org/citation.cfm?doid=3291279.3339413
22. Gorson, J., O'Rourke, E.: Why do CS1 Students Think They're Bad at Programming? Investigating Self-efficacy and Self-assessments at Three Universities. In: Proceedings of the 2020 ACM Conference on International Computing Education Research. pp. 170–181 (2020)
23. Jadud, M.C.: Methods and tools for exploring novice compilation behaviour. p. 73. ACM Press (2006). https://doi.org/10.1145/1151588.1151600, http://portal.acm.org/citation.cfm?doid=1151588.1151600
24. Kinnunen, P., Simon, B.: Experiencing programming assignments in CS1: the emotional toll. In: Proceedings of the Sixth international workshop on Computing education research. pp. 77–86. ACM (2010)
25. Kinnunen, P., Simon, B.: CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. In: Proceedings of the seventh international workshop on Computing education research. pp. 19–26. ACM (2011)
26. Kinnunen, P., Simon, B.: My program is ok – am I? Computing freshmen's experiences of doing programming assignments. Computer Science Education **22**(1), 1–28 (Mar 2012). https://doi.org/10.1080/08993408.2012.655091, http://www.tandfonline.com/doi/abs/10.1080/08993408.2012.655091
27. Köksal, M.F., Başar, R., Üsküdarlı, S.: Screen-Replay: A Session Recording and Analysis Tool for DrScheme. In: Proceedings of the Scheme and Functional Programming Workshop, Technical Report, California Polytechnic State University, CPSLO-CSC-09. vol. 3, pp. 103–110. Citeseer (2009)
28. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Proceeding of the 28th international conference on Software engineering - ICSE '06. p. 492. ACM Press, Shanghai, China (2006). https://doi.org/10.1145/1134285.1134355, http://portal.acm.org/citation.cfm?doid=1134285.1134355
29. Lewis, C., Bruno, P., Raygoza, J., Wang, J.: Alignment of Goals and Perceptions of Computing Predicts Students' Sense of Belonging in Computing. In: Proceedings of the 2019 ACM Conference on International

Computing Education Research - ICER '19. pp. 11–19. ACM Press, Toronto ON, Canada (2019). https://doi.org/10.1145/3291279.3339426, http://dl.acm.org/citation.cfm?doid=3291279.3339426

30. Lewis, C.M., Anderson, R.E., Yasuhara, K.: "I Don't Code All Day": Fitting in Computer Science When the Stereotypes Don't Fit. pp. 23–32. ACM Press (2016). https://doi.org/10.1145/2960310.2960332, http://dl.acm.org/citation.cfm?doid=2960310.2960332
31. Lewis, C.M., Yasuhara, K., Anderson, R.E.: Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In: Proceedings of the seventh international workshop on Computing education research. pp. 3–10. ACM (2011)
32. Lishinski, A., Yadav, A., Enbody, R.: Students' Emotional Reactions to Programming Projects in Introduction to Programming: Measurement Approach and Influence on Learning Outcomes. In: Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17. pp. 30–38. ACM Press, Tacoma, Washington, USA (2017). https://doi.org/10.1145/3105726.3106187, http://dl.acm.org/citation.cfm?doid=3105726.3106187
33. Lishinski, A., Yadav, A., Good, J., Enbody, R.: Learning to Program: Gender Differences and Interactive Effects of Students' Motivation, Goals, and Self-Efficacy on Performance. In: In Proceedings of the 2016 ACM Conference on International Computing Education Research. pp. 211–220. ACM Press (2016). https://doi.org/10.1145/2960310.2960329, http://dl.acm.org/citation.cfm?doid=2960310.2960329
34. Loksa, D., Ko, A.J., Jernigan, W., Oleson, A., Mendez, C.J., Burnett, M.M.: Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems. pp. 1449–1461. ACM Press (2016). https://doi.org/10.1145/2858036.2858252, http://dl.acm.org/citation.cfm?doid=2858036.2858252
35. Marwan, S., Gao, G., Fisk, S., Price, T.W., Barnes, T.: Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. In: Proceedings of the 2020 ACM Conference on International Computing Education Research. pp. 194–203. ACM, Virtual Event New Zealand (Aug 2020). https://doi.org/10.1145/3372782.3406264, https://dl.acm.org/doi/10.1145/3372782.3406264
36. Master, A., Cheryan, S., Meltzoff, A.N.: Computing whether she belongs: Stereotypes undermine girls' interest and sense of belonging in computer science. Journal of educational psychology **108**(3), 424 (2016), publisher: American Psychological Association
37. Miura, I.T.: The relationship of computer self-efficacy expectations to computer interest and course enrollment in college. Sex Roles **16**(5-6), 303–311 (Mar 1987). https://doi.org/10.1007/BF00289956, http://link.springer.com/10.1007/BF00289956
38. Munson, J.P., Zitovsky, J.P.: Models for early identification of struggling novice programmers. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. pp. 699–704 (2018)
39. O'Rourke, E., Haimovitz, K., Ballweber, C., Dweck, C., Popović, Z.: Brain points: a growth mindset incentive structure boosts persistence in an educational game. In: Proceedings of the SIGCHI conference on human factors in computing systems. pp. 3339–3348. ACM (2014)

40. O'Reilly, M., Parker, N.: 'Unsatisfactory Saturation': a critical exploration of the notion of saturated sample sizes in qualitative research. Qualitative Research **13**(2), 190–197 (Apr 2013). https://doi.org/10.1177/1468794112446106, http://journals.sagepub.com/doi/10.1177/1468794112446106
41. Perscheid, M., Siegmund, B., Taeumel, M., Hirschfeld, R.: Studying the advancement in debugging practice of professional software developers. Software Quality Journal **25**(1), 83–110 (Mar 2017). https://doi.org/10.1007/s11219-015-9294-2, http://link.springer.com/10.1007/s11219-015-9294-2
42. Piech, C., Sahami, M., Koller, D., Cooper, S., Blikstein, P.: Modeling how students learn to program. In: Proceedings of the 43rd ACM technical symposium on Computer Science Education. pp. 153–160. ACM (2012)
43. Ramalingam, V., LaBelle, D., Wiedenbeck, S.: Self-efficacy and mental models in learning to program. In: ACM SIGCSE Bulletin. vol. 36, pp. 171–175. ACM (2004)
44. Reiser, B.J., Anderson, J.R., Farrell, R.G.: Dynamic Student Modelling in an Intelligent Tutor for LISP Programming. In: IJCAI. vol. 85, pp. 8–14 (1985)
45. Relich, J.D., Debus, R.L., Walker, R.: The mediating role of attribution and self-efficacy variables for treatment effects on achievement outcomes. Contemporary Educational Psychology **11**(3), 195–216 (1986), publisher: Elsevier
46. Saunders, B., Sim, J., Kingstone, T., Baker, S., Waterfield, J., Bartlam, B., Burroughs, H., Jinks, C.: Saturation in qualitative research: exploring its conceptualization and operationalization. Quality & Quantity **52**(4), 1893–1907 (Jul 2018). https://doi.org/10.1007/s11135-017-0574-8, http://link.springer.com/10.1007/s11135-017-0574-8
47. Schunk, D.H.: Self-efficacy, motivation, and performance. Journal of Applied Sport Psychology **7**(2), 112–137 (Sep 1995). https://doi.org/10.1080/10413209508406961, https://www.tandfonline.com/doi/full/10.1080/10413209508406961
48. Shapiro, J.R., Williams, A.M.: The Role of Stereotype Threats in Undermining Girls' and Women's Performance and Interest in STEM Fields. Sex Roles **66**(3-4), 175–183 (Feb 2012). https://doi.org/10.1007/s11199-011-0051-0, http://link.springer.com/10.1007/s11199-011-0051-0
49. Shenton, A.K.: Strategies for ensuring trustworthiness in qualitative research projects. Education for Information **22**(2), 63–75 (Jul 2004). https://doi.org/10.3233/EFI-2004-22201, https://www.medra.org/servlet/aliasResolver?alias=iospressdoi=10.3233/EFI-2004-22201
50. Simard, P.Y., Amershi, S., Chickering, D.M., Pelton, A.E., Ghorashi, S., Meek, C., Ramos, G., Suh, J., Verwey, J., Wang, M., others: Machine teaching: A new paradigm for building machine learning systems. arXiv preprint arXiv:1707.06742 (2017)
51. Soloway, E., Bonar, J., Ehrlich, K.: Cognitive strategies and looping constructs: an empirical study. Communications of the ACM **26**(11), 853–860 (Nov 1983). https://doi.org/10.1145/182.358436, https://dl.acm.org/doi/10.1145/182.358436
52. Sonnentag, S.: Expertise in professional software design: A process study. Journal of applied psychology **83**(5), 703 (1998), publisher: American Psychological Association
53. Steele, C.M., Aronson, J.: Stereotype threat and the intellectual test performance of African Americans. Journal of personality and social psychology **69**(5), 797 (1995)

54. Veilleux, N., Bates, R., Allendoerfer, C., Jones, D., Crawford, J., Floyd Smith, T.: The relationship between belonging and ability in computer science. In: Proceeding of the 44th ACM technical symposium on Computer science education. pp. 65–70. ACM (2013)
55. Watson, C., Li, F.W., Godwin, J.L.: No tests required: comparing traditional and dynamic predictors of programming success. In: Proceedings of the 45th ACM technical symposium on Computer science education. pp. 469–474 (2014)
56. Zadeh, L.A.: Fuzzy logic. Computer **21**(4), 83–93 (1988), publisher: IEEE