Exploring the Coding Behaviour of Successful Students in Programming by Employing Neo-Piagetian Theory

Natalie Culligan
Department of Computer Science
Maynooth University
natalie.culligan@mu.ie

Kevin Casey
Department of Computer Science
Maynooth University
kevin.casey@mu.ie

Abstract

We have collected data from approximately 300 students in their third-level first year Introduction to Programming module as they learn to write code using our in-house pedagogical coding environment, MULE. This data includes performance in lab exams and pseudocode questions, and data on code compiled, code run, and code evaluated, which we call CRE data. Evaluations are automatically graded and feedback is provided to students on their code. The student can only evaluate their code in the scheduled lab place and times but can evaluate as many times as they wish without penalty. The pseudocode questions are used to examine the students' understanding of programming concepts, by removing the use of the compiler and comparing their performance in pseudocode questions to CRE data. Using a Neo-Piagetian framework, we examine pseudocode performance, lab exam performance and programmer behaviour in terms of CRE data. We investigate CRE data as signs of a student's progression through the three stages of Piagetian understanding and build a series of Deep Neural Net binary classifiers to test if this passively collected behavioural data can be used to detect students in danger of failing.

1. Introduction

Computer Science has one of the highest failure and dropout rates in 3rd level education (Bennedsen, & Caspersen, Corney *et al.* 2010, Lang *et al.*, Watson & Li). In this paper, we will investigate if students in introductory computer science courses are failing to reach the later stages of Neo-Piagetian understanding, and if we can investigate and observe signs of these stages through passive data collection, and the results of pseudocode tasks in the weekly practical coding labs. The research question for this study is:

- Can we observe signs of progression through the Neo-Piagetian stages of learning by examining passively collected data on students' coding behaviour?

The coding behaviour data we discuss in this paper is the order in which students compile, run, and evaluate their code. Evaluation provides the student with automatic grades and feedback. The students use the pedagogical coding system MULE to complete their weekly coding tasks. In this system, students are unable to run their code until they have successfully compiled and cannot evaluate their code until it has run successfully.

In the doctoral thesis "*Neo-Piagetian Theory and the Novice Programmer*" (Teague), the author states that "*Programming competence requires abstract reasoning skills and learning to program is about the sequential and cumulative development of those abstract reasoning skills in an unfamiliar domain*." We wanted to introduce pseudocode questions into our first-year curriculum to encourage students to build mental models of programming concepts by requiring students to predict code output, without relying on the compiler. These pseudocode questions are English language representations of code that cannot be run with a compiler but represent programming concepts such as loops and arrays (Lopez *et al.*). With pseudocode, we can see if the students can abstract the concepts away from Java and apply what they have learned in class in a much more generalised way. This is useful as if the students are able to do so, they are more likely to be able to reuse the skills and apply them in a variety of ways, instead of memorizing and replicating techniques they have used in the past.

In this paper, we will discuss our findings when investigating CRE data in weekly labs as students graduate from random/loosely guided "tinkering" to more intentional code-writing. While previous work has discussed "tinkering" as a viable method of learning programming, we will discuss if this is true throughout the first semester, or if CRE data that implies an over-use of tinkering is in fact an indication that a student is not developing a good mental model of fundamental programming concepts and is therefore in danger of falling behind.

2. Related Research

2.1. Student Behaviour when Learning to Code

There have been numerous studies that investigate novice programmer behaviour such as patterns of compilation and running of code and how it relates to student success.

Perkins *et al*., investigate the different strategies that novice programmers adopt when learning to code, and describe what they term "stoppers", "movers", and "extreme movers". "Stoppers" are novices who, when faced with a problem without a clear course of action, stop attempting to find a solution to the problem and appear to be unwilling to explore the problem any further. "Movers" are novices who will constantly modify and test their code when faced with a problem. "Extreme Movers" will also constantly modify and test their code but are different from movers in that they do not seem to learn from attempts that previously did not work, and they do not continue to work on solutions that fail the first time so do not end up "homing in" on a working solution. The authors do not specifically speak about how these different patterns relate to compilation and run behaviour, but the below papers do touch on it in direct reference to this study.

Two papers on the programming environment BlueJ (Jadud, 2005, Jadud 2006) discuss the behaviours of the authors' students, and how similar their students' behaviours are to those in the above Perkins et. al. paper. They discuss their own "extreme movers", which they describe as "tinkerers", and how these students would sometimes allow their experimental code to accumulate, causing their code to become increasingly complex and, eventually, incomprehensible. The BlueJ studies found that 24% of all compilation events followed less than 10 seconds after a previous compilation, and half of all compilation events occurred less than 40 seconds after a previous compilation. Students spent more time working on their code after a successful compilation than they did trying to fix a syntax error. The authors found that students tend to program in large blocks, then spend time writing and compiling code in small bursts in order to fix syntax errors. Accordingly, multiple compilations may indicate a large number of syntactic problems.

In "*Studying the Novice Programmer*" (Soloway & Spohrer) the authors discuss the need for students to build plans. As mentioned above, students who tinker aimlessly create bugs, and without clear goals may fail to progress towards a working solution. The authors used natural language to investigate if students with plans, broken into small tasks, are more successful when programming.

In "*Analysis of Code Source Snapshot Granularity Levels"* (Vihavainen) the author discusses the ratio of "snapshots to submissions", where a snapshot is a copy of the code taken every time the student saves, compiles, runs, or tests their code. Submissions are final versions of a program submitted for correction/grading, provided by a plugin for NetBeans that provides feedback and grading to the student. Using a Wilcoxon rank sum test, the authors found a statistically significant difference between the number of runs and tests for students with previous programming experience and those without. This difference continued to be visible throughout the course, although the behaviour of the participants was more alike in the final weeks of the course, perhaps implying that these behaviours are indicators of programming proficiency.

One of the research questions in the paper "*Evaluating Neural Networks as a Method for Identifying Students in Need of Assistance*" (Castro-Wunsch) is "*Are neural network (NN) models appropriate for the task of identifying students in need of assistance?"* The authors found that, yes, neural networks predicted at-risk students at least as well as Bayesian and decision tree models, and had the advantage of being "pessimistic", meaning that the neural networks were more likely to incorrectly classify students as at-risk, rather than incorrectly classify students as not at-risk. From this research, we decided to use neural networks as our classifier.

2.2. Neo-Piagetian Theory and Abstraction in Programming

There are also a number of studies that use Neo-Piagetian theory in examining student behaviour in computer science and discuss abstraction in relation to novice and expert programmers.

In "*Concrete and Other Neo-Piagetian forms of Reasoning in the Novice Programmer"*, (Lister) the author discusses the reasoning behind the use of Neo-Piagetian theory. Classical Piagetian theory considers the progress through different stages of learning to be a consequence of a biological maturing of the brain. Neo-Piagetian theory, on the other hand, considers this instead a result of gaining experience, and in particular, the ability to "chunk" knowledge within a certain knowledge domain.

Corney et. al (2011) describe a study in which almost half of the sample students were unable to answer a simple explain-in-plain-English question in the third week of their introductory programming course, showing that students were encountering problems much sooner than could be detected by traditional programming questions/examinations.
In "*Neo-Piagetian Theory and the Novice Programmer"* (Teague, 2015), the author found that the development of programming skills is both "*sequential and cumulative*", and that behaviours associated with sensorimotor and preoperational reasoning are evident from very early in the semester.

The authors of "*Mired in the Web: Vignettes from Charlotte and Other Novice Programmers"* (Teague *et al*.) ask if a student can have different levels of ability for different tasks which test similar programming concepts – if a student can trace and understand code, can they also *write* that code? They also ask why some students do not seem to be able to understand code with abstractions and instead rely on tracing code with specific values. The study found that students who were still operating at the sensorimotor level in week 2 were often still operating the same way in week 5, and were lagging behind students who were operating at the preoperational level in week 2. They defined students in the preoperational stages by certain behaviours which they observed using think-aloud data from students. Preoperational behaviours were guessing, a fragile grasp of semantics, confused use of nomenclature, an inability to trace simple code, as well as general misconceptions. Errors due to cognitive overload and reluctance to trace were considered behaviours associated with both sensorimotor and preoperational. The ability to trace but not explain code, as well as a reliance on specific values, were signs of the preoperational stage. The authors note that students may achieve marks for guessed answers, but it is not until they listen to the students speak aloud their thought process that they were able to get a clear picture of the students understanding and ability.

Shneierman and Mayer found that expert programmers were able to recall more of a program than novices when it was presented to them in normal order, but not when it was scrambled, implying that the experts were able to "chunk" information together when the code made sense. The authors proposed that experienced programmers construct functional representations of computer programs.

Adelson found that expert programmers' memory chunks tended to be semantically or functionally related, while novices typically chunked by syntax. Semantic knowledge consists of programming concepts that are generalized, and independent of programming language, whereas syntactic knowledge is more precise and rooted in exact representations of concepts in specific programming languages. For example, a novice may think of a loop as a specific for loop in Java, but an expert planning a piece of code may simply think of a loop abstractly, as something that performs a needed function, without thinking about the exact type of loop, the details of the iteration, or the syntax associated with it (Bisant & Groninger, Wiedenbeck).

3. Methodology

For this study, we collected data from around 300 students as they completed their introduction to programming module in Java using MULE, our in-house, browser-based pedagogical coding environment (Culligan & Casey). This system resembles a desktop with both built-in applications for content and assignment delivery, and a code editor for completing, running, and evaluating code for assignments. MULE also includes mechanisms for making sections of the material invisible to some users until some constraints are satisfied such as date/time and IP address – this was used to allow certain assignments to only be accessible in the scheduled lab times and locations. Within MULE, each attempt the student makes on an assignment is recorded, and the student can easily recover any previous attempt, allowing the student to "tinker" and experiment with their code without fear of losing any work. There is evidence to suggest that a certain amount playing/tinkering with code is an indication of student success (Berland *et al.*, Berland & Martin).

For 5 of the 10 mandatory computer lab sessions during the first semester of their computer science course, students were asked to predict the outcome of pseudocode snippits, along with their usual lab consisting of two programming questions, and some peer-programming tasks. The students were told that they are not awarded any marks towards their continuous assessment for answering the pseudocode questions. The students have access to most of the programming tasks before the lab and can write code, compile, and run it, but not evaluate it for continuous assessment grades. Some of the exercises are only accessible in the labs at the assigned times, so students must write, run, compile, and evaluate the code in the lab.

Although students were able to work on an assignment before assigned lab times, we chose to look exclusively at the data from lab times. Our reasoning is that students outside of labs can be in very different environments – some may have a quiet place to work undisturbed, others may be working in a noisy environment or may be frequently interrupted, so comparisons of their behaviour may be less insightful than those from a formal lab. For most of the semester, the students can only evaluate from inside the lab during the specified lab times, so the data from outside the labs would only have compile and run events.

We did not include data from students who did not participate in the weekly labs (missing more than four), as we wanted to investigate changes in behaviour from week to week and to look at at-risk students who are actively engaging in the course labs on a weekly basis (Castro-Wunsch). After removing students who did not complete 4 or more labs, we were left with 266 subjects. The gathered data is the patterns of student compile, run and evaluate actions:

- Compile: Students cannot run their code until it compiles successfully
- Run: Students cannot evaluate their code until it runs successfully
- Evaluate: The student's code is assigned a grade, and feedback is provided.

This data was used to build Deep Neural Net binary classifiers, that would classify students as being in either the top 50% or the bottom 50% of the class lab exam grades on a week-to-week basis. Each weekly classifier would use the CRE data for each assignment for that week, and from all previous weeks. Below we discuss the results of statistical tests exploring correlations between student behaviour and outcome, and the classifier built to predict student outcome.

4. Analysis

When analysing the data, for every time a student performs a CRE action, we look at that action and the one before and record it as a "movement" - the student moves from a Compile to a Run, is recorded as C2R, or a Run to an Evaluate in R2E for example. When processing this data, we looked at each movement as a percentage of all actions a student took during that lab. From the previous studies on programming and Neo-Piagetian stages, we expected to see the following as signs of progression through the stages:

Sensorimotor Stage: Interacting almost randomly, with little understanding of the outcome, resulting in more C2C movements, less C2R movements and less participation and success with pseudocode questions.

Preoperational Reasoning Stage: The student is beginning to master writing compilable code, and can predict code outcome, resulting in higher amount of C2R movements and R2C movements, fewer C2C movements and more participation and success with pseudocode questions.

Concrete Operational Stage: At this stage, programmers have a good grasp of concepts allowing the programmer to write more complex code, resulting in fewer C2C movements, fewer C2R movements, more R2E movements and more participation and success with pseudocode questions.

The students in the study were divided into two groups: those in the top 50% of the class in lab exam grades, and those in the bottom 50%. The two data sets contain the percentages of total movements per week for each student. A sample of the student data for a week would look like the following:

C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
0.33997	0.254913	0.127186	0.01639	0.130208	0.111902	0.003655	0.004159

Table 1: Example of an average sample of student weekly data

The following tests were then run on the two data sets:

- To examine if the differences between the two groups were significant, t-tests were used.
- Linear regression was used to find which movements were most related to lab exam outcome, on a week-to-week basis, to select which movement data would be used in the classifier.
- Finally, the data from the most significant movements each week are used to create a Deep Neural Net binary classifier, to classify each student as being in the top or bottom 50% of the class.

5. Results

To find if there were significant differences between the top and bottom 50% of the students, t-tests were used, the results of which are considered significant differences between the two groups if the result is less than 0.05. These results are in bold. The p-value results of the groups according to lab exam results are in Table 2, and the results of the groups divided by pseudocode performance are in Table 3. Lab 6 and lab 10 included lab exams, during which the students could not look at their previously written code from earlier labs.

	C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
1	**0.001214**	0.470967	0.539369	0.448587	0.090213	0.070492	0.650004	0.24305
2	**0.004757**	**0.02424**	0.665045	0.674198	**0.005787**	0.060203	0.388143	0.260525
3	**4.80E-06**	**5.04E-05**	0.112107	0.585846	**0.031448**	0.498212	0.120838	0.280715
4	**1.50E-06**	**3.16E-08**	**4.04E-06**	**0.032048**	0.305453	0.031562	0.951828	0.748698
5	**4.03E-10**	**1.47E-08**	**0.008756**	0.100132	**0.000341**	**0.004397**	**0.015933**	0.10881
6	**9.00E-14**	**7.94E-15**	**5.90E-11**	**0.019153**	0.064127	0.122571	0.026066	0.940655
7	**2.05E-06**	**6.28E-09**	**6.96E-07**	0.561189	0.111728	0.140013	0.924634	0.579504
8	**5.73E-13**	**2.60E-09**	**0.00853**	**0.00715**	**3.36E-05**	**2.50E-05**	0.501948	**0.04282**
9	**0.002878**	0.187736	0.187655	0.386204	**0.008422**	**0.0083**	0.234538	0.389353
10	**6.05E-06**	**4.70E-07**	**0.012479**	0.683429	**0.034407**	0.082467	0.609155	0.758995

Table 2: Results of t-test on groups divided by lab exam results

There were significant differences between the two groups found in C2C every week, C2R most weeks, and R2E and R2C in 8 of the 10 weeks. In Table 3, we see that the results are similar results to the lab exam t-tests, the main difference being that the R2E movements are almost never significant.

	C2C	C2R	R2C	R2R	R2E	E2C	E2R	E2E
1	0.656744	**0.010296**	0.167739	0.096637	0.368741	0.27848	0.979841	0.062224
2	**0.007839**	**0.01318**	0.698311	0.79891	**0.005272**	0.518315	0.074192	0.384717
3	**0.001236**	**0.000551**	**0.049986**	0.141391	0.498473	0.615511	0.872643	0.817136
4	**0.00078**	**0.000378**	**0.015029**	0.089415	0.768301	0.095344	0.72989	0.085109
5	**0.000306**	**0.001537**	0.268965	**0.043827**	0.051354	0.050773	0.147545	0.185538
6	**0.000562**	**0.001477**	**0.038343**	**0.018253**	0.07908	0.088534	0.328674	0.182885
7	**0.014912**	**0.00068**	**0.001243**	0.371309	0.114922	0.140932	0.328346	0.52572
8	**0.014178**	0.279716	0.265897	0.414656	0.768084	0.718394	0.434119	0.501809
9	**0.043973**	0.184825	0.768122	0.165121	0.268245	0.31005	0.380506	0.77323
10	**0.027376**	**0.009383**	**0.042855**	0.350375	0.601622	0.888391	0.139526	0.910241

Table 3: Results of t-test on groups divided by pseudocode results

From our predicted behaviour of the Neo-Piagetian stages outlined at the start of the analysis section we expected to see students who did poorly in the exams displaying different behaviour in the C2C, C2R and R2C movements as more successful students moved onto preoperational reasoning stages. Higher achieving students have a consistently lower average percentage of C2C when groups are divided by lab exam results. The difference in C2C movements gets steadily larger from week 1 until week 7, when it slightly reduces. This is also true for the pseudocode results, with smaller margins of difference. The difference is smaller in the last weeks of the module, which may indicate that our students who do not do well are moving through the Neo-Piagetian stages but are not moving quickly enough for the course.

Higher achieving students have a consistently higher average percentage of C2R when divided by lab exam results. This difference peaks in week 7, for both lab exam and pseudocode results. Both groups have a similar percentage of R2C when divided by lab exam results, but the difference peaks in weeks 6 and 7, when the higher achieving students have a higher average percentage of R2C movements. This may be the point where successful students have reached preoperational reasoning, as an increase in R2C movements indicate the student is in the "tinkering" stage as described by Perkins *et al*.

Using the dataset containing the CRE percentages for each student for each week, Deep Neural Net binary classifiers were trained to classify students as being in the top 50% or the bottom 50% of grades for the lab exams. Linear Regression tests were used to compare the CRE actions and their relation to student performance in lab exams. This was used to select movement data to be used in the Deep Neural Nets. Multicollinearity can be an issue for DNN, so a check was run on the features (where each movement was a feature) and we removed the most highly correlated CRE data and tested again. This was repeated until the remaining data was sufficiently nonlinearly related, when all features had a variance inflation factor (a test for correlation between independent variables) of less than 5. The resulting data set was used to train and test our DNN classifier. A classifier was built for each week of the semester, using the CRE data from that week, and from all previous weeks. The results are shown in Table 4. The results of week 9 and 10 are identical to week 8, as it uses the same CRE data after the multicollinearity tests.

Week	**Average Classifier Success Rate**
1	0.62
2	0.6
3	0.68
4	0.7
5	0.62
6	0.6
7	0.72
8	0.76

Table 4: Classifier results

5. Discussion

Other studies have referred to lab 4/week 4 (Teague) as the time around which students who are in danger of failing begin to perform badly or separate in behaviour from the other students. Of course, what takes place at this point varies across different institutions and courses. Nonetheless we see that in line with this estimated timescale, the differing behaviour among students becomes more pronounced around lab 4, and at this point the classifier has a success of 70%. At this point, if students are consistently compiling without progressing to run, this is a sign the student is in danger. This is not hugely surprising. It implies that the student is failing to write compilable code, and we would expect that a student who cannot write compilable code would be in danger.

The percentage of R2C becomes more significant around week 4. A student who compiles code, then runs, but then goes back to compile, is most likely working on a semantic issue, rather than a syntactic one, as mentioned in the BlueJ papers (Jadud 2005, Jadud 2006). We suspect that the reason it becomes relevant to the students' overall performance in lab exam results is because week 4 is when most students should be beginning to master syntax and to abstract solutions, allowing them to construct more complex programs using multiple concepts together. The result is that we see successful students compiling successfully and rewriting their code until they reach a solution, causing successful students to have more C2R movements and fewer C2C movements. Students who are still struggling to write semantically correct code will have even more C2C movements as the assignments get more difficult.

Research Question: Can we observe signs of progression through the Neo-Piagetian stages of learning by examining passively collected data on students coding behaviour?

Yes, we have described the expected signs in CRE movements of progression through the stages of Neo-Piagetian learning, observed these signs in novice programmers and found these signs relate to student success. From our analysis section, we see that the CRE movements that are associated with success change as the semester progresses. Using a Neo-Piagetian framework, we examine these differences.

The three Neo-Piagetian stages in learning to program (Lister, du Boulay, Teague, Teague *et al.*):

(1) Sensorimotor Stage - interacting almost randomly, with little understanding of the outcome

> A high percentage of C2C movements may indicate that a student is tinkering almost randomly with their code and is unable to write compilable code. From our analysis, we see that a lower amount of C2C movements, and a higher amount of C2R movements is associated with better performance in lab exams. This is similar to the findings in the paper (Vihavainen) which found a statistically significant difference between the number of runs and tests for students with previous programming experience and those without.

(2) Preoperational Stage – beginning to master syntax, deeper understanding and being able to predict behaviour from interactions

> Students with a higher amount of C2R movements may be in this stage, as they become able to write compilable code, but are still be unable to predict the outcome of their code. As a result of this, the student will repeatedly "tinker" with their code, resulting in increased R2C movements. We found R2C movements became significant from week 4, indicating that students should reach this stage by week 4 if they are to be successful in the module lab exams.

(3) Concrete Operational Stage – can "chunk" (Shneiderman & Mayer) programming concepts and abstractions of the code's behaviour, allowing the programmer to write more complex code.

> At this point, students should be able to write compilable code and successfully predict their code's outcome. Students at this stage should have fewer C2C movements, fewer C2R movements, and a higher percentage of R2E movements. This indicates that they have a good grasp of semantics and are able to predict code behaviour with less tinkering and playing with code. We would expect to see a higher correlation between outcome and R2E movements as students reach this stage, and while R2E is related to success at some points in the semester, the average difference between the two groups is consistently low. We strongly suspect that most students do not reach concrete operational stage until after their first semester (Teague).

6. Conclusions

We have found that C2C and C2R movements are important indicators of student performance in their first semester of programming. While the highest classifier success of 76% used data from throughout the 8 weeks, we had success with the week 4 classifier which had a success percentage of 70%, showing there is evidence of a student success or failure as early as week 4. This version of the classifier used all 4 weeks C2C percentages as the input data in predicting the student outcome. In future work, it would be worth looking at which specific assignments and topics are key clues in a student's eventual outcome.

We would expect that student coding behaviour would correlate to lab exam performance and pseudocode performance, if the coding behaviour in question indicates progress through the Neo-Piagetian stages of learning. We have seen that patterns of student behaviour contain indications from an early stage if they are likely to perform well in lab exams. We have discussed how this relates to previous work done in the area of Neo-Piagetian theory in the context of students learning to program. We have established a strong case for the connection between students' programming behaviour and their stage of Neo-Piagetian learning by showing the correlation between student CRE movements, and their lab exam outcomes, and we discussed the reasons behind those behaviours and how they relate to Neo-Piagetian theory.

Introduction to programming modules that emphasize only how to write code, and grade based primarily on written code may be problematic. Results of pseudocode assignments in a programming module can help us as researchers and educators to identify students who have not developed mental models of programming concepts, and are instead relying on "hacking", where students attempt to complete a coding assignment by writing code and testing input/output without planning and predicting their code's behaviour. Students who are "hacking" may still perform reasonably well in their weekly labs, and so may believe that they are keeping up and do not need to continue to work on their grasp of fundamental coding concepts. These students will then progress to more difficult modules without the programming basics required to engage with the material. This may be a scenario unique to computer science and a significant contributary factor as to why computer science failure rates are so high. In future work, we will examine how a novice programmer's pseudocode results and patterns of behaviour may relate to code complexity, as a reflection of their ability to "chunk" programming concepts, in order to combine them to create solutions for programming problems.

In conclusion, the most significant findings from this study are, firstly, that the divergence in behaviour between high and low achieving students takes place in week 4. Students who are not displaying signs of progression to the preoperational stage of Neo-Piagetian learning do not do well in their lab exams at the end of the semester. Secondly, we found that these differences in behaviour are less pronounced later in the semester – implying that the students who were behind in week 4 are capable of progression to preoperational stage, but crucially, not at the pace dictated by the module.

7. References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. Memory & cognition, 9(4), 422-433.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. AcM SIGcSE Bulletin, 39(2), 32-36.J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming.ACM SIGCSE Bulletin,39(2):32–36, 2007.

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. Journal of the Learning Sciences, 22(4), 564-599.

Berland, M., & Martin, T. (2011). Clusters and patterns of novice programmers. In The meeting of the American Educational Research Association. New Orleans, LA.

Bisant, D. B., & Groninger, L. (1993). Cognitive processes in software fault detection: a review and synthesis. International Journal of Human-Computer Interaction, 5(2), 189-206.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. International Journal of man-machine studies, 14(3), 237-249.

Castro-Wunsch, K., Ahadi, A., & Petersen, A. (2017, March). Evaluating neural networks as a method for identifying students in need of assistance. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 111-116).

Corney, M. W., Lister, R., & Teague, D. M. (2011, January). Early relational reasoning and the novice programmer: Swapping as the "Hello World" of relational reasoning. In Conferences in Research and Practice in Information Technology (CRPIT) (Vol. 114, pp. 95-104). Australian Computer Society, Inc..

Corney, M. W., Teague, D. M., & Thomas, R. N. (2010, January). Engaging students in programming. In Conferences in Research and Practice in Information Technology, Vol. 103. Tony Clear and John Hamer, Eds. (Vol. 103, pp. 63-72). Australian Computer Society, Inc..

Culligan, N., & Casey, K. (2018). Building an Authentic Novice Programming Lab Environment. Irish Conference On Engaging Pedagogy

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. Computer Science Education, 15(1), 25-40.

Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ (Doctoral dissertation, University of Kent).

Lang, C., McKay, J., & Lewis, S. (2007). Seven factors that influence ICT student achievement. ACM SIGCSE Bulletin, 39(3), 221-225.

Lister, R. (2011, December). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In Conferences in Research and Practice in Information Technology Series.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008, September). Relationships between reading, tracing and writing skills in introductory programming. In Proceedings of the fourth international workshop on computing education research (pp. 101-112).

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer & Information Sciences, 8(3), 219-238.

Soloway, E., & Spohrer, J. C. (2013). Studying the novice programmer. Psychology Press.

Teague, D. (2015). Neo-Piagetian theory and the novice programmer (Doctoral dissertation, Queensland University of Technology).

Teague, D., Lister, R., & Ahadi, A. (2015, January). Mired in the Web: Vignettes from Charlotte and Other Novice Programmers. In ACE (pp. 165-174).

Vihavainen, A., Luukkainen, M., & Ihantola, P. (2014, October). Analysis of source code snapshot granularity levels. In Proceedings of the 15th Annual Conference on Information technology education (pp. 21-26).

Watson, C., & Li, F. W. (2014, June). Failure rates in introductory programming revisited. In Proceedings of the 2014 conference on Innovation & technology in computer science education (pp. 39-44).

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. International Journal of Man-Machine Studies, 23(4), 383-390.