Analyzing coding behaviour of novice programmers in different instructional settings: Creating vs. Debugging

1st Tom Neutens
Ghent University
IDLab, Department of Information Technology, Ghent University
Ghent, Belgium
Tom.Neutens@UGent.be

2nd Francis wyffels
Ghent University
IDLab, Department of Information Technology, Ghent University
Ghent, Belgium
Francis.wyffels@UGent.be

***Abstract*—Many methods exist for teaching programming in a physical computing context. However, it is not clear what the advantages and disadvantages of these methods are in practice. Our research compares two methods for integrating programming in a primary robotics workshop. Both methods use a custom-designed programming environment based on Google Blockly. Moreover, one method lets learners create their programs from scratch while the second method requires the learners to fix faulty programs. We compared the differences between the integration methods by analyzing programming environment logging data and linked it to the results on a programming knowledge test. Our results show that the learners in the create group spend more time solving the programming assignment and are more often distracted by code blocks they don't need. The learners in the debug group require less time. However, they perform a disproportionate amount of code changes and apply the "tinkering" strategy more often. Nevertheless, the drawbacks of the create method, the learners in this group did show significantly higher scores on our programming knowledge test.**

***Index Terms*—k-12, graphical programming, debugging, code analysis, physical computing**

I. Introduction

Primary school robotics covers a wide range of interrelated topics. One of these topics is computer programming. Integrating programming into a robotics course can be done in a variety of ways. This paper investigates how two different integration methods affect learning by analyzing how learners interact with the coding environment.

Similar to language learning, learning programming is split into reading and writing skills. Consequently, many proposed teaching techniques focus either on reading, writing, or a combination of both. Many introductory programming courses mostly expect learners to write code, also known as code generation [1]. This teaching technique is often chosen since it is similar to what programmers do in real life. Mimicking real life should provide an authentic learning environment [2]. However, multiple studies have shown that this method has drawbacks. In [1], they have shown that students often had difficulties in finding a solution to the problems presented to them when having to generate code. Additionally, they showed that learners taught using the code generation method scored lower on a statements knowledge test than students who were taught using code completion problems. Moreover, in [3], it was shown that teaching programming using the generation method required more time and triggered more questions from the learners than the part-complete solution method they describe. The part-complete solution method provides the learners with an incomplete program and requires them to add an element. One specific code completion strategy often used when teaching programming is making learners debug faulty programs [4]. Finding errors in your program and correcting them is an essential skill programmers have to learn. The authors of [5] have shown that teaching programming through debugging can lead to more success when learners create their own programs and possibly helps sustain learners' motivation. Additionally, In [6], they argue that acquiring debugging skills increases a programmer's confidence. Moreover, studies have shown that teaching programming using this technique has a positive effect on problem-solving skills in general and facilitates the transfer of these skills to other domains [7]. However, others have shown that teaching programming through debugging can result in learners applying the "tinkering " strategy for solving the problems [8].

II. Research questions

Our main research objective is to try and confirm the advantages and drawbacks of each of the methods in a realistic classroom setting using logging data from our programming environment and the results on a programming test. For clarity, we limited the analysis in this paper to four different dimensions for which we formulated specific research questions:

- Do the learners in the create group require more time to get to a solution than the ones in the debug group?
- Are the learners in the create group more distracted by the open problem context than the ones in the debug group?
- Do the learners in the debug group show more tinkering behavior than the ones in the create group?
- Do the learners in the create group score lower on a programming test than the ones in the debug group?

III. Method

To compare the integration of programming using either the create or debug methods, we created two identical robotics workshops for which only the programming assignments were changed. We took the educational context in which teachers in the region of Flanders (Belgium) work today as a reference for our experiment. Using this context as a reference, we identified constraints enforced by both practice and policy that the instructional design had to meet. The first major constraint in primary school is time. A governmental policy defines ten topics primary schools have to teach [9]. From these ten topics, one focuses on science. Within the science topic, only a limited number of educational goals focus on technical systems and processes. Consequently, in practice, teachers look for content that covers a wide range of educational goals within a limited time. The second constraint teachers often face is their limited set of resources. Most of the budget provided by the government is used to pay teacher salaries [10]. This leaves little resources to invest in teaching materials, especially for subjects that represent only a small part of the curriculum. The third and final constraint we identified is teacher content knowledge. Primary school teachers have minimal knowledge about more complex STEM topics since they were not educated to

teach them. Consequently, teachers often depend on external partners to help them bring more complex content into the classroom. We took these constraints into account when creating our instructional design.

A. Experimental group

Since STEM education is getting a more prevalent role in primary education in Flanders, our experiment focuses on the last two years of primary school. The learners who participated in our experiment were between the ages of ten and twelve. They were randomly selected from 10 different primary schools in Flanders. Since we selected a random sample from Flemish primary education, our experimental group reflects the underlying cultural and gender distributions present in primary education. Our results show our final group had 53% girls and 46% boys 1% preferred not to identify their gender. About 70% of participants indicated speaking mostly Flemish at home while about 20% sporadically or never speaks Flemish at home. These distributions match the ones for all of the primary schools provided by the Flemish government [1].

B. Instructional design

We used the constraints described above to create an authentic learning path containing multiple different learning experiences. The learning path consists of three workshop sessions of about 150 minutes, resulting in 7 hours and 30 minutes of learning. The workshops took place in the classroom of the students. The teachers of the classes were always present to help where possible and to help. However, the workshops themselves were given by either a researcher or a supporting teacher who was part of the research team. Using an expert teacher reflects how STEM is often integrated into primary school today. Since primary school teachers lack the necessary content knowledge, they often rely on external partners to fulfill some parts of the curriculum. When selecting content for the educational design, we took into account the governmental educational goals and aimed to match the content with a selection of these goals. After careful consideration, we created a physical computing learning path that seamlessly integrates the governmental educational goals with technical systems and processes. To test our hypotheses, we created two variations of the learning path in which we change the way the learners program the physical system. The first variation focuses on creating programs from scratch allowing learners to build their program step by step. This learning path incrementally introduces new programming concepts by explaining them using an example and asking the students to solve one or more exercises using that concept. The exercises require the learners to write a program from scratch trying to achieve the functionality we requested. The second variation uses a learning approach that focuses on changing and fixing programs. In this setup, similar to the *create* workshop, the learners first get a short explanation of a specific coding concept using an example. After each concept is introduced, the learners perform one or more exercises on that concept. In the exercises, they get a faulty or incomplete program and have to change the program to reach their goal. To keep the focus on a specific concept, the learners only have to either change one block in the program or add one block to the program. Moreover, during the first session, the learners are instructed to write down the answers to the following questions: (1) What should the program do? (2) What does the program do now? (3) At what point in the program does it go wrong? (4) How can we fix the error? These questions provide a strategy for tackling the problems facilitating the learning process. In the second and third session we did not explicitly ask the learners to answer these questions enabling them to choose the solution strategy they wanted.

Figure 1 shows an overview of the learning path with the variations labeled as *create* and *debug*. All building and programming activities were done in groups of two. We paired up the students for multiple reasons. Firstly, a robotics workshop requires a lot of hardware, including a robot kit with mechanical and electrical components as well as a laptop. Classrooms simply don't have the physical space to provide each learner with a full setup. Secondly, previous research suggests pair programming is advantageous for the learning process. In [11], the authors have shown that learning to program in pairs did not affect achievement and confidence among groups but did make girls more productive and confident. Moreover, the authors of [12] have shown that programming in pairs exposes learners to different ideas, reduced their frustrations, and helped them form social connections. Additionally, as explained in [13], programming in pairs improves pass rates and retention. Because literature shows working in pairs has mostly benefits and limited shortcomings, we found it appropriate for our experiment.

Each workshop session contains at least one hour of programming exercises. The content of these exercises is different between the *debug* and *create* workshops. A detailed list of the exercises for each track is also shown in Figure 1. In addition to the coding sessions, the workshop contains two CSUnplugged activities [14]. These familiarize the students with a computational concept which they are going to need during the exercises that follow. The first Unplugged activity familiarizes the learners with the concept of programming by making them write a program that lets their teacher make a sandwich. The second activity introduces the concept of conditional statements by making the students perform a certain act when the condition shown on the blackboard is true for them. Even though the *debug* and *create* workshops use different exercises, their objectives are the same. An overview of the learning objectives is shown in the following list.

- **Session 1**
  1) The learners know that a program is a sequence of instructions that are accurate and in the right order;
  2) The learners write, change, run, upload code;
  3) The learners know how to connect the microcontroller;
  4) The learners print text to the lcd-screen;
  5) The learners know the difference between setup and loop;
  6) The learners correctly use the wait-block;
  7) The learners use counting loops;
  8) The learners control LEDs and the buzzer;
  9) The learners convert microseconds into seconds;
- **Session 2**
  1) The learners construct a two-wheeled riding robot:
     a) The learners connect the structural parts;
     b) The learners connect the battery using a rubber band;
     c) The learners connect the motors using a screwdriver;
  2) The learners use the DC-motor block;
  3) The learners run their program inside the simulator;
  4) The learners use a wait-block to add time-based behavior;
  5) The learners make their robot drive into different shapes;
- **Session 3**
  1) The learners connect a sonar distance sensor to their robot;
  2) The learners understand the function of a sensor;
  3) The learners evaluate conditionals and perform different;
  4) The learners use an if statement with the condition to read the distance from a sonar sensor;

[1] https://onderwijs.vlaanderen.be/nl/nl/onderwijsstatistieken/statistisch-jaarboek/statistisch-jaarboek-van-het-vlaams-onderwijs-2018-2019

5) The learners program their robot to act on sensor input;
6) The learners understand the effect of the real world on the programming process;

To reach these learning goals, we created a custom set of teaching tools specifically designed for our workshop. These teaching tools include: (1) A custom-designed Arduino-based microcontroller board for robotics in education. The main features of the board include: An LCD-screen, 9 LED-lights, a buzzer, five buttons, and a built-in motor driver to control different types of motors. The board enables learners to connect and control the components of a basic robot easily. (2) A custom robotics kit, designed to be inexpensive and easy to produce in a maker lab. It includes two inexpensive DC-motors with wheels, a set of laser-cut parts to construct a frame, a sonar sensor, and a set of standard screws. (3) A tailor-made open-source graphical programming environment based on Google Blockly [2], which includes a robotic simulator and debugger. Figure 2 shows a screen-shot of the environment. It has the standard elements like the Google Blockly toolbox and workspace but extends on it in different ways. The most important extension is the simulator view. This view provides the learners with a simulated environment in which they can test their code. All the tools are open source and designed to be inexpensive so they can be used in all schools, even with a limited budget for STEM education.

Since we designed the programming environment ourselves we were able to create logs of all interactions the learners had with it. Each time a learner clicks one of the buttons or makes a change to the code, a log entry is saved to a database. All log entries are timestamped and contain metadata about the event that occurred. This metadata includes a session-id generated when the environment is started, the state of the simulation, and information about the event itself. For a code change event, this information contains the current program on the screen. For other events, this information includes an identifier of the event type.

C. Metrics

Using the logging data, we performed an analysis of how the interaction patterns differ between the create and debug workshop. To extract relevant information from our dataset, we selected indicators based on the advantages and disadvantages of both methods described in the literature. As described in the introduction, the create method gives the learners a more open environment increasing cognitive load. This can result in difficulties finding a solution, and more time needed to get to a correct solution [1]. The debug method should provide a more clearly defined goal and require less time to get to a solution [3]. However, the debug method can lead to a solution strategy called “Tinkering”, where the learners change random elements of the program and execute it to see what it does unit they find a solution. This strategy can be a valid learning experience. However, it limits the acquisition of a deeper understanding of the program [4], [8]. To detect differences in an effort to get to a solution, we analysed the time learners spent solving the programming problems during the session. In our data log, each code change is timestamped, so we can easily calculate how much time the learners needed to solve the problem. To assess if the create method impeded learning by offering too many options, we analyzed if the learners used programming blocks which are not required for solving the problem they are working on. To identify the level of ”tinkering” the learners use, we calculated the ratio between the number of code executions and the number of code edits. Frequent runs and minor edits have been used by others to identify tinkering [4].

To quantify the learning effect, the students in both groups filled out a small programming test with four programming questions. The questions were multiple-choice, showing the learners a small program and asking what it does. The questions only contained programming concepts the learners had to use during the workshop sessions. Literature suggests that teaching programming using the code generation technique can lead to lower scores on a statements knowledge test compared to using code completion strategies [1]. When working in a block-based programming environment like Scratch, the code completion strategy is similar to the debugging strategy. The completion strategy requires learners to add missing elements to the code. The debugging strategy adds to this by also requiring them to make changes to an existing code block.

IV. Results

Our workshops resulted in a dataset with 363793 entries for the create workshop and 203615 entries for the debug workshop. This data includes a lot of information that was not relevant for this paper. For example, when blocks in the workspace were moved or when programs were opened or saved. We eliminated all unneeded events from the dataset and only kept the events for when a code change occurred, or the program was run. This resulted in a final dataset with 57349 entries for the create group and 39516 for the debug group. The create dataset contains 4948 run events and 52401 code change events while the debug dataset has 5682 and 33834 run and code change events, respectively. Each event has a timestamp, which we can use to identify if it came from a debug or create workshop. Additionally, each event is tagged using a unique session number. This session number is generated each time the environment was started. Consequently, this number corresponds to one learner during a specific workshop session, identifying all interactions that a specific learner had with the environment during one workshop session. First, we compared the time learners in each group needed to solve the programming problems during the different sessions. To get an accurate image of how much time the learners needed, we converted the code edit log-items to a list of time differences between the events. Since we only want to register active programming time, we had to filter this list of time differences to exclude long intervals without code edits. During these intervals, the learners were performing some other aspect of the workshop, like building their robot or connecting sensors. We chose a threshold of two minutes for excluding time differences. If the difference between two log events is greater than two minutes, we assume there was no active programming between these events. Consequently, we exclude this time difference from our total time spent programming. After filtering these time differences, we added all time differences for each session and created a normalized distribution of the number of sessions with a specific duration. This distribution can be seen in Figure 3. In each session in the create group, an average of 1974 seconds was spent programming. In the debug group, this was only 1659 seconds. Figure 3 seems to confirm that learners in the create group spend more time programming than the ones in the debug group. Comparing the distribution of the two groups using a z-test shows a significant difference ($p < 0.05$).

The results in the paragraph above confirm that the learners in the create group spend more time solving the programming challenges. However, does the extra time spent mean that the learners were distracted by code blocks they don’t need? To check this, we analysed the number of times each type of block was added in both the create

[2]https://developers.google.com/blockly

Fig. 1: Graphical overview of the instructional design for the intervention. In the center, a high-level description of each session is given. Above and below a list of programming exercises for the *create* and *debug* group are given respectively.

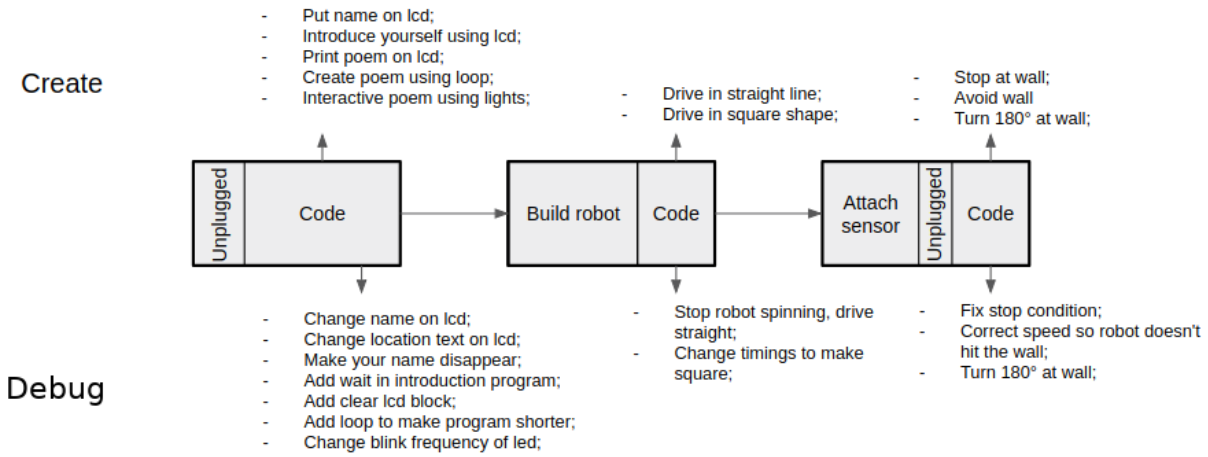
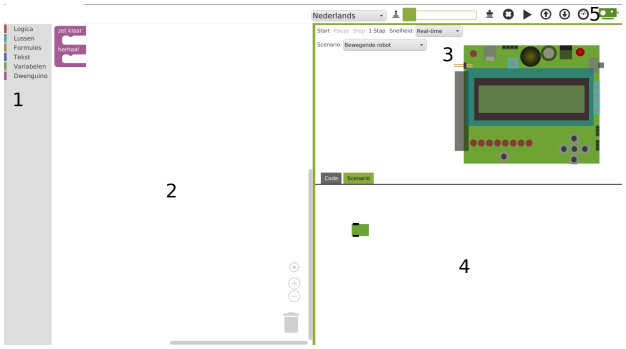


Fig. 2: Overview of the custom programming environment. (1) The toolbox repository with all the blocks that can be used. (2) The workspace area, this is where programs are constructed. (3) The microcontroller board simulation. (4) A simulation of a riding robot similar to the one they build during the sessions. (5) Controls for saving, restoring and uploading the code to the microcontroller board



and debug workshops. As shown in figure 1, the three workshop sessions have specific challenges. Consequently, only a subset of the available code-blocks is required during each session. For each workshop session, we identified the blocks the learners had to use and grouped them together into a set. We labeled this set as the *undistracting* block set. All other blocks available in the toolbox were grouped together in the *distracting* set. This grouping resulted in two sets of blocks for each workshop session (six sets in total). Most of the blocks available to the learners were included in the analysis however, some blocks were omitted since they were required in all sessions. The blocks that were omitted were terminal blocks like a number or a string. A list of the blocks that were analysed is shown in table I. For each logged session in our database, we counted how many blocks were in the respective distracting and undistracting sets. By calculating the ratio between the number of items in the distracted set and the sum of the items in the undistracting and distracting sets we get a metric for how distracted the learners were during that session. Using this metric, we calculated the distraction score for each session in both the create and debug workshops. The create group had an average distraction score of 16.6 percent, while the debug group had an average distraction score of 11.8 percent. Figure 4 shows the distribution of the distraction scores over the different sessions. Even though the means of the distribution would suggest that the learners in the create group were more distracted than the ones in the debug group, the difference between the distributions is not statistically significant. This shows that when learners in both have to add new blocks, they are equally distracted by other blocks in the toolbox they don't need. However, the number of blocks that was added by the debug group is low compared to the create group. The learners in the create group added a total of 8065 code blocks to their programs, while the learners in the debug group only added 478 blocks to their programs. This difference is not unexpected since the learners in the debug group get a significant part of the code at the start of each exercise. Moreover, the learners in the create group have to add all their code. With each block they add, they can be distracted by other blocks in the toolbox. Consequently, since the learners in the create group added more blocks they also added more blocks they did not need compared to the debug group.

Alternative to adding new blocks, learners also have to change values in existing blocks. If the learners in both the create and debug groups were perfect students, we would expect the learners in the create group to perform more value changes than the learners in the debug group. When creating code from scratch the learners have to add all the blocks themselves. These blocks have default values. To get to a correct solution, the values for each added block have to be changed. Moreover, the learners in the debug group get an incorrect solution which requires only one code edit to get to a correct solution. This code edit can either be the addition of a code block or a value change. Investigating the log data reveals that the create group performs an average of 3739 value changes per session while the debug group performs an average of 3208 value changes per session. Figure 5 shows the normalized distribution of the number of sessions had a certain amount of value edits for both groups. Comparing these distributions using a z-test reveals that there is no

Fig. 3: The normalized distribution of how many sessions spent a specific amount of time programming. The programming time is grouped in bins of two minutes.

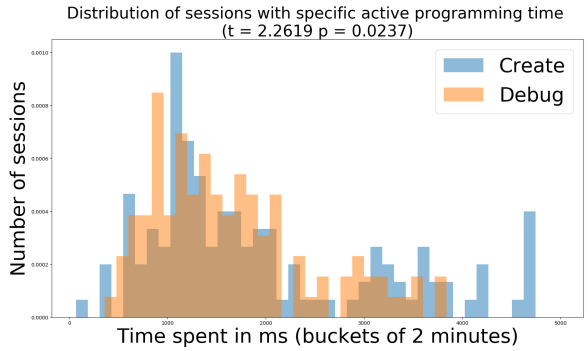
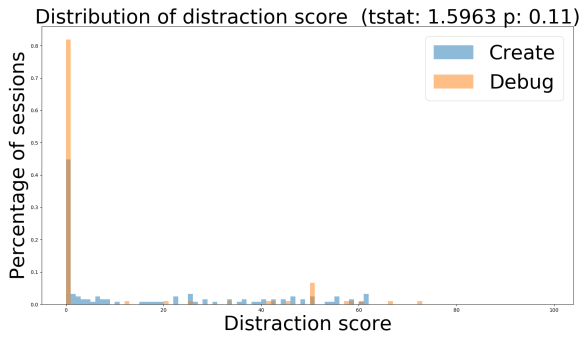


TABLE I: The blocks analysed in the distraction analysis.

Block name	Function
DC-motor	Sets the speed of a dc-motor on a channel
If-then-else	A selection statement based on a condition
Count	Indexed loop from start to end with a step
While	Loop with stop condition
Dwenguino-lcd	Prints a piece of text on a specified location on the screen
Clear-lcd	Removes all text from the lcd-screen
Delay	Waits for a number of miliseconds
Sonar	Measures the distance to the nearest object
Servo	Sets a servo motor on a channel to an angle
Tone on pin	Plays a tone on the buzzer
No tone on pin	Stops the tone on the buzzer
LED on/off	Turns a specific led on or off
LEDs on/off	Turns all 8 LEDs of the microcontroller on or off in one command

Fig. 4: The normalized distribution of the distraction score over all sessions for both the create and debug groups.



significant difference between the distributions ($p = 0.0954$). This shows that the number of extra value edits the learners in the create group have to perform are offset by the value edits required by the learners in the debug group for identifying the bug in the program and fixing it. This unexpected number of code edits in the debug group indicates that the learners in this group more often resort to the strategy of "tinkering".

To identify how much of the learners in each group used tinkering, we extracted the number of code edits and code executions from each session. To quantify the amount of tinkering, we calculated the ratio between the number of runs and the number of code edits. A higher

Fig. 5: The normalized distribution of the number of sessions with a specific number of value edits for both the create and debug group.

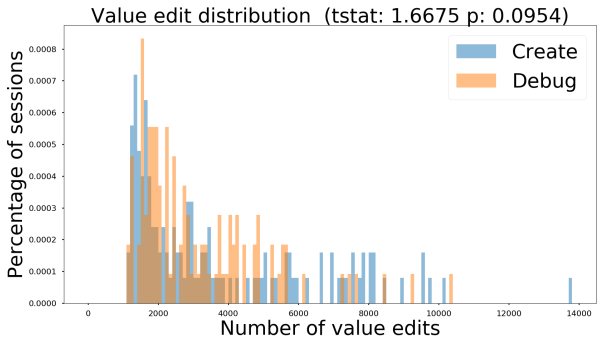


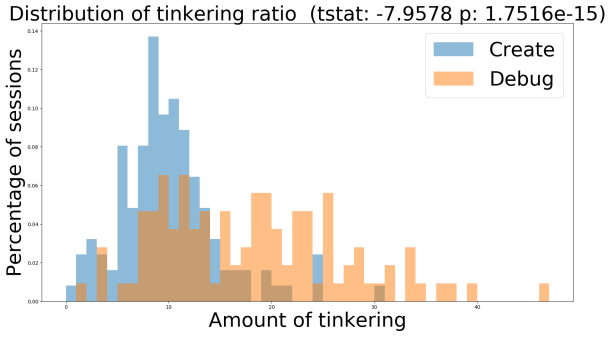
TABLE II: Scores on the programming test for the create and debug groups. (The groups are of different size because some test result were removed because they were incorrectly administered by the teacher.)

Metric	Create (N = 83)	Debug (N = 128)
Average score	1.99	1.63
p-value	0.029	
t-statistic	2.198	

amount of runs compared to code edits indicates more tinkering while more code edits and fewer runs indicates less tinkering [4]. The create group had an average tinkering ratio of 10.35 percent, while the average tinkering ratio for the debug group was 18.34 percent. Figure 6 shows the distribution of the number of sessions that had a specific tinkering ratio. Using a z-test to compare the distributions reveals a significant difference between the groups ($p < 0.001$). Consequently, the learners in the debug group seem to use more tinkering strategies to solve the problems.

To assess if there is a difference in learning performance between the two groups, all learners completed a test with four programming questions after the workshop. The results of these tests, shown in table II, reveal a significant difference between the two groups. The learners in the create group score significantly higher than the ones in the debug group. This shows that the learners in the create group gained a better understanding of the coding concepts they used.

Fig. 6: The normalized distribution of the number of sessions with a specific tinkering ratio for both the create and debug groups.



V. Discussion and Conclusions

Our research aimed to identify the advantages and disadvantages of different methods for integrating programming into a robotics workshop. Based on previous research, we selected two methods for integrating programming as well as four criteria to assess our workshop. These criteria are time spent solving programming problems, distraction by the open programming context, amount of tinkering, and learning performance. We have confirmed that the create group needs more time to solve the programming problems and adds significantly more blocks to their programs [1]–[3]. Consequently, since the learners in both groups are equally distracted by blocks they don't need, the create group is more distracted when counting the absolute amount of distractions. Nevertheless, the debug group scores significantly lower on our programming test. This is likely the result of learners in the debug group resorting to the *tinkering* strategy for solving problems. This is similar to what others have observed when teaching programming to preservice teachers [15]. Based on these results we recommend the create method when integrating programming into an introductory primary physical computing workshop for absolute beginners. The create method requires more time, which results in the learners being distracted more often. However, the need to create code from scratch requires them to get a better understanding of how the different code blocks work, resulting in higher learning performance. Additionally, the debug method facilitates the use of the tinkering strategy which has been shown to result in a lower understanding of programming concepts [8].

We have to point out that our results are valid for our specific context. Our workshop was performed with learners who had no or very limited prior experience with programming. These learners have to acquire the factual and procedural knowledge of programming during this first workshop. We can imagine that learners with a little programming experience would perform better when presented with the debug challenges. This knowledge would allow thew to solve the debugging problems in a more targeted way since they have more knowledge about the context. However, in our group the debug exercises enable the students to avoid the acquisition of factual and procedural knowledge by just trying to change blocks and see what works without understanding what the blocks represent.

Our work brought together two techniques for teaching programming in a physical computing context and quantified their differences using programming log data. We linked these logging results to scores on a programming test to get a better understanding of the advantages and disadvantages of both integration methods leading deeper insights into both methods. Future work should investigate the value of other integration techniques for programming as well as apply the techniques from this paper in other contexts to see if they have different outcomes.

References

[1] J. J. Van Merriënboer and M. B. De Croock, "Strategies for computer-based programming instruction: Program completion vs. program generation," *Journal of Educational Computing Research*, vol. 8, no. 3, pp. 365–394, 1992.

[2] D. W. Shaffer and M. Resnick, "" thick" authenticity: New media and authentic learning," *Journal of interactive learning research*, vol. 10, no. 2, pp. 195–216, 1999.

[3] S. Garner, "A quantitative study of a software tool that supports a part-complete solution method on learning outcomes," *Journal of Information Technology Education: Research*, vol. 8, no. 1, pp. 285–310, 2009.

[4] Z. Liu, R. Zhi, A. Hicks, and T. Barnes, "Understanding problem solving behavior of 6–8 graders in a debugging game," *Computer Science Education*, vol. 27, no. 1, pp. 1–29, 2017.

[5] M. J. Lee, F. Bahmani, I. Kwan, J. LaFerte, P. Charters, A. Horvath, F. Luor, J. Cao, C. Law, M. Beswetherick *et al.*, "Principles of a debugging-first puzzle game for computing education," in *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 2014, pp. 57–64.

[6] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging among novice computer science students," in *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 2005, pp. 84–88.

[7] D. Klahr and S. M. Carver, "Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer," *Cognitive Psychology*, vol. 20, no. 3, pp. 362–404, 1988.

[8] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies," *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 163–167, 2008.

[9] O. Vlaanderen, "Onderwijsdoelen basisonderwijs vlaanderen," 1997, 1997 (accessed 2019-12-09). [Online]. Available: https://onderwijsdoelen.be/resultaten?intro=basisonderwijs

[10] ——, "Onderwijsbegroting," 2019, 2019 (accessed 2019-12-09). [Online]. Available: statistiekvlaanderen.be/nl/onderwijsbegroting

[11] B. Zhong, Q. Wang, and J. Chen, "The impact of social factors on pair programming in a primary school," *Computers in Human Behavior*, vol. 64, pp. 423–431, 2016.

[12] M. Celepkolu and K. E. Boyer, "Thematic analysis of students' reflections on pair programming in cs1," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 771–776.

[13] C. McDowell, B. Hanks, and L. Werner, "Experimenting with pair programming in the classroom," in *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, 2003, pp. 60–64.

[14] T. Bell, J. Alexander, I. Freeman, and M. Grimley, "Computer science unplugged: School students doing real computing without computers," *The New Zealand Journal of Applied Computing and Information Technology*, vol. 13, no. 1, pp. 20–29, 2009.

[15] C. Kim, J. Yuan, L. Vasconcelos, M. Shin, and R. B. Hill, "Debugging during block-based programming," *Instructional Science*, vol. 46, no. 5, pp. 767–787, 2018.