Logistics, Affordances, and Evaluation of Build Programming

A Code Reading Instructional Strategy

Amanpreet Kapoor
Engineering Education
University of Florida
kapooramanpreet@ufl.edu

Tianwei Xie
Computer & Info. Science & Engineering
University of Florida
xietianwei@ufl.edu

Leon Kwan
Computer & Info. Science & Engineering
University of Florida
kwanleon@ufl.edu

Christina Gardner-McCune
Computer & Info. Science & Engineering
University of Florida
gmccune@ufl.edu

ABSTRACT

Computing students are expected to contribute to large unfamiliar codebases as they transition from university to industry settings. While computing courses provide students ample opportunities to write code independently or utilize abstract functionalities from standard libraries, students have fewer opportunities to read or extend codebases written by other programmers. This paper presents the logistics, affordances, and empirical evaluation of a novel instructional strategy, *Build Programming*, which is designed to promote code reading and extension in CS courses. In this strategy, a student (1) solves a programming problem, (2) is assigned a new codebase from a peer who solved the same problem, and (3) is asked to extend the assigned codebase to solve another problem. This allows a student to understand and extend an authentic codebase that is situated in a familiar context. In this paper, we shed light on the logistics of operationalizing this strategy in the context of an undergraduate Data Structures and Algorithms course (N=206). We also describe the affordances of this strategy through student experiences and evaluate the efficacy of one of these affordances, improving code quality through source code analysis. Most students (91%) proposed continuing Build Programming and students' code quality significantly improved after our strategy. Our findings underscore the benefits of Build Programming, and we hope that more instructors incorporate it in CS courses.

CCS CONCEPTS

• **Social and professional topics**~Computing Education

KEYWORDS

code reading, instructional strategy, code quality, pedagogy

ACM Reference format:
Amanpreet Kapoor, Tianwei Xie, Leon Kwan, and Christina Gardner-McCune. 2023. Logistics, Affordances, and Evaluation of Build Programming: A Code Reading Instructional Strategy. In *Proceedings of 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15-18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569756

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569756

1 INTRODUCTION

Computing students are expected to contribute to large unfamiliar codebases as they transition from university to industry settings in order to extend functionality [34, 40] and complete code reviews to verify functionality [36]. Research shows that software engineers spend 60% of their time reading code in the industry [27]. While core computing courses provide students several opportunities to produce code or reuse code from standard libraries, students have fewer opportunities to read or extend codebases written by other programmers [6]. Our paper presents the logistics, affordances, and evaluation of a novel instructional strategy, *Build Programming*, which is designed to promote code reading and extension in computing courses. In this strategy, a student (1) solves a programming problem, (2) is assigned a new codebase from a peer who solved the same problem, and (3) is asked to extend the assigned peer's code to solve another programming problem. This allows a student to understand and extend an authentic codebase that is situated in a context that they are familiar with. Thus, we hypothesize that *Build Programming* can scaffold code reading whereby the familiar context of the problem acts as a scaffold and the student is expected to navigate the structural differences in the assigned codebase in order to extend the functionality. Our work presents rich descriptions on how we implemented *Build Programming* in the context of a Data Structures and Algorithms (DSA) course. We also describe student perceptions of affordances of our strategy and conclude with evaluation of one of these affordances, improving code quality.

2 PRIOR WORK

Code reading: Code reading is a common software engineering practice as developers are expected to work in teams extending the functionality of code written by other programmers [32, 34, 40]. Within Computing Education Research (CER), work on code reading has focused on designing systems to scaffold code reading [18, 33] and understanding students' code reading or code comprehension behavior through eye tracking [4, 5, 7, 17, 22]. There is also a large body of work [23, 25, 26] that has examined students' code tracing and code comprehension behavior and some studies have used systems [28, 42] or diagrams [12] for scaffolding code tracing. These

studies have used short programming problems for tracing rather than large codebases and have found that students often have difficulties tracing and reading code written by others.

Instructional strategies that promote code reading have also been suggested such as *code deconstruction* which encompasses the process of reading, tracing, and debugging code [14], *Explain in plain english* activities in computing courses [10], or *Reading aloud* where a learner reads out loud their code [43]. *Pair programming* [8, 24] is also prominently used but more often for collaboration rather than code reading. Our strategy contrasts pair programming as our focus is on individual contributions and code reading. Other instructional techniques for promoting code reading include DeClue's work on pair programming coupled with pair trading [13]. DeClue found that students recognized the need for documentation and understood the importance of comments and code design after pair trading (working on other codebases). Our intervention is similar to DeClue's work as students extend codebases written by others. However, our work did not require students to pair program and we deployed a mixed methods approach to evaluate the efficacy of our intervention rather than a purely qualitative approach taken by DeClue. In addition, while the motivation behind our intervention was to promote code reading by using context as a scaffold, DeClue's work was motivated by the collaborative aspects of pair programming. Lastly, *remix* approaches to programming have been used to introduce students to coding [1, 35, 37]. In this approach, students inspect and edit the code of existing projects enabling them to see the underlying structures of code and reverse engineer the solution. The remix approach reduces student anxiety [37] and are usually designed for scaffolding learning of CS concepts rather than code reading.

Code quality: Code quality can be assessed in different dimensions such as readability, maintainability, reusability, etc. [32]. Within computing education, researchers have assessed code quality extensively in the context of pair programming [8, 16, 29, 45], creative projects [15], and have determined the linkage between perception of code quality (readability) and code writing [44]. For example, Omar et. al. [29] developed the Java Quality Measurement Tool (JaQMeT) to assess code quality in terms of correctness or complexity in the context of pair programming and Hanks et. al. [16] assessed the code quality of students who participated in pair programming. Hanks et. al. [16] found moderate evidence that students wrote shorter and less complex code if they participated in pair programming.

The most common dimension to assess code quality in CER is readability [2]. *Code readability* has been defined as the human judgment of how easy it is to understand a program [3, 30]. Buse and Weimer [3] investigated the association between the human notion of readability and source code features related to the structure, density, logical complexity, and documentation in a given program. They found that the *average number of identifiers* and *average line length* were the top two predictors of highly readable code, and both features negatively correlated with readability i.e., the longer the *average line length* or greater the *average number of identifiers*, the lower the readability. We use these two metrics to evaluate the impact of Build Programming on students' code quality.

Papers investigating code readability in CER have also used *comment ratio* as a metric. Comment ratio is the number of comments per line of code and higher values of the ratio indicate better readability [3]. For instance, Ciolkowski and Schlemmer [8] investigated if pair programming can improve readability metrics and found that students who worked in pairs had a lower comment ratio when compared with students who collaborated without pair programming. On the contrary, Hulkko and Abrahamsson [19] observed that practitioners and students who participated in pair programming had a higher comment ratio than solo developers, suggesting that pair programmed code was more readable than solo code. Given the prominence of *comment ratio* as a metric for assessing code quality, we use it as a third metric to gauge the impact of Build Programming on code quality.

3 LOGISTICS

3.1 Description of Build Programming

Build Programming is an instructional strategy that we have designed to promote code reading and extension. In this strategy, a student first independently solves a programming problem, then is randomly assigned a codebase of another student who solved the same problem, and finally the student is asked to build upon the assigned codebase to solve another problem. This allows a student to understand and extend a codebase that is situated in a context that they are familiar with. Instructors can leverage Build Programming in a variety of courses as all that is needed to incorporate this strategy is a decomposable problem. Varying complexity can be added to a course assignment and our strategy can be used for eclectic types of assessments such as short coding problems, lab assignments, or projects.

3.2 Operationalizing Build Programming

We operationalized Build Programming in the context of a large undergraduate DSA course at a public university in the US in Fall 2021. Students were expected to complete three projects as a part of our course: two independent projects (each carrying 10% weight of the grade) and a third group project. We utilized Build Programming in the first two independent projects. In the second project, each student was assigned a random peer's codebase of the first project and was instructed to read and extend the assigned codebase (see Figure 1).

Project 1 (Pre-measure of code quality): As a part of the first project, students individually implemented a non-templated self-balancing binary tree data structure called AVL tree in C++. No starter files were provided to the students and the students were expected to design their own interface. The goal of the project was to model a student database as an AVL tree where the student ID was the primary key that was used to organize the elements in the binary search tree-based data structure. Students

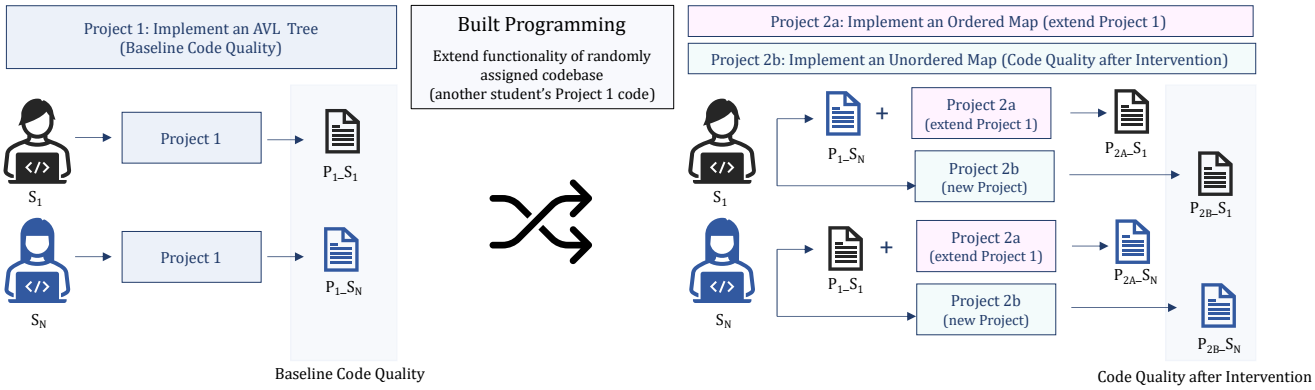

Figure 1: Logistics of Build Programming

were expected to parse input commands and call respective methods for nine operations (e.g., insert, remove, search, etc.). They were given five input and output-based public test cases to test their program and were also encouraged to build unit tests using the C++ catch framework. They were graded for correctness based on 15 test cases (5 public and 10 additional hidden tests) each carrying 5 points (75 points in total), a documentation report consisting of an analysis of the runtime of their program (15 points), and their coding style and design (10 points). For the code style and design, we graded them based on comments, whitespaces, naming convention, proper modularity, clean API, and appropriate memory management. Additionally, they could score five bonus points for unit testing their code and submitting the catch tests. However, their total score was capped at 100 points. While 387 students completed our course, 206 students consented to research and submitted both projects which form our corpus for analysis (see Section 4.2). These 206 students, on average, wrote 586 lines of code (Min = 207, Max = 2182, SD = 194) in Project 1 and the Project 1 grade average was 81.4 out of 100 (Min = 16.2, Max = 100, SD = 14.4).

Project 2 (Intervention and post-measure of code quality): The goal of Project 2 was to compare an ordered map and an unordered map by implementing and extending the functionality of an assigned codebase. This project was segmented into five parts: (A) Implementing the tree-based or ordered map [20 points], (B) Implementing the hash-table based or unordered map [40 points], (C) Comparing the performance of ordered and unordered map [20 points], (D) Reviewing the code quality of the assigned codebase [10 points], and (E) Getting an approval from the author of the assigned codebase [10 points].

For this project, students were randomly assigned AVL Tree codebases (Project 1) of another student. They were asked to extend the assigned codebase to implement another data structure called an ordered map (Project 2a), an abstraction over a self-balancing binary tree. Students were expected to read and comprehend the peer code and create an abstraction over the peer's AVL Tree. They were encouraged to reach out to the author in case they had a question but were required to reuse the peer's code. To ensure a student built on top of the assigned codebase, 10 points were allocated for the author's approval (Part D). The assignee who extended the codebase was asked to reach out to the author, who independently filled out a Google form to verify reuse. The students were not informed beforehand (during Project 1) that they will be randomly assigned Project 1 peer codebases for Project 2. This was deliberate as we wanted students to write code more naturally in Project 1.

During the assignment of random codebases of Project 1, some students did not pass all tests in our test suite. To ensure students could work on codebases that were functional, we assigned codebases that passed at least 80% tests. In professional settings, developers may be asked to work on messy and unfamiliar codebases and we deliberately wanted to introduce this randomness which improves external validity [11]. 80% of the 206 students were assigned random codebases (n=164) while 42 students were assigned a codebase that was volunteered by a student for use in place of projects that didn't meet the 80% pass rate. We did not provide an editorial solution codebase to ensure students have an authentic experience of navigating a codebase written by a student rather than the course staff. For the 42 students who were assigned the voluntary codebase, the course staff acted as an author for code extension approval.

In the second half of this project (Project 2b), students individually implemented an Unordered Map and compared this implementation with their extended implementation of an Ordered Map (Project 2a). This codebase (2b) can give us insight into students' code quality after they had worked on the assigned codebase. On average, 206 students wrote 452 lines of code (Min = 114, Max = 1502, SD = 179) in Project 2b and the Project 2 grade average was 91.9 (Min = 0, Max = 100, SD = 15.0).

We used the Instructure's Canvas Learning Management System [20] to organize the projects and the random assignment peer review feature to assign codebases. The codebases that did not pass the threshold tests were manually reassigned. In addition, we used Google forms to gather feedback on Part D - Reviewing the code quality of your peer and Part E - Getting approval from your peer on the usage of their codebase [21].

4 METHODS FOR EVALUATION

4.1 Study Design

The purpose of our study was to gather preliminary feedback from the students on Build Programming and develop potential hypotheses from a qualitative analysis that can be subsequently assessed using experiments or quasi-experiments. To achieve this purpose, we designed a mixed methods study that followed a pre-

post design [39] in the context of a large undergraduate DSA course in Fall 2021. A pre-post design is a form of pre-experimental design and does not have a control group. This design can aid researchers in discerning whether a phenomenon is worthy of potential investigation with fewer overheads before running a formal experiment [31]. We aim to answer the following research questions through our study:

RQ.1. What are the student perceptions of the affordances of *Build Programming* instructional strategy? How did they receive the activity?

RQ.2. How effective is the *Build Programming* instructional strategy in improving a student's code quality as measured through readability metrics such as comment ratio, average line lengths, and average identifiers per line?

To understand student perceptions of the affordances of Build Programming (RQ.1), we used qualitative responses from student reflections in a post-survey. Based on this analysis, we came up with RQ.2 as students stated that our activity helped them in recognizing the importance of code quality. To evaluate if our strategy influenced students' code quality and answer RQ.2, we took a quantitative approach and analyzed students' source code before and after exposure to Build Programming (see Figure 1).

4.2 Participants and Study Context

Our study population is undergraduate students enrolled in a computing program. Our sample is drawn from students enrolled in an undergraduate DSA class at a public university in the US. The DSA course is a required course for CS and Computer Engineering majors and follows the CS1, CS2, and Discrete Mathematics courses in our program. The language of instruction is C++. 387 students completed the course in Fall 2021 of which 224 students consented to share their data for research in an IRB-approved survey (Response rate: 58%). There was no incentive offered to gain students' consent and students voluntarily completed this survey. 206 of the 224 students completed both projects and hence we discarded 18 students with missing data. Therefore, our final corpus consists of codebases and survey data from 206 students.

4.3 Data collection and analysis

To understand student perceptions of the affordances and reception of *Build Programming* (RQ.1), we use student responses from two open-ended questions that were a part of a post-survey: (1) *What did you learn from this activity?*, and (2) *Should this project be continued in the future? Any other comments?* These responses were analyzed using inductive content analysis and open coding following a constant comparison technique[41].

To evaluate if the strategy influenced the student's code quality and answer RQ.2, we followed a quantitative approach to analyzing source code before (Project 1) and after (Project 2b) the Build Programming intervention (Project 2a). Our independent variable is a repeated measures variable which is *time* (i.e., before and after the build programming intervention). Our dependent variable includes three code readability metrics: *comment ratio* (of block as well as single line comments), *average line length*, and *average number of identifiers per line*. These metrics were selected based on: (1) their high correlation to the human notion of readability as described in prior work from Buse and Weimer [3] and (2) their usage in prior CER literature (see Section 2).

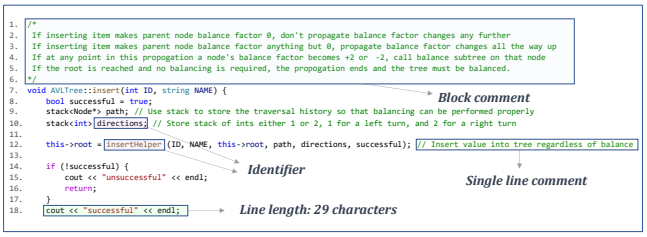

Figure 2: Examples of code quality metrics

In these readability metrics, the *comment ratio of block comments* denotes the number of comments spread over multiple lines divided by the total lines of code whereas the *comment ratio of single-lined comments* denotes the number of comments spanned over one line divided by the total lines of code. *Average line length* is defined as the average number of characters in a line (see Figure 2). Finally, *identifiers* are any names used to describe a variable, function, class, module, or user-defined entities. Based on Buse and Weimer [3], the metrics that are positively correlated with readability include the *average number of block comments* and *average lines of single-lined comments*. The *average line length* and the *average number of identifiers* per line are negatively correlated with readability. This implies that shorter source code line lengths may suggest a more readable program or having more comments may improve readability. Similarly, long chains of objects and sub-properties should be avoided as they negatively impact code readability [38].

To compute each readability metric, a parser script was written in Python 3.8, which imported clang.cindex [9], a python binding of clang library that parsed C++ source code into tokens. Using this script [21], tokens labeled as identifiers and comments were extracted and counted for each of the source files and a csv file was generated that consisted of anonymous student ids along with readability metrics for both projects.

To compare differences in code quality across our population, we used a paired samples t-test for normally distributed data and a non-parametric equivalent when the assumptions of normality were not met. A Shapiro-Wilk test was used to check for normality, and we found that only the *average number of identifiers per line* data was normally distributed. Thus, we used a paired sample t-test to test the following null hypothesis: *There is no difference between the mean paired average number of identifiers per line before and after the build programming intervention in the source code of our population of undergraduate computing students.* A p-value of less than 0.05 was used to reject our null hypotheses. For the other dependent variables: *comment ratio* and *average line length* the assumptions of normality were not met. Therefore, we used a Wilcoxon signed rank test which is used to test the following null hypothesis: *There is no difference in the median paired code quality metrics (e.g., comment ratio) before and after the build programming intervention in the source code of our population of undergraduate computing students.* The alternative hypothesis assumes that the difference in the median is greater than 0 for paired code quality metrics. The tests were conducted using scipy.stats library in python.

5 FINDINGS

5.1 Affordances of Build Programming

We asked students in the post-survey what they learned from our project which used Build programming. We inductively coded these 104 open-ended responses using five unique codes.

Promoting and scaffolding code reading: Unsurprisingly, most students (40%, n=42) described that Build programming promoted and scaffolded code reading and extension which was our original intention behind the intervention. For instance, S7 described that they "*learned how to navigate another person's code and what aspects of the code to look at first in order to understand the structure of the code*". Students also described that our technique scaffolded code reading because of familiar context. For example, S214 stated that "*the part on working with someone else's code is simple enough that the focus is entirely on understanding the code without the stress of figuring out the implementation, which I think is a really good way to introduce it*".

Learning computing concepts: 32% of 104 students (n=33) stated that reading the codebase provided them an opportunity to learn various computing concepts such as wrapper classes, pass by reference, helper functions, traversals, command arguments, pointers, memory management, and organization of files. For instance, S88 stated, "*I learned that you could do post/pre/in order traversals with stacks. This is much more efficient than the way I employed*". Along similar lines, S107 stated that, "*the peer code provided to me [for the assignment] taught me a lot of new concepts that I had not touched on, such as shared pointers*".

Importance of code quality: 25% of the 104 students (n=26) suggested that Build Programming supported them in recognizing the importance of code quality metrics such as appropriate whitespace and comments, and consistent code style. S123 reported that "*I learned that I should make my code more readable in case others need to review my work or continue from where I started. I take a lot of shortcuts to cut out code and it might be confusing with how little I comment*". Another student, S125 described that "*I learned that documenting your code is extremely useful to better understand it and work with it faster. In addition, I learned that documenting your code is really important so that others can understand it to be able to work with it*". Not only did the students realize the importance of writing readable code, but they also suggested that they will alter their behavior by writing more readable code. For example, S125 continued that in the future, "*I will document and add more comments everywhere as if I am explaining it to another person. I will add lines of comments in between my functions for better readability*". To investigate this behavioral change, we undertook a source code analysis to determine changes in code quality (Section 5.3).

Alternate ways to solve a problem: A few students (9%, n=9) described that the project involving Build Programming exposed them to alternative ways of solving a problem, new ideas, and made them reflect on what they could have done differently. For instance, S216 stated, "*It was a good learning experience since I learned about a different approach/style of coding than mine*". Another student, S104, mentioned, "*I got to see another way of going about the code that I had already done. It was kind of interesting comparing her code to mine, and seeing how I had it more optimized in places and vise versa*".

Authentic assessment: Finally, five students (5%) described our assessment as authentic and comparable to what they might encounter in the industry. For example, S153 stated, "*I learned what it will be like in the real industry. I am only used to working on my own coding projects in which I know and am familiar with. A lot of my time in the future will be spent reading code to understand its functionality rather than just writing it*".

5.2 Student reception

We asked students in our survey if we should continue the project with Build Programming in the future. 119 student responses to this question were coded into three categories based on valence (affective tone): *Positive*, *Positive with changes*, and *Negative*. In total, 91% of the 119 student responses suggested that we should continue the activity as-is or with the logistical change of assigning working codebases (RQ1).

72.3% of the 119 student responses (n=86) were coded as positive and students enthusiastically responded that the project should continue as-is or with minor modifications. For instance, S1 stated, "*I think so. A lot of my friends from industry in CS (FAANG., AMEX, BOA) have told me its good practice to learn how to use different codebases to adapt to your own solution. A lot of code is already written, sometimes its up to you to understand it, identify problems, and fix it*". 18.5% of responses (n=22) were coded as positive with changes and these students suggested that the project should be continued only after making a change. This change pertained to the assignment of codebases that were either written by the course staff or a working codebase from another student. For instance, S49 stated, "*I believe the project should absolutely be continued in the future. Writing data structures from a small bit of skeleton code is very good practice. […] A critical point though, I do not believe it is a good idea to randomly assign codebases to people. It is a waste of time to be assigned codebases that do not function correctly […]. The instruction team may wish to take a few known-to-be-functional codebases […] and only pull from that pool in the future*". Finally, 9.2% of responses were categorized into negative valence (n=11) as students suggested that the project should be discontinued due to an unfavorable experience. An example response in this category was S38's response who suggested "*I do not believe this project should be continued in the future. It is too dependent on the competence of others*". These 9.2% of students share the same general sentiment as the 18.5% who were also frustrated with working with someone else's code that did not work. This suggests that if codebases are curated for 100% test passing or additional time is given to students to fix their failing tests, this assignment is one that is viable for all students.

5.3 Code Quality Analysis

We conducted a source code analysis to determine the efficacy of Build Programming on improving students' code quality. *Comment ratio* for single line comments decreased significantly before (Median=0.09, Mean(μ)=0.105, SD(σ)=0.06) and after (Median=0.07, μ=0.08, σ=0.06) Build Programming (Z=4.99, N=206, p<0.001).

Although we expected that our strategy would increase the *comment ratio* as more students might be influenced to write comments, our results were in the opposite direction as students wrote fewer comments (24%↓) after our intervention. This difference could be attributed to the grading rubric of Project 1 which allocated one point for writing comments while the project after Build Programming had no explicit points for writing comments. Since we designed our intervention for promoting code reading and not for gauging code quality initially, the two projects did not have similar rubrics and grades might have motivated students to write more comments.

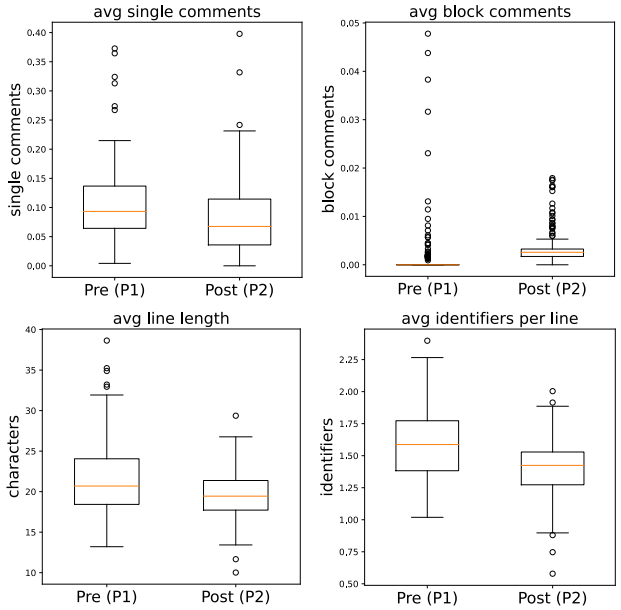

Figure 3: Box plots of code quality metrics

In addition, *comment ratio* for block line comments was significantly different (Z=8.08, N=206, p<0.001) before (Median =0, μ=0.002, σ=0.006) and after Build Programming (Median= 0.003, μ=0.003, σ=0.003). For block comments, however, the results were in the expected direction as students wrote more block comments on average after the intervention. But students wrote fewer block comments than single-line comments (23x of the total block comments on average). The other metrics of code quality: *average line length* and the *average number of identifiers per line* were not explicitly stated in the rubrics of our projects and hence might be more accurate metrics for gauging code quality differences. Students wrote shorter lines of code (7%↓) after Build Programming. The difference between *average line length* before (Median=20.7, μ=21.2, σ=4.3) and after (Median=19.4, μ=19.7, σ=2.9) our activity was significant (Z=5.16, N=206, p<0.001). Finally, for the last metric, the *average number of identifiers per line*, the data followed the normal distribution, and we used a paired two-tailed t-test to test our hypothesis. Students wrote less number of identifiers per line on average (12.5%↓), which indicates better code quality after our intervention, and the results indicate a significant difference between the *average number of identifiers per line* before (μ=1.6, σ=0.3) and after (μ=1.4, σ=0.2) our activity (t(205)=11.2, p<0.001).

6 DISCUSSION AND CONCLUSION

We initially designed the *Build Programming* intervention to scaffold students' code reading by providing them with an opportunity to work with unfamiliar codebases albeit in a familiar context. Through our qualitative analysis, we found that our intervention not only promoted code reading, but also increased students' exposure to alternate ways to implement solutions to DSA problems, enhanced their knowledge of computing constructs, and increased their awareness of the importance of code quality through an authentic experiential ("show") learning approach rather than a "tell" approach. A significant majority of students (91%) expressed that we should continue Build Programming as-is or with minor modifications. Students recommended that they should be provided with fully functional codebases as they were frustrated to work with codebases that had bugs. Since we assigned codebases that passed 80% or more tests, 91% of the 206 students (n=188) were assigned codebases that were not completely correct which led to an unpleasant experience for some students. Instructors can resolve this problem in the future by handpicking selected working codebases for random assignment and then approving the codebase reuse on their end. In our future work, we plan to incorporate Build Programming in more courses as well as other projects as well as run carefully designed experiments to test the efficacy of Build Programming on scaffolding code reading and improving code quality. We recommend other instructors to incorporate Build Programming in their courses given the importance of learning how to read and extend codebases.

7 LIMITATIONS

Our analysis assumes that students' code style would remain the same across projects and we are comparing the code quality before and after the intervention. Hence, our study might be subject to maturation and testing effects as the students might improve their quality of code over the semester while being tested on two different projects. To prevent these effects, counterbalancing cannot be used due to the nature of the intervention, but a better experimental design would include a control group that coded the Ordered map and Unordered map without Build Programming, followed by a comparison of code quality metrics. Another limitation that impacts our analysis is the lack of consistent rubrics across two projects. A more careful experiment would use a similar rubric. Lastly, data coded using qualitative analysis is subject to interpretation biases [11]. We supplement our codes with representative quotes to increase validity and we are transparent about our coding process.

REFERENCES

[1] Amanullah, K. and Bell, T. 2019. Evaluating the Use of Remixing in Scratch Projects Based on Repertoire, Lines of Code (LOC), and Elementary Patterns. *2019 IEEE Frontiers in Education Conference (FIE)* (2019), 1–8.

[2] Börstler, J., Störrle, H., Toll, D., van Assema, J., Duran, R., Hooshangi, S., Jeuring, J., Keuning, H., Kleiner, C. and MacKellar, B. 2018. "I Know It When I See It" Perceptions of Code Quality: ITiCSE '17 Working Group Report. *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (New York, NY, USA, 2018), 70–85.

[3] Buse, R.P.L. and Weimer, W.R. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*. 36, 4 (2010), 546–558. DOI:https://doi.org/10.1109/TSE.2009.70.

[4] Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J.H., Schulte, C., Sharif, B. and Tamm, S. 2015. Eye Movements in Code Reading: Relaxing the Linear Order.

2015 IEEE 23rd International Conference on Program Comprehension (2015), 255–265.

[5] Busjahn, T., Bednarik, R. and Schulte, C. 2014. What Influences Dwell Time during Source Code Reading? Analysis of Element Type and Frequency as Factors. *Proceedings of the Symposium on Eye Tracking Research and Applications* (New York, NY, USA, 2014), 335–338.

[6] Busjahn, T. and Schulte, C. 2013. The Use of Code Reading in Teaching Programming. *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2013), 3–11.

[7] Busjahn, T., Schulte, C. and Busjahn, A. 2011. Analysis of Code Reading to Gain More Insight in Program Comprehension. *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2011), 1–9.

[8] Ciolkowski, M. and Schlemmer, M. 2002. Experiences with a case study on pair programming. *Workshop on Empirical Studies in Software Engineering* (2002).

[9] clang/bindings/python at master · llvm-mirror/clang: *https://github.com/llvm-mirror/clang/tree/master/bindings/python*. Accessed: 2022-08-18.

[10] Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R. and Murphy, L. 2014. "explain in Plain English" Questions Revisited: Data Structures Problems. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2014), 591–596.

[11] Creswell, J.W. and Creswell, J.D. *Research design : qualitative, quantitative, and mixed methods approaches*.

[12] Cunningham, K., Blanchard, S., Ericson, B. and Guzdial, M. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (New York, NY, USA, 2017), 164–172.

[13] DeClue, T. 2003. Pair programming and pair trading: effects on learning and motivation in a CS2 course. *Journal of Computing Sciences in Colleges*. 18, (2003), 49–56.

[14] Griffin, J.M. 2016. Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging. *Proceedings of the 17th Annual Conference on Information Technology Education* (New York, NY, USA, 2016), 148–153.

[15] Groeneveld, W., Martin, D., Poncelet, T. and Aerts, K. 2022. Are Undergraduate Creative Coders Clean Coders? A Correlation Study. *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022), 314–320.

[16] Hanks, B., McDowell, C., Draper, D. and Krnjajic, M. 2004. Program Quality with Pair Programming in CS1. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2004), 176–180.

[17] Herman, G.L., Meyers, S. and Deshaies, S.-E. 2021. A Comparison of Novice Coders' Approaches to Reading Code: An Eye-tracking Study. ASEE Conferences.

[18] Hoffman, D.M., Lu, M. and Pelton, T. 2011. A Web-Based Generation and Delivery System for Active Code Reading. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2011), 483–488.

[19] Hulkko, H. and Abrahamsson, P. 2005. A Multiple Case Study on the Impact of Pair Programming on Product Quality. *Proceedings of the 27th International Conference on Software Engineering* (New York, NY, USA, 2005), 495–504.

[20] Instructure | Educational Software Development: *https://www.instructure.com/*. Accessed: 2022-08-18.

[21] Kapoor, A., Xie, T., Kwan, L. and Gardner-McCune, C. 2022. Logistics, Affordances, and Evaluation of Build Programming: A Code Reading Instructional Strategy.

[22] Kather, P., Duran, R. and Vahrenhold, J. 2021. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *ACM Trans. Comput. Educ.* 22, 2 (Nov. 2021). DOI:https://doi.org/10.1145/3480171.

[23] Kumar, A.N. 2013. A Study of the Influence of Code-Tracing Problems on Code-Writing Skills. *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2013), 183–188.

[24] Kuppuswami, S. and Vivekanandan, K. 2004. The Effects of Pair Programming on Learning Efficiency in Short Programming Assignments. *Informatics in Education*. 3, 2 (2004), 251–266. DOI:https://doi.org/10.15388/infedu.2004.18.

[25] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2004), 119–150.

[26] Lopez, M., Whalley, J., Robbins, P. and Lister, R. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. *Proceedings of the Fourth International Workshop on Computing Education Research* (New York, NY, USA, 2008), 101–112.

[27] Mistrik, I., Galster, M., Maxim, B.R. and Tekinerdogan, B. 2021. *Knowledge Management in the Development of Data-Intensive Systems*. CRC Press.

[28] Nelson, G.L., Xie, B. and Ko, A.J. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (New York, NY, USA, 2017), 2–11.

[29] Omar, M., Romli, R. and Hussain, A. 2008. Automated tool to assess pair programming program quality. (2008).

[30] Posnett, D., Hindle, A. and Devanbu, P. 2011. A Simpler Model of Software Readability. *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), 73–82.

[31] Pre-Experimental Designs | Research Connections: *https://www.researchconnections.org/research-tools/study-design-and-analysis/pre-experimental-designs*. Accessed: 2022-08-18.

[32] Pressman, R.S. 2005. *Software Engineering: A Practitioner's Approach*. Boston.

[33] Raymond, D.R. 1991. Reading Source Code. *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research* (1991), 3–16.

[34] Reading Code Is an Important Skill. Here's Why.: 2021. *https://builtin.com/software-engineering-perspectives/reading-code*. Accessed: 2022-08-17.

[35] Richardson, I., Cypher, M., Hinton, S., Hutchinson, A., McMullan, J. and Whitkin, J. 2011. Remix, mash-up, share: Authentic assessment, copyright and assessment policy in interactive media, games and digital design. (2011).

[36] Sadowski, C., Söderberg, E., Church, L., Sipko, M. and Bacchelli, A. 2018. Modern Code Review: A Case Study at Google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2018), 181–190.

[37] Sanchez, J. 2017. Overcoming the Fear of Coding: A Qualitative Analysis of the Remix Approach. *EdMedia+ Innovate Learning* (2017), 1006–1010.

[38] dos Santos, R.M. and Gerosa, M.A. 2018. Impacts of Coding Practices on Readability. *Proceedings of the 26th Conference on Program Comprehension* (New York, NY, USA, 2018), 277–285.

[39] Shek, D.T. and Zhu, X. 2018. The SAGE Encyclopedia of Educational Research, Measurement, and Evaluation. SAGE Publications, Inc.

[40] Six Reasons Why Reading Code Is More Important Than Writing: 2021. *https://betterprogramming.pub/6-reasons-why-reading-code-is-more-important-than-writing-21e7b0b62203*. Accessed: 2022-08-17.

[41] Strauss, A. and Corbin, J. 2008. *Basics of qualitative research: Grounded theory procedures and techniques*. SAGE Publications, Inc.

[42] Striewe, M. and Goedicke, M. 2014. Code Reading Exercises Using Run Time Traces. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (New York, NY, USA, 2014), 346.

[43] Swidan, A. and Hermans, F. 2019. The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. *Proceedings of the ACM Conference on Global Computing Education* (New York, NY, USA, 2019), 178–184.

[44] Wiese, E.S., Rafferty, A.N. and Fox, A. 2019. Linking Code Readability, Structure, and Comprehension Among Novices: It's Complicated. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2019), 84–94.

[45] Xu, S. and Rajlich, V. 2005. Pair Programming in Graduate Software Engineering Course Projects. *Proceedings Frontiers in Education 35th Annual Conference* (2005).