Visualizing Code Patterns in Novice Programmers

Jeff Bulmer
Dept. of Computer Science
University of British Columbia
bulmer.jeffk@gmail.com

Angie Pinchbeck
Dept. of Computer Science
University of British Columbia
angie.pinchbeck@gmail.com

Bowen Hui
Dept. of Computer Science
University of British Columbia
bowen.hui@ubc.ca

ABSTRACT

Many researchers have investigated the difficulties faced by novice programmers. However, these approaches have so far focused primarily on the identification and correction of common syntax errors, or that of topic difficulty in the CS1 curriculum. Meanwhile, poor coding practices adopted by students have gone mostly unaddressed. While these practices may not necessarily lead to erroneous code, they may nonetheless indicate areas of difficulty and lead to poorly structured programs. To address these issues, our project examines students' coding habits and common errors in CS1 exercises gathered from 77 first-year students. This data was collected in real time so that we may later reconstruct the thought process of the student while solving the programming exercises. To assist our analysis, we built a code visualizer that animates the programming process dynamically and summarizes error metrics simultaneously. Our ultimate goal is to use the code visualizer to help either an instructor or a student to identify poor programming practices during the coding process. With the error metrics gathered, an instructor can inspect potential improvements in coding behaviors for an individual student at a given point in time or over time, and identify bad coding habits common to populations of students.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Information systems** → *Data analytics*;

KEYWORDS

Source code snapshot analysis; programming session trace analysis; code patterns; programming behavior;error diagnosis; CS1; code metrics; learning analytics; self-regulation

ACM Reference Format:
Jeff Bulmer, Angie Pinchbeck, and Bowen Hui. 2018. Visualizing Code Patterns in Novice Programmers. In *WCCCE '18: 23rd Western Canadian Conference on Computing Education, May 4–5, 2018, Victoria, BC, Canada.* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3209635.3209652

1 INTRODUCTION

Many programmers' first experience with code is in Java-based CS1 courses. For this reason, it is crucial that these courses address common student mistakes and misconceptions as effectively as possible. As educational data mining and learning analytics have grown in popularity [8], so too has the number of studies that aim to identify and classify these mistakes. To date, most of these studies have focused on syntax errors, i.e., errors in punctuation, spelling, or order of terms within code. Although most modern integrated development environments (IDEs) used in CS1 courses are quick to point out these errors, they are rarely helpful when trying to address what specific coding behavior led to the error. Here, we define "behavior" as the series of coding habits that lead up to and ultimately cause an error, either syntactically or conceptually. For example, a `NullPointerException` in Java occurs any time a program attempts to access a non-existent reference variable. In the context of CS1, this problem could arise due to multiple reasons, such as not properly setting the values of an array, or writing a constructor that does not assign a value to every attribute. Ideally, helping a student identify and prevent these causes in the code arguably improves one's learning, more so than simply generating a `NullPointerException`.

To this end, studies have aimed to improve the quality of error reporting. Hristova et al. [7] used an enhanced compiler programmed to look for specific errors, which would then display a message with advice on how to correct it. While this approach begins to address the issue of improving student coding habits, it stops just short of identifying bad coding behaviors. After all, some behaviors may cause errors which do not necessarily produce a compiler message (e.g., using a single equal sign = instead of double equal signs == in conditions for comparisons). Moreover, some behaviors may result in errors only when the code is expanded upon, but do not at the time of compilation. For example, suppose only a single element of a large array is undefined, but each run of the program so happens to utilize one of the other array elements that are properly defined. In this case, no error is caught at compile or run time. However, an error-causing behavior is still present, and is left unnoticed.

A few studies examine habits that do not directly result in a compiler error. For example, Hristova et al. [7] caught cases where a condition in a conditional statement or a loop was immediately followed by a semi-colon unintentionally. While this pattern does not result in an explicit error, it produces code that works in unexpected, incorrect ways. Altadmri and Brown [1, 2] caught cases where a method of a non-void type was called, but its return value ignored. While this behavior does not result in an error, it still demonstrates a fundamental misunderstanding regarding how methods work.

While not specifically error causing, the aforementioned behaviors may nonetheless indicate a fundamental misunderstanding of important CS1 concepts. If left unaddressed, these behaviors can lead novice programmers into developing bad coding habits that will undoubtedly hinder them later in their careers. For example, a student who never learns to properly indent their code will go on to create code that is difficult to read and cumbersome to debug. Likewise, a student who does not know when to use `==` versus `.equals` in conditions will create programs that cannot operate with objects properly.

In this study, we attempt to identify and address bad coding habits by automatically detecting relevant code patterns and visualizing the coding process in real time. The main difficulty is that little consensus exists as to what constitutes a bad habit outside of explicitly error causing code. For this reason, we examine a subset of commonly known syntax and semantic errors, in addition to a selection of poor coding practices chosen by the authors. This combination allows us to examine the link between certain coding practices and the occurrence of well known errors, as well as providing a point of reference to similar work.

We begin in Section 2 by discussing work related to student error analysis and code visualization. In Section 3, we describe the design, implementation, and intended usage scenarios of the code visualization tool. Section 4 explains the data collection phase of our study and Section 5 reports on the results of our study. We hope that this work will serve as a starting point for others to further examine habits and non-error causing patterns in novice code.

2 RELATED WORK

2.1 Detecting Students' Coding Errors

In recent years, there has been a significant amount of research on how students code. Many efforts have focused on identifying difficult concepts in a curriculum (e.g., see [3, 4, 9]). In these works, the researchers usually present students with a set of questions that aim to test a variety of CS1 concepts, and then track the number of correct responses (as in [3, 4]). Caceffo et al. [3] made the questions used in their study available for use as a concept inventory. Norris et al. [9] developed an extension for the IDE BlueJ that tracked successful compilations, using these as a metric instead of "correct" responses. After using their extension to monitor 75 students from three CS1 sections, they noted several trends, including a positive correlation between performance and time spent and number of compilations. The researchers in [9] also noted there may be patterns associated with successful and unsuccessful students, though they were unable to directly identify any in their work.

Another area of research focuses on examining specific compilation errors (e.g., see [1–3, 7]). Altadmri and Brown [1] used both compiler error messages and customized parsers to identify instances of several specific errors from a defined set. In their work, bypassing the compiler provided them with the ability to detect semantic errors that did not directly result in a compiler response, in addition to differentiating between errors that result in the same compiler message. The focus on identifying specific errors allowed these researchers to use error frequency as a metric to better identify not only what concepts students have trouble with, but what about those concepts is so difficult. For example, while Hristova et al. [7] posited that using characters other than semi-colons as separators in for-loops was a common error among CS1 students, Altadmri and Brown's study [1] used a significantly larger dataset and found it was actually among the rarest of those errors examined.

The most relevant study is the work by Hristova et al. [7]. The authors identified three categories of errors: syntax, semantic, and logic errors. The authors counted as syntax errors any mistakes in the spelling, punctuation, or word order of the code. Semantic errors included any errors in the meaning of the code. While syntax errors are nearly always caught by the compiler, semantic errors are not. Logic errors were the most general type of error, and included errors resulting from a misunderstanding of how the language interprets the code. In contrast, we fold the category of logic errors into semantic errors and reuse these definitions.

Hristova et al. interviewed computer science professors and educators to develop a list of 20 common Java programming errors made by CS1 students. In their work, Hristova et al. implemented an multi-pass pre-processor called "Expresso" which addressed each instance of an error from the list with a tailored error-message. Their goal was to enhance the functionality of the compiler by improving the understandability of error-messages. Notably, their focus was almost entirely on the development and implementation of their error-detection tool, and made no further conclusions about their list of common errors. While the goals of our study are different, we adopt a subset of the errors identified, and include additional coding patterns that go undetected by the compiler.

Building upon the previously mentioned study are those of Altadmri [1] and Brown [2], which examined the same errors as Hristova in a large data set spanning a whole academic year. These papers looked at both error frequencies and educator beliefs to check first the accuracy of educators' views on mistake frequency [2], and second how the frequency of various errors evolves over the course of the academic year [1]. These papers aimed mostly to inform educators of difficult topics and student issues to improve their courses, and while our goal deviates slightly from that, much of Altadmri and Brown's methodology is interesting to us. Firstly, while both studies used mistakes first identified by Hristova et al., two mistakes were removed from Hristova's original twenty. These mistakes were "leaving a space after a period when calling a method" and "improper casting". The first of these does not cause an error in Java programming, and therefore is more of an unfavorable stylistic choice. The second was not clearly enough described, and was therefore omitted. For our study, we have omitted the same errors. Any errors counted among our "habits" are taken from Altadmri and Brown‘s subset of Hristova‘s original list of errors.

In a similar vein, most other researchers used errors and warning messages to track student progress and identify problematic subjects in computer science education. The work of Caceffo et al. [3], for example, aimed to create a concept inventory for computer science education, and as such grouped C errors together into several families. The study by Fernandez-Medina et al. [6] provided additional insight into this, as their study also involved the categorization of errors into well-defined groups.

Finally, the work done by Vihavainen et al. [10] should be mentioned. This study specifically examined coding assignments done in an IDE in the earliest part of a course for students with no prior coding experience. While their work was specific to an IDE, the researchers were interestingly able to group students into categories of learners based on how they write print statements. Due to our methodology, this exact categorization is impossible for our study, but this study brings into consideration the possibility of identifying learner-types based on their errors or idiosyncrasies.

2.2 Program Visualizers

The second area of research we looked at involved using text animated transitions to navigate document histories. Chevalier et al. [5] developed a stand alone tool called Diffimation which allowed users to scroll through a timeline of all a document's revisions. Like with similar tools, at each point in the timeline, insertions and deletions were highlighted at every point in the time line. However, unlike existing tools, which most often would show side by side snapshots of points in the document's history, Diffimation animated the changes at each point to increase visual clarity. This was done by computing the diff at each point, and applying basic animations at each step, i.e. deletions between steps were first highlighted in red, then shrank until no longer visible. Our code visualizer draws its primary inspiration from this tool. However, our goal is not to compare distinct versions of a document, but rather to animate the writing of a document in order to visualize a student's thought process. For this reason, our code visualizer uses direct keystroke input to recreate a written code segment instead of computing the difference between several file revisions. At the same time, we also do not highlight specific revisions between versions of the code, since at any point in time, our visualizer will show the final output generated by all saved keystrokes up to that point.

3 CODE VISUALIZER

When a student is presented with a programming assignment, she may exercise a design-implement process while typing partially working code into an IDE. This process may include several iterations of re-designing the solution and fixing the code. Once the assignment has been submitted, the code would have typically gone through several iterations of changes. As such, we felt that working with submitted code limited our ability to analyze the kinds of errors that students come across during their coding process.

For this reason, we conducted an experiment to gather code samples in real time from CS1 students (see Section 4). An advantage in gathering the code in real time is that we can discover errors that may have otherwise been missed in a final submission. Another reason we collected code in real time is to be able to simulate the coding experience, and hence, reconstruct the thought process of the student while they solved the programming exercises.

As such, we built a code visualizer that animates the programming process dynamically and summarizes error metrics simultaneously. Our aim is to use the code visualizer to help either an instructor or a student to identify poor programming practices during the coding process. The rest of this section elaborates on the design and implementation of the code visualizer.

3.1 System Design

From the user's perspective, the interface for our visualizer consists of three components:

- **Code Panel**: A text pane that shows the (animated) code
- **Error Panel**: A tally of errors for a given code snapshot
- **Media Bar**: A slider that transitions the animation

The design of our system is summarized in the architecture diagram in Figure 1. Our code visualizer allows a user to load in a text file consisting of either a block of Java code, or comma-separated key-codes corresponding to a series of Javascript keystrokes. A known set of bad habits along with the associated code patterns to detect them are stored as part of the system. A text parser then runs to analyze the text for errors, keeping track of how many of each habit has occurred and where. Because each habit is different, and may occur anywhere throughout the code, a different method is required for the identification of each. For this reason as well, our text parser requires one full pass of the code for each specified habit. Once the parser has completed its analysis, the frequency and location of each habit are returned as an error report to be displayed in the error panel.

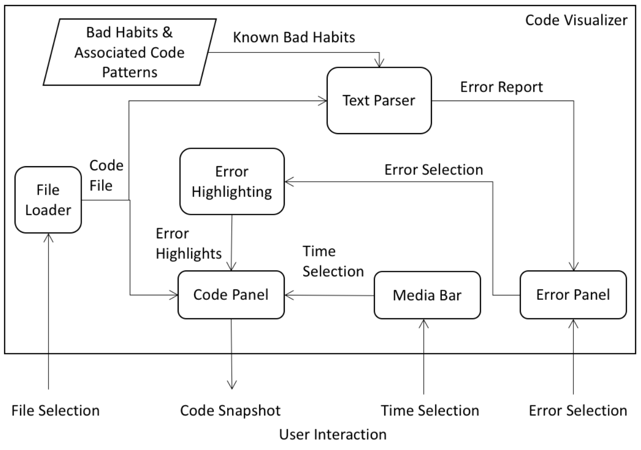
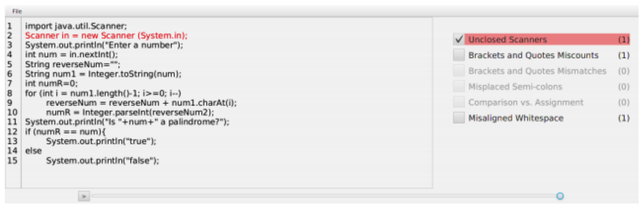


Figure 1: Architecture of the system.

If Java code is loaded into the system, the visualizer will display the code with line numbers in the code panel. Figure 2 shows a screenshot of our implementation. On the other hand, if keystroke data is loaded, then the system will automatically generate Java code from key-codes to create a similar display.



```
import java.util.Scanner;
Scanner in = new Scanner (System.in);
System.out.println("Enter a number");
int num = in.nextInt();
String reverseNum="";
String num1 = Integer.toString(num);
int numR=0;
for (int i = num1.length()-1; i>=0; i--)
        reverseNum = reverseNum + num1.charAt(i);
        numR = Integer.parseInt(reverseNum2);
System.out.println("Is "+num+" a palindrome?");
if (numR == num){
        System.out.println("true");
else
        System.out.println("false");
```

✓	Unclosed Scanners	(1)
☐	Brackets and Quotes Miscounts	(1)
☐	Brackets and Quotes Mismatches	(0)
☐	Misplaced Semi-colons	(0)
☐	Comparison vs. Assignment	(0)
☐	Misaligned Whitespace	(1)

Figure 2: Screenshot of our code visualizer. The left side shows the code panel, the right side shows the error panel, and the bottom shows the media bar.

The error panel is located to the right in the visualizer. A user may select a subset of bad habits to be displayed via a list of check boxes. Each bad habit also has a tally showing the total number of occurrences of that habit present in the code. If there were no instances of that particular error, the check box becomes disabled. If a check box is selected, it will highlight the area of the code where those errors occur in the code panel. For ease of inspection, each error has its own color assigned.

Lastly, the media bar at the bottom of the visualizer enables a user to play the animated code in real time, or jump to any point of the keystroke data. The habit check boxes are enabled as the bad habits appear, and disabled once they are gone.

3.2 Errors and Habits Detected

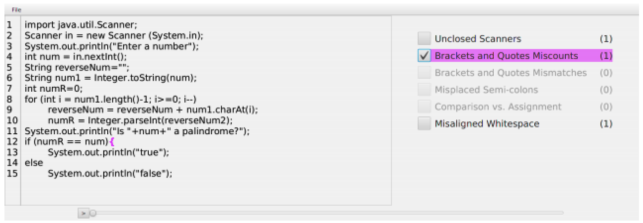
As a start, our code visualizer was designed to detect a small number of errors and habits. Based on our literature review and past teaching experience, we chose the following five to focus on:

- **Unclosed Scanners:** Creating a `Scanner` object but never calling its `close` method
- **Brackets and Quotes Miscounts:** The overall number of open brackets or open single/double quotes being different from the number of close brackets or quotes
- **Brackets and Quotes Mismatches:** Closing a bracket or a quote with a different type of bracket or quote (e.g., "{]")
- **Misplaced Semi-colons:** Placing a semi-colon after a condition in a conditional statement or after a loop declaration
- **Comparison vs. Assignment:** Confusing the equality and assignment operators
- **Misaligned Whitespaces:** Failing to indent consistently within a block of code

3.3 Usage Scenarios

Here, we illustrate the utility of the code visualizer via examples.

3.3.1 Example 1. Figure 2 shows an example that has errors involving: Unclosed Scanners, Brackets and Quotes Miscounts, and Misaligned Whitespace. By checking the box for "Unclosed Scanner", instances of that error becomes highlighted in red in the code panel. Selecting another checkbox, such as "Brackets and Quotes Miscount" as in Figure 3, will result in highlighting a different set of error instances in the code.

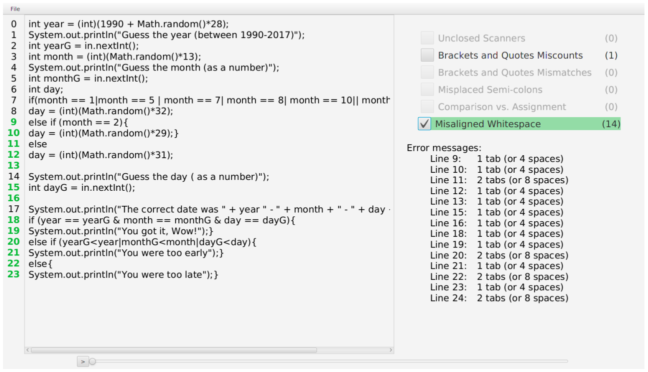


```
import java.util.Scanner;
Scanner in = new Scanner (System.in);
System.out.println("Enter a number");
int num = in.nextInt();
String reverseNum="";
String num1 = Integer.toString(num);
int numR=0;
for (int i = num1.length()-1; i>=0; i--)
        reverseNum = reverseNum + num1.charAt(i);
        numR = Integer.parseInt(reverseNum2);
System.out.println("Is "+num+" a palindrome?");
if (numR == num){
        System.out.println("true");
else
        System.out.println("false");
```

```
☐ Unclosed Scanners               (1)
✓ Brackets and Quotes Miscounts   (1)
☐ Brackets and Quotes Mismatches  (0)
☐ Misplaced Semi-colons           (0)
☐ Comparison vs. Assignment       (0)
☐ Misaligned Whitespace           (1)
```

Figure 3: Example for Brackets and Quotes Miscounts.

3.3.2 Example 2. One behavior that is not recognized by a compiler is poor indentation resulting in misaligned whitespace as in Figure 4. When this error checkbox is selected, we see a series of error messages displayed in the error panel. These messages inform the user the number of tabs or spaces needed for correction, and list the line numbers of each occurrence in the code.



```
0   int year = (int)(1990 + Math.random()*28);
1   System.out.println("Guess the year (between 1990-2017)");
2   int yearG = in.nextInt();
3   int month = (int)(Math.random()*13);
4   System.out.println("Guess the month (as a number)");
5   int monthG = in.nextInt();
6   int day;
7   if(month == 1|month == 5 | month == 7| month == 8| month == 10|| montl
8   day = (int)(Math.random()*32);
9   else if (month == 2){
10  day = (int)(Math.random()*29);}
11  else
12  day = (int)(Math.random()*31);
13
14  System.out.println("Guess the day ( as a number)");
15  int dayG = in.nextInt();
16
17  System.out.println("The correct date was " + year " - " + month + " - " + day
18  if (year == yearG & month == monthG & day == dayG){
19  System.out.println("You got it, Wow!");}
20  else if (yearG<year|monthG<month|dayG<day){
21  System.out.println("You were too early");}
22  else{
23  System.out.println("You were too late");}
```

```
☐ Unclosed Scanners               (0)
☐ Brackets and Quotes Miscounts   (1)
☐ Brackets and Quotes Mismatches  (0)
☐ Misplaced Semi-colons           (0)
☐ Comparison vs. Assignment       (0)
✓ Misaligned Whitespace           (14)

Error messages:
    Line 9:    1 tab (or 4 spaces)
    Line 10:   1 tab (or 4 spaces)
    Line 11:   2 tabs (or 8 spaces)
    Line 12:   1 tab (or 4 spaces)
    Line 13:   1 tab (or 4 spaces)
    Line 15:   1 tab (or 4 spaces)
    Line 16:   1 tab (or 4 spaces)
    Line 18:   1 tab (or 4 spaces)
    Line 19:   1 tab (or 4 spaces)
    Line 20:   2 tabs (or 8 spaces)
    Line 21:   1 tab (or 4 spaces)
    Line 22:   2 tabs (or 8 spaces)
    Line 23:   1 tab (or 4 spaces)
    Line 24:   2 tabs (or 8 spaces)
```

Figure 4: Example for Misaligned Whitespace.

4 DATA COLLECTION

To gather new code samples with real time data, we conducted a data collection experiment with CS2 students who had recently completed a CS1 course. The rest of this section provides the details of this study.

4.1 Participants

Seventy-seven students attending a CS2 course participated in this study. Among them, 60 were first-year undergraduates in a Bachelor of Science program with no declared major, 1 student was a first-year undergraduate in the Management program, and the remaining 16 students were a mix of second, third, and fourth-year students from a variety of majors. The gender ratio composition was 75.3% male, 23.4% female, and 1.3% unspecified.

4.2 Materials

We developed an online system to facilitate the data collection process. As part of this system, we created a short confidence survey and a series of exercises which we call the *coding review* that covers core concepts taught in a CS1 course. The survey asks the participant to indicate their confidence level on the following 11 topics: variables and statements, pre-defined Java classes, user input and output, characters and strings, conditionals, basic loops, nested loops, methods, one-dimensional arrays, objects, and classes.

In the coding review, we asked 10 programming questions on the above topics. Example questions include detecting for palindromes, parsing for odd integers in a number, generating and comparing birthdates, drawing ASCII patterns using nested loops, checking for vowel-ending words in an array, sorting an array of numbers, and creating a bank card class with various methods.

4.3 Procedure

In the first week of a CS2 course in Winter 2018, we recruited students to complete the study online. The online system was available for the entire month of January. Upon completion, students would receive a bonus mark of 1% towards their course grade.

4.4 Measures

In addition to specific errors and habits defined in Section 3.2, we tracked the following metrics to be used in our analysis.

4.4.1 Frequency of Individual Errors. The frequency of analyzed errors is the most obvious metric for analysis, as well as the most common in similar work [1, 4].

4.4.2 Confidence Level of Students. We created a survey with 4-point Likert scale questions to collect self-reported data on students' confidence in various CS1 topics. We chose to use an even-point response scale (strongly agree, agree, disagree, strongly disagree) in order to force participants to evaluate their strengths and weaknesses on CS1 topics. Results were automatically scored, averaged, and associated with a submission.

4.4.3 Amount of Code Written. It has been noted that the best code is not necessarily the longest [9]. We track code length in order to investigate its relationship with bad habits.

4.4.4 Number of Mistakes per User. The number of mistakes made per user may offer some insight into which concepts are most difficult to grasp for students. This metric was also used in [1].

5 RESULTS

Recall that there are 10 programming exercises in the coding review. The structure of the online data collection system saved results once after the first 5 questions and once after the second 5 questions, allowing the participant to take a short break in between. Due to this setup, if the participant decided to quit after the first set of questions, we would not be able to gather data from that student for the remaining questions. As such, among the 77 participants, only 13 completed all 10 programming exercises in the coding review, while the others only completed half of the exercises. We suspect this problem may be due to a lack of clarity in our instructions that all questions must be completed to obtain the bonus marks. Where appropriate, we discuss the results based on data from participants who have completed the first 5 questions (which we refer to as group "Q1-5") versus data from those who completed all 10 questions (refered to as group "Q1-10").

5.1 Frequency of Bad Habits

Table 1 shows the overall number of bad habits made across all participants and questions.

	Frequency
Unclosed Scanners	293
Brackets and Quotes Miscounts	205
Brackets and Quotes Mismatches	44
Misplaced Semi-Colons	11
Comparison vs. Assignment	81
Misaligned Whitespace	40

Table 1: Overall frequency of bad habits.

Among the bad habits detected in our system, we found the most common ones to be Unclosed Scanners and Brackets and Quotes Miscounts. We believe that Unclosed Scanners had the highest count due to the fact that it is not typically taught in the standard CS1 curriculum and it does not generate an error in Eclipse. For these reasons, a student need not address the issue in order for the code to execute properly.

In the case of Brackets and Quotes Miscounts, an IDE, such as Eclipse, usually takes care of this for the programmer. In Eclipse, anytime an open bracket is typed, a closing bracket is automatically generated. (Likewise for a single or double quote.) As a result, a programmer using Eclipse is unlikely to have this error in the code. However, in our online system used for data collection, this feature is not available. Therefore, we suspect that a high frequency count for this bad habit is due to students' overreliance on using IDEs.

With 77 participants in our experiment, Comparison vs. Assignment occurred roughly once per participant. This suggests that students generally understood the difference between the two types of operators, but that they still make the mistake occasionally.

Recall that Brackets and Quotes Mismatches refers to the wrong closing bracket (or quote) being used, whereas Brackets and Quotes Miscounts refers to the absence of a closing bracket (or quote). Our data shows that mismatches occur significantly less often than miscounts. As such, we conclude that students often forgot to close brackets (or quotes), but when they remembered doing so, they typically used the correct ones.

Surprisingly, Misaligned Whitespace did not occur as often as we had anticipated. This suggests that students are manually indenting their code consistently.

Finally, Misplaced Semi-colons was caught only 11 times. This indicates the error is likely not very common, in contrast to what had previously been claimed by Hristova et al. [7]. However, two points should be noted. First, our data was collected from CS2 students who have successfully passed a CS1 course. It is possible that CS1 students learning about conditionals and loops make this mistake much more often when they are learning the new concepts. Second, our dataset is significantly smaller than that of Altadmri and Brown [1, 2], which corroborated Hristova's claim.

5.2 Negative Correlation between Confidence and Number of Errors

Using the self-reported confidence scores on 11 CS1 topics, Figure 5 shows the results averaged over participants.

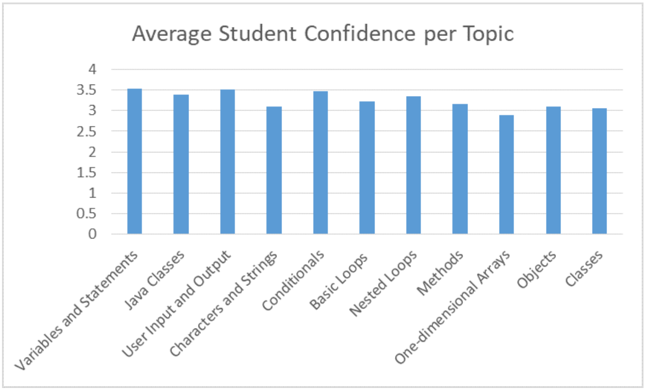


Figure 5: Confidence scores by topic, averaged over participants (N=77)

Given 11 topics and that the most confident score is 4 points on a 4-point Likert scale question, each student can have a total maximum of 44 points for overall confidence. Our data shows the average total is 32.81 (min. 20.0, max. 40.0).

Using Pearson correlation, we found that higher student confidence was negatively associated with total errors in a student's submission ($r = -0.230$ for group Q1-5 and $r = -0.379$ for group Q1-10). This indicates that students who are more confident exhibit fewer bad habits.

Looking at each type of bad habit more closely, Table 2 show that confidence has a weak positive correlation to the number of errors made by individuals in a few cases. However, since these values are so low, the analysis is inconclusive.

	Q1-5	Q1-10
Unclosed Scanners	-0.061	-0.395
Brackets and Quotes Miscounts	-0.117	-0.231
Brackets and Quotes Mismatches	0.114	0.095
Misplaced Semi-Colons	0.056	-0.060
Comparison vs. Assignment	-0.258	-0.130
Misaligned Whitespace	-0.086	-0.403

Table 2: Correlation coefficients on total confidence vs. individual errors (N=77 for Q1-5 and N=13 for Q1-10).

5.3 Negative Correlation between Number of Lines and Number of Errors

Table 3 summarizes the number of lines of code written for each question. It is interesting to note that the maximum lines of code for the many questions (nearly) doubled the average lines of code.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Min.	7	1	5	3	9	5	9	1	16	24
Avg.	23	18	12	14	34	9	20	19	30	39
Max.	74	74	22	33	76	16	30	32	50	49

Table 3: Lines of code for each question (N=77 for questions Q1-Q5 and N=13 for questions Q6-Q10).

Using Pearson correlation, we found that the number of lines written overall was negatively correlated with the total number of errors in a student's submission ($r = -0.177$ for Q1-5 and $r = -0.462$ for Q1-10). Surprisingly, this correlation suggests that fewer lines of code corresponds to more bad habits, even though fewer lines of code means fewer opportunities for mistakes.

Combined with the finding from Section 5.2 that self-reported confidence is negatively correlated with number of errors, we have: (i) higher confidence corresponding with fewer bad habits and (ii) more lines of code corresponding with fewer bad habits. These two results suggest that students with higher confidence also write more code. Indeed, using Pearson correlation, we found a weak positive correlation between the overall number of lines written and the total self-reported confidence score of each student ($r = 0.1594$ for Q1-5 and $r = 0.2995$ for Q1-10).

These findings seem to contradict the results of Norris et al. [9] that found that the best-performing students most often did not write the greatest amount of code. However, it is worth noting that confidence is not equivalent to performance. Further work is needed to test the relationship between the variables discussed here and performance.

6 CONCLUSION AND FUTURE WORK

Our work demonstrates that bad habits can be automatically identified and quantified similarly to syntax errors. Our analysis suggests that some of the habits result from students' overreliance on IDE features (e.g., brackets and quotes miscounts). Surprisingly, we also saw that more confident programmers wrote more code, which exhibited fewer bad habits. We presented a code visualizer that reconstructs student submitted code via text animation, in hopes of identifying why certain mistakes are made in one's coding process. Our aim is to use the code visualizer, in combination with error metrics, to help students stay clear of bad coding habits.

Future work on this project involves conducting a usability study of the code visualizer with our target populations. The visualization and real-time identification of student habits may prove useful to instructors in assisting students to overcome these habits. Additionally, students who wish to improve their own code may use the visualizer for self-monitoring purposes. For these reasons, we believe the code visualizer and similar programs could be indispensable tools for instructors looking to tailor their curricula to better serve the needs of their students.

Additionally, we would like to expand the set of bad habits for analysis, and correlate them with topic-specific as well as overall performance. It would also be interesting to perform a cluster analysis on the data to see if subsets of habits form groups that lead to certain academic performances.

REFERENCES

[1] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 522–527.

[2] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. ACM, New York, NY, USA, 43–50.

[3] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 364–369.

[4] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 695–700.

[5] Fanny Chevalier, Pierre Dragicevic, Anastasia Bezerianos, and Jean-Daniel Fekete. 2010. Using Text Animated Transitions to Support Navigation in Document Histories. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 683–692.

[6] Carlos Fernandez-Medina, Juan Ramón Pérez-Pérez, Víctor M. Álvarez García, and M. del Puerto Paule-Ruiz. 2013. Assistance in Computer Programming Learning Using Educational Data Mining and Learning Analytics. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 237–242.

[7] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)*. ACM, New York, NY, USA, 153–156.

[8] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR '15)*. ACM, New York, NY, USA, 41–63.

[9] Cindy Norris, Frank Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree. 2008. ClockIt: Collecting Quantitative Data on How Beginning Software Developers Really Work. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. ACM, New York, NY, USA, 37–41.

[10] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*. ACM, New York, NY, USA, 109–116.