Teaching Students to Recognize and Implement Good Coding Style

Eliane S. Wiese, Michael Yen, Antares Chen
UC Berkeley
Berkeley, USA
{eliane.wiese, mayen, antaresc} @berkeley.edu

Lucas A. Santos
Federal University of São Carlos
São Carlos, SP, Brazil
lukeaugusto@berkeley.edu

Armando Fox
UC Berkeley
Berkeley, USA
fox@cs.berkeley.edu

ABSTRACT

Teaching students to write code with good style is important but difficult: in-depth feedback currently requires a human. AutoStyle, a style tutor that scales, offers adaptive, real-time holistic style feedback and hints as students improve their code. An in-situ study with 103 undergraduate students in a CS class compared AutoStyle to a control tutor which only offered ABC score. While students improved the style of their code in both cases, students working with AutoStyle were more likely to use an appropriate language idiom and to improve their recognition of good style. However, students struggled to implement style improvements, even when hints recommended specific functions.

Author Keywords

Computer science education; programming style tutor; in-situ experiments

INTRODUCTION: ENCOURAGING BEAUTIFUL CODE

It's hard to teach programmers to write beautiful code, but it's vitally important. We use the term *beautiful code* to mean code that is elegant, efficient, idiomatic, and revealing of design intent [5, 2].

Beautiful code is crucial in professional settings. As Knuth put it in 1984, "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do" (emphasis in the original) [12]. The importance of human-readable code has been borne out quantitatively. Robert Martin, a thought leader in software engineering, recounts replaying a keystroke log of one of his own programming sessions in his book *Clean Code* [17]. He discovers that when he creates new code, he spends only 10% of his time typing it out. 90% of his time is spent reading existing code that the new code would interact with. Since the vast majority of his time is spent reading existing code, he concludes that it is imperative for programmers to write beautiful code that is easy to read. Indeed, the dominant cost incurred during the lifecycle of a successful (long-lived) software system is not bug fixing, but rather maintenance and enhancement of legacy code [8]. Code that is functional but stylistically poor incurs high maintenance costs because it is difficult and time-consuming for a new programmer to understand and modify poor-quality code written by someone else.

L@S 2017, April 20–21, 2017, Cambridge, MA, USA

ISBN 123-4567-24-567/08/06…$15.00

DOI: http://dx.doi.org/10.475/123_4

Measuring the Beauty of Code

Beautiful code goes beyond simply adhering to syntactic coding standards such as indentation, use of whitespace, and the placement of delimiters (e.g., braces). Ward Cunningham is quoted in [17] as stating that beautiful code "makes it look like the language was made for the problem." The Related Work section reviews both quantitative and qualitative methods used by professional programmers to assess the beauty of their code, but these do not eliminate an expert programmer's subjective judgment, including the ability to recognize where improvements in beauty are possible. For novice programmers, therefore, we start at the level of individual methods (functions) and ask: What makes a short function (5–15 lines) beautiful, and can we teach students to recognize and improve code beauty at this level?

Cunningham's goal focuses on matching the problem to the facilities and abstractions available in the language and its core libraries. For example, while both JavaScript and Ruby have some features associated with functional languages, the use of collection idioms (e.g., `map`) is far more prevalent in Ruby, whereas the use of higher-order functions is much more frequent in JavaScript. As another example, both Python and Ruby include basic control flow constructs such as `if...then`, but Ruby also includes `unless`, and allows reversing the clauses in a conditional. As these examples illustrate, stylistic usages depend on the language. Therefore, language-independent guidelines are insufficient.

While Fowler and Beck have stated [7] that "no set of [code] metrics rivals informed human intuition" for improving code quality, Mäntylä et al. [15] found empirically that for the simple "code smells" at the method level, there was high inter-rater agreement between humans and source code metrics on

what refactoring would be appropriate. Informal experimentation by teaching assistants in our intermediate programming courses has shown that the Assignment–Branch–Conditional (ABC) score [6], which measures a weighted sum of three basic types of statements within a function, is a good proxy for idiomatic code at the method level. ABC score rewards conciseness, and code with the best ABC score generally combines an efficient approach and appropriate language idioms. Consequently, ABC score is one of the metrics computed by virtually all tools that compute code metrics, including online services such as CodeBeat.com and CodeClimate.com. While we use ABC score to operationalize good style at the level of individual methods, we note that it does not capture all important style features (e.g., the usefulness of variable names and comments). As idiomatic code is important and often unsupported in introductory CS courses, it is our focus here.

Prior Work: The AutoStyle Tutor

Computing the ABC score does not tell students how to improve it. Providing holistic suggestions for improving style currently requires an instructor to hand-inspect code, a solution that is asynchronous and does not scale to large courses.

AutoStyle [4] is a research system that provides automated, adaptive style hints. The hints suggest syntax shortcuts and offer better approaches to solving the problem. Hints may also include code skeletons, in which the control flow is given but the student must fill in missing lines. AutoStyle's hints are intended to help the student improve their code's ABC score. A prior, randomized, controlled study of AutoStyle with 80 paid participants (students in an introductory computer science class) showed that AutoStyle helped students improve their code [4]. Students were given the standard "style manual" created for the course and were asked to write code that solved a programming problem and achieved a target ABC score. All students were allowed unlimited attempts, and received their ABC score after each one. The intervention group additionally received hints from the AutoStyle tutor [4]. In that study, 70% of the AutoStyle condition reached the style threshold, compared to 13% of the control group.

While this initial evaluation is promising, it examines performance, not learning - it does not demonstrate that students can apply new knowledge or practices outside of the tutor. A follow-up study [3] found that students could learn a coding idiom from AutoStyle and correctly apply while working independently, but it did not compare AutoStyle to a control condition. Finally, while both experiments measured how well students could write beautiful code, they did not examine how well students could *recognize* beautiful code. Recognizing beautiful code is important when deciding between different code implementations, and when identifying areas of one's own code for improvement.

Research Questions

We address our research questions by comparing AutoStyle to a control condition in which students see their ABC score but do not receive hints. Our research questions and brief summary of our results are as follows:

1. **RQ1.** Does working with a style tutor help students write beautiful code? *Yes, like [4], and students use more appropriate idioms with AutoStyle.*

2. **RQ2.** Does working with a style tutor help students write beautiful code independently? *No, unlike [3].*

3. **RQ3.** Does working with a style tutor improve students' ability to recognize beautiful code? *Yes, and students improve more with AutoStyle.*

To investigate these questions we conducted an *in situ* experiment during the summer offering of an introductory programming course. The experiment consisted of three parts: code improvement, multiple choice questions, and coding from scratch. After completing two coding assignments for the course, students were asked to improve the style of their code to meet an ABC score threshold. All students received their ABC score on each attempt, with some students assigned to receive additional hints from AutoStyle. Before and after the code improvement task, multiple-choice questions measured recognition of beautiful code by asking students to select the example with the best style. Finally, students did a code-from-scratch challenge without any feedback.

RELATED WORK

The existing literature on teaching coding style is sparse, both in terms of practical suggestions for teaching and theory for guiding future designs and experiments. This is in contrast to the literature and systems for teaching code correctness, which includes identification of student errors and synthesis of feedback that might help the student transform a defective program into a correct one. Real-time adaptive feedback is a hallmark of these systems, whether given by peer evaluation [13] or automated tools [10].

While automated, real-time feedback is an obvious instructional strategy for coding style, it has not been feasible to provide. Ideally, such feedback would offer personalized, holistic suggestions for improving style. However, currently there are few techniques and tools designed to offer any help on coding style at all.

Current style checkers similar to `lint` [11, 14] can provide style feedback, but only for relatively low-level stylistic problems such as redundant use of Boolean expressions (e.g. shortening `if (b!=false)` to `if (b)`) [1]. Another tool, UglyCode [18] can illustrate the importance of good style. UglyCode starts with good code and allows the student or teacher to apply transformations to obfuscate variable names, mess up indentation/line breaks, add useless comments, etc., to teach good style by uglifying a good piece of code rather than improving a bad one. Foobaz helps students choose useful variable names [9].

However, writing beautiful code goes beyond the low-level transformations addressed by these tools. Grady Booch, a pioneer in object-oriented design, writes: "Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designers' intent but rather is full of

crisp abstractions and straightforward lines of control" [2]. Adhering to low-level code formatting guidelines is insufficient to produce code with these properties.

Refactoring, in which the structure of code is improved without changing its behavior, is closely tied to good code style. However, most scientific work on analyzing refactoring has focused on identifying refactoring opportunities, either by using formal methods and tools or (more widespread) identifying *code smells*, which are (anti-)patterns in source code that increase the cost of maintaining and enhancing it by creating unnecessary complexity [19]. In contrast, the role of humans as decision makers in refactoring has been largely neglected in the professional software engineering literature [16].

OVERVIEW OF THE STYLE TUTOR

Pedagogical Framework

Coding with good style is a metacognitive task: it involves not just producing code, but assessing, planning, and evaluating changes. Closely related to practicing good coding style is *refactoring* - improving the structure of a piece of code without changing its functionality. While refactoring typically occurs across methods or classes, the techniques used can also be applied to individual methods. Our three-step process for improving style draws on a six-step process for refactoring: (1) recognize that a piece of code needs refactoring, (2) determine which refactoring techniques to apply, (3) ensure the refactoring preserves correct behavior, (4) apply the refactoring, (5) assess the effect on quality (i.e. whether further refactoring is needed), (6) modify related artifacts such as technical documentation to reflect the changes [19].

Our three steps for improving coding style are intended to guide assessment and instruction:

i **Assessment.** Identify areas for style improvement. A novice may have no basis for doing this step. This step is similar to step 1 for refactoring.

ii **Information Retrieval.** Efficiently search for a better idiom or strategy. This step is similar to step 2 for refactoring. While experienced programmers don't memorize every language idiom, they do know how to look for them. Novices may not know how to conduct such a search or how to recognize what information is likely to be helpful. Further, even when a novice is familiar with a useful language idiom, they may not realize that it is applicable to their current problem.

iii **Implementation.** Correctly apply the new idiom or strategy to the particular coding task. This step is similar to step 4 for refactoring. This step may be particularly difficult for a novice, as incorporating unfamiliar language idioms may introduce bugs.

This three-step procedure should be repeated as necessary (similar to step 5 for refactoring). This pedagogical framework was developed after AutoStyle was implemented. Consequently, AutoStyle performs step 1 (assessment), supports students in doing step 2 (information retrieval), and does not support step 3 (implementation). AutoStyle detects which aspect of the code should be improved and suggests a function or an approach. When offering syntax hints, the tutor provides the name of a function and directs students to online documentation for more details. This is intended to give the students practice using documentation. AutoStyle does not offer hints on code that is not functionally correct.

Implementation

We briefly review the description of the style tutor; for further details see [4]. In order for AutoStyle to automatically generate hints, a corpus of several hundred previously collected submissions is needed to capture the style variations in students' submissions. Each submission's style is measured with the scalar quantity of Assignment–Branch–Condition (ABC) complexity, also known as ABC score.

Whereas [4] calculates the ABC score as the weighted L2 norm of the ABC vector, we used the equivalently weighted L1 norm, which allowed us to calculate ABC score gains and compare the progression of students' improvements. We preferred the linear properties of the L1 norm since code complexity should scale linearly with the number of operations. The original inventor of the ABC metric, who used the L2 norm, has stated that using a different vector norm may be preferable [6].

The corpus of submissions is clustered using density-based clustering, which is superior to K-means in that outliers far away from a cluster centroid are not "forced" into a cluster; this allows the instructor to identify students who may require more general guidance on problem strategy as opposed to hints for fine-tuning a sound strategy. Normalized tree edit distance is used as a distance metric. Previous studies have shown that clustering ASTs using this distance metric often captures groups of code with similar high-level design [4]. Through the instructor interface, an instructor can provide hints tailored to a specific cluster. In our implementation, AutoStyle provides instructor-written (A) *approach* hints that include links to resources such as language documentation, and (B) *skeletons* created by redacting a similar-but-correct submission from the corpus. Scale is effectively handled as instructor-created hints are proportional to the number of clusters, not the number of submissions, and in general the number of clusters and the number of submissions do not correlate [4].

Upon receiving a submission from a student, AutoStyle grades the submission for correctness by running a suite of test cases. If the code is functionally correct, its abstract syntax tree (AST) is extracted and compared to all other submissions to determine which cluster it would best fit into. Submissions mapping to "weak" clusters (poor ABC scores) are shown approach hints, skeletons, or both.

Submissions mapping to "strong" clusters are shown one or more automatically generated syntax hints. Syntax hints are generated by identifying another submission that has a similar structure but a better ABC score. Features that are present (or absent) from the superior submission are extracted and displayed to students as features they may want to add (or remove) from their submission. Hints can be provided until the student reaches the best ABC score in the corpus. Noted software engineer Michael Feathers says of clean code that "There is nothing obvious that you can do to make it better" [5].

His definition is operationalized by our approach, in which students improve their code style until reaching an optimal threshold determined by examining a corpus of submissions assumed to contain at least one "ideal" exemplar. AutoStyle supports students in improving their code incrementally, by pointing them to idioms and approaches that make their code slightly better. Students are not immediately pointed to the best style solution, since the differences between a poor solution and and the best one may be hard to understand in one jump. Figure 1 shows hints provided to an AutoStyle student.

EXPERIMENT

This study examined the effect of AutoStyle in the context of a large introductory CS course. On two selected homework problems, after writing a functional solution, students could work on improving their code's style for extra credit. Students who wanted the extra credit were invited to participate in the study (unpaid). Only study participants were given AutoStyle. Students did untutored problems as assessments.

Class Context and Materials

330 students were enrolled in the summer offering of the on-campus introductory CS course. This course, geared toward rising sophomores, is intended to be students' first CS class and is required for the major. Style is a component of this course, and students' large programming projects are hand-graded for style. Of the style features that determined students' style grades in the course, some overlap with measures that ABC captures (e.g., avoiding unnecessary function calls and variable assignments, avoiding code that is never called) while others are distinct (e.g., meaningful variable names, clear and useful comments). ABC was not used directly to determine style grades in the course.

In study sessions 1 and 2, students revised a completed homework problem to improve its style (`AddUp` and `Permute`). Instructors did not grade those problems for style. We selected these two problems because past students' answers had a wide range of style quality, suggesting both that students may benefit from style support and AutoStyle could construct chains to provide it automatically. For `AddUp` and `Permute`, we used approximately 500 historical submissions to set up the clustering and hints in the AutoStyle tutor. For sessions 1 and 2, students worked with a tutor to improve their style on their homework problems, and then independently did experimenter-created tasks that drew on key idioms and concepts targeted in the homework (`Letters` and `CountAnagrams`). For study session 3, students only did an untutored, experimenter-created coding task, which drew on functions from students' most recent homework. All of the problems in the study can be solved in 4-10 lines of Python. The tutored homework problems and independent coding challenges were:

(1) Session 1 - with tutoring. `AddUp:` Given an integer `n` and a list of integers `lst`, return true if there exists two unique elements in `lst` that add up to `n`.

(2) Session 1 - independent. `Letters:` Given a list of words `lst`, return a Python set of letters that are common amongst all words in `lst`.

(3) Session 2 - with tutoring. `Permute:` Given a list of unique integers `lst`, implement a Python generator that returns all possible permutations of `lst`.

(4) Session 2 - independent. `CountAnagrams:` Given a word `word` as well as a list of valid words `word_list`, return the number of anagrams for `word` that are also in `word_list`.

(5) Session 3 - assessment. `MaxDepth:` Write a function that takes a deep list, and determines its maximum depth. You may use any built-in python functions.

We chose these problems to examine AutoStyle in a natural setting with normal coursework. While we designed the tasks to be suitable for students' prior knowledge, the course had not taught students to use the target functions to minimize ABC score in their code. In these problems and throughout the course, instructors asked students to write their own logic rather than using built-in functions. While this gives students practice coding, it does not mimic professional practices.

The multiple-choice questions presented a programming problem and 6 or 7 solutions. Part one of each question asked students to select the correct solution with the best style (Figure 2). To see if students were overlooking good solutions because they mistakenly thought they were not functional, part two asked students to select all functionally correct solutions. To avoid ordering effects, the answer order was randomized at pre-test and post-test for each student.

Participants

200 students participated in at least one study activity. The analysis below only includes the 103 students who participated in study activities for sessions 1 and 2. 99 of those students also participated in session 3. Students were randomly assigned to three groups: (1) get AutoStyle for the first assignment only, (2) for both assignments, or (3) for none of the assignments. Students' random assignment into study conditions did not affect the instruction they received in class. Each study session was limited to a two-hour block (see Method), and students did not receive class instruction during the study sessions. Therefore, it is unlikely that any differences between conditions across the study, and especially within a single study session, were caused by the course instruction.

Method

For sessions 1 and 2, students began by writing a working solution to their homework problem and submitting it to the course auto-grader. This solution had to pass a set of instructor-written test cases. If the solution was correct, the student could open the extra-credit interface from the auto-grader and work on improving the code's style. At this point, students were invited to participate in the study.

Consenting students did a pretest of two-part multiple-choice questions (two questions in session 1 and and one in session 2). Then students worked with their assigned style tutor (AutoStyle or ABC score only) until they reached a target ABC score (9 or below; lower scores are better), or until 45 minutes elapsed. We used 9 as the score threshold after examining prior submissions. Those submissions showed acceptable

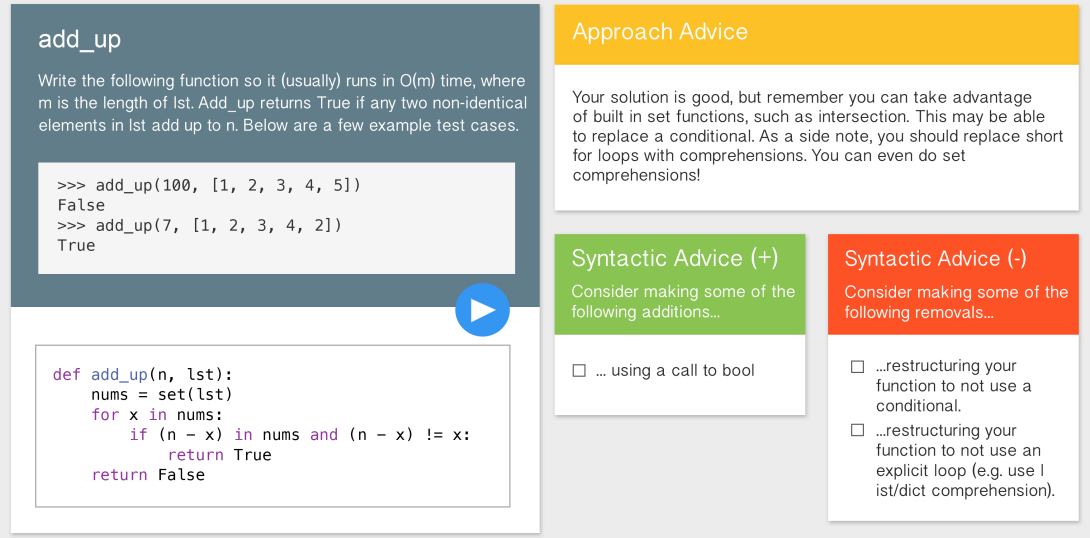

Figure 1. A student's submission and AutoStyle hints. Students in both conditions are also shown their ABC score (not pictured here).

style, with the best ones scoring around 2. Students had unlimited submissions. After each functionally correct submission, students were shown their ABC score. If a submission was not functionally correct, students were shown the expected and received results of experimenter-written test cases.

After working with the style tutor, students did a post-test (with the same multiple-choice questions as the pre-test). Finally, students were given a coding challenge, where they were asked to solve a new programming problem, from scratch, without hints or ABC score feedback. Students had an unlimited submissions, and worked until their program passed all experimenter-written test cases, or until 2 hours had elapsed for the whole experiment session. The third study assignment was a cumulative assessment with multiple-choice questions (one each from the previous assignments and one novel) and an independent coding challenge.

RESULTS

103 students participated in sessions 1 and 2 (respectively, 72 AutoStyle and 31 Control, and 49 AutoStyle and 54 Control). One AutoStyle student and three Control students each from session 1 and 2 did not do session 3.

Coding Style Improved

On each problem, a few students (≤ 6) did not submit any functional solutions. Only students who submitted at least one functional solution are included in the analyses of coding style. Students improved their style when they worked with either tutor (AutoStyle or ABC score only), but did not improve their style on the untutored coding challenges. Most students submitted at least two unique, functional solutions for the tutored problems, indicating that the feedback prompted revisions. By their last submissions, most students surpassed the ABC threshold for the tutored problems (`AddUp`: Control - 71%, AutoStyle - 74%; `Permute`: Control - 75%, AutoStyle - 78%). Means for ABC scores improved from first to last attempts for the tutored problems (from 11.7 to 8.5 for `AddUp` and from 11.3 to 8.7 for `Permute`). For the untutored problems `Letters` and `CountAnagrams`, few students submitted more than one functional solution, and scores did not improve on those problems. For the untutored problem `MaxDepth`, 7 AutoStyle students and one control student improved their code through revision, resulting in a slight improvement in scores for the AutoStyle condition but not the control. Table 1 shows how many students in each condition completed each problem and revised their work, along with initial and final style scores.

Although students' style scores do not satisfy the Shapiro-Wilk test for normality, we proceeded with ANOVAs and t-tests under the assumptions of the Central Limit Theorem since there were more than 30 students in each condition. Within-condition paired t-tests indicate significant differences in initial and final style scores on the tutored problems for both conditions, indicating that both tutors helped students improve through revision. The only untutored problem that showed a significant difference in initial and final style score was `MaxDepth`, and only for the AutoStyle condition, indicating that AutoStyle students improved their score through revision while working independently. However, while the result is statistically significant, the improvement was small (see table 1 for means, p-values, and t-values). To compare improvement on tutored vs. untutored problems, we ran separate repeated measures ANOVAs for ABC scores in study sessions 1 and 2. These ANOVAs followed the pattern suggested by the means: students in both conditions improved on the tutored problems but not the untutored ones, and there were no differences in improvement by condition. The repeated measures ANOVAs were run on students' ABC scores by submission time (first/last), if the problem was tutored (true/-false), and by assigned condition (AutoStyle/Control). These analyses were done with the 101 students who submitted functionally correct code for both `AddUp` and `Letters`, and with the 97 students who submitted functionally correct code for both `Permutation` and `CountAnagrams`. Submission time

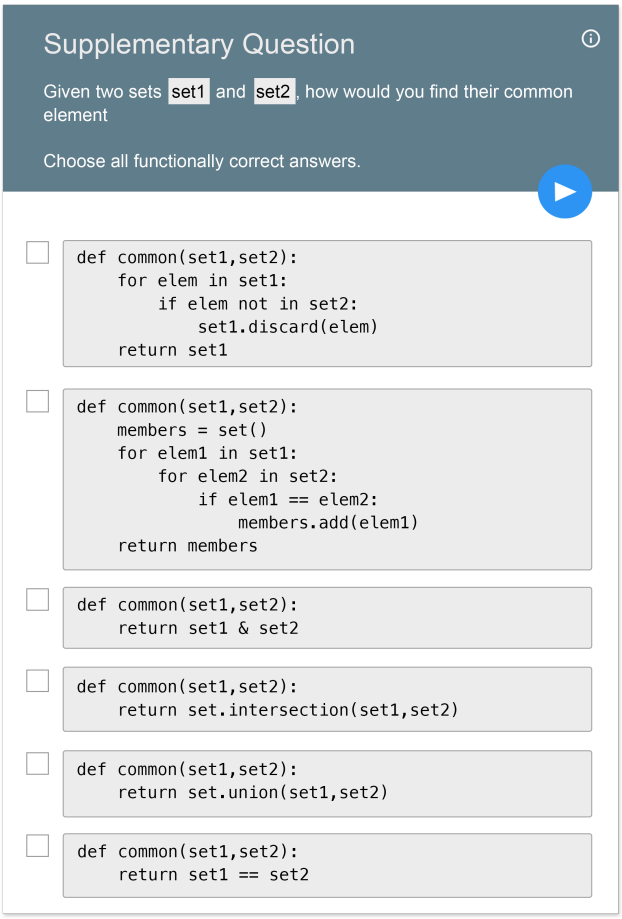

Figure 2. Answer options for a multiple choice question: Given two sets, set1 and set2, how would you find their common element? Students were asked to select the best style solution and then identify all functionally correct solutions. The middle two choices are the best stylistically, and the last two are not functionally correct.

($F(1) = 67.99; 41.9$), tutoring ($F(1) = 18.3; 209$), and their interaction ($F(1) = 69.1; 40.9$) were significant in both analyses (Pillai's Trace: $p < .001$ for all), while condition was not significant as a main effect ($F(1) = 1.1; .22, p = .3; .6$) or in interactions with submission time ($F(1) = .39; .31, p = .5; .5$; values are given for session 1 and 2, respectively). To compare condition differences in improvement on the untutored problem `MaxDepth`, we ran an ANOVA on data from the 98 students who completed the problem. We ran the ANOVA on final ABC score for `MaxDepth`, with initial ABC score as a covariate. Condition was not significant ($F(1) = 3.65$, $p = .059$). The t-test for the 70 students who had used AutoStyle was driven by 10 students who made two attempts on `MaxDepth` instead of one. Seven of those students improved their score, and three maintained their score. Of the 28 control students, five made more than one attempt. One student improved their score, one maintained it, and three worsened their score (note, students did not get ABC score as feedback on the coding challenges).

Students were allowed unlimited code submissions. Submissions that were not functionally correct were not given an ABC score. Further, some students submitted the same exact code several times in a row. Therefore, in examining students' attempts to improve their code, we consider an attempt to be a functional solution that is not a duplicate of the immediately-preceding submission. Across conditions, students averaged about 4 attempts when they worked with a style tutor (4.7 and 4.1 for `AddUp` and `Permutation`, respectively). 90% of AutoStyle students and 87% of control students made two or more attempts on `AddUp`. 67% of AutoStyle students and 64% of control students made two or more attempts on `Permute` (see table 1 for raw numbers). Most of the remaining students met the style threshold on their first attempts. Only one student in each condition (for `AddUp`) and one control student (for `Permute`) made only one attempt without meeting the style threshold. Repeating the ANOVAs above without students who met the style threshold on their first attempt does not change the significance levels of the results. Students averaged 1 attempt on the coding challenge, when they did not receive any feedback or have a style threshold (1.1, 1,1, and 1.2 for `Letters`, `CountAnagrams`, and `MaxDepth`). Fewer than 20% of students made multiple attempts on these problems.

Identification of Good Style Improved

Students improved in their identification of the best style solution from a given set. Multiple choice questions on each assignment were given before and after the tutored coding problem. These questions presented a coding problem and sample solutions, and asked which solution exhibited the best style. Answers were scored 1 if correct and 0 if incorrect or blank. Session 1 included two of these questions and session 2 had one. On AddUp question 2 and on the question for Permute, AutoStyle students improved more than the control, with no significant difference in improvement on AddUp question 1 (see table 2 for mean scores). Three separate logistic regressions on the post-test scores, with tutoring condition and corresponding pre-test score as factors, indicate significant effects for pre-test score (all $p < .015$), with significant effects for condition on question 2 for `AddUp` ($p = .04$) and on the question for `Permute` ($p = .03$), in favor of AutoStyle. Nagelkerke's pseudo R^2 for the three regressions are .15, .38 and .16, respectively. These analyses were done on all 103 students who participated in the study. Additional logistic regressions were run on scores from session 3, with corresponding pre-test scores from session 1 or 2 and exposure to AutoStyle as factors. Condition was not significant in any of those analyses, suggesting that AutoStyle outperformed the control on immediate learning but not longer term retention.

While students improved on their identification of the best coding style, they did not improve on their identification of which code blocks were functionally correct. After asking which solution exhibited the best style, a follow-up question asked which ones correctly solved the problem. For each of the 6 code samples, students got 1 point if they correctly identified it as functional or not. Students' improvement on these items were not significant from pre- to post-test or significantly different by condition, as indicated by a repeated measures ANOVA on the three questions (`AddUp` 1 and 2, and `Permute`)

Task	AutoStyle *N*	Mean Attempts	Initial style	Final style	Paired t-test	Control *N*	Mean Attempts	Initial style	Final style	Paired t-test
`AddUp` (tutored)	71 (64)	3.6 *(2.4)*	11.9 *(6.4)*	8.5 *(5.7)*	$p < .001$	31 (27)	7.2 *(5.9)*	11.4 *(4.3)*	8.4 *(4.1)*	$p < .001$
`Letters` (no tutor)	70 (2)	1.0 *(0.17)*	15.4 *(9.2)*	15.4 *(9.2)*	same mean	31 (6)	1.3 *(0.63)*	13.5 *(4.5)*	13.6 *(4.4)*	$p = .432$
`Permute` (tutored)	49 (33)	5.0 *(6.07)*	11.1 *(4.6)*	8.6 *(2.70)*	$p < .001$	53 (34)	3.4 *(4.26)*	11.5 *(4.7)*	8.7 *(3.1)*	$p < .001$
`CountAnagrams` (no tutor)	48 (2)	1.0 *(0.20)*	22.5 *(7.4)*	22.5 *(7.4)*	$p = .159$	49 (4)	1.1 *(0.27)*	21.4 *(7.4)*	21.4 *(7.5)*	$p = .322$
`MaxDepth` (no tutor)	70 (10)	1.1 *(0.35)*	18.3 *(5.34)*	18.1 *(5.31)*	$p = .017$	28 (5)	1.3 *(0.59)*	17.9 *(4.7)*	18.0 *(4.8)*	$p = .395$

Table 1. For each task, the number of students who submitted a functional solution (and who submitted at least two attempts). Means (and standard deviations) for number of attempts, and for initial and final ABC scores. Students are divided by condition. Paired t-tests compare the first and last ABC scores of each problem, by condition, to determine if the improvement is significant. t-values for each test are (left to right): 7.9, 4.7, NA, -.80, 4.4, 4.9, -1.4, 1.0, 2.4, -.86. In session 3 (`MaxDepth`, untutored), students who had used AutoStyle previously improved their score with revision.

Multiple Choice	Group	Pre-test	Post-test	Session 3	Condition (pre-post)
AddUp Q1	AS	.86	.92	NA	$p = .7$
	Control	.71	.90		
AddUp Q2	AS	.28	.46	.44	$p = .04$
	Control	.26	.26	.36	
Permute	AS	.18	.31	.31	$p = .03$
	Control	.17	.13	.24	

Table 2. Multiple-choice questions at pre- and post-test asked students which code example was stylistically best (two questions for session 1, one for session 2). Two of these questions were repeated at session 3. Percent correct for each condition shows more improvement for AutoStyle. Logistic regressions on post-test scores with pre-test as a covariate show that the greater improvement in the AutoStyle condition for `AddUp` Q2 and for `Permute` are significant. This table includes all participants.

by test time (pre or post), with condition as a fixed factor ($p > .05$ for test time, test time * question, condition, and condition * test time; n = 103).

STUDENTS' INTERACTIONS WITH THE TUTORS

AutoStyle is intended to help students by pointing them to new style idioms and approaches, which the student ideally implements. Case studies illustrate how students actually interacted with AutoStyle and the control tutor. Our case studies come from three categories: AutoStyle students who improved their code and met the style threshold on their last attempt; AutoStyle students who did not meet the style threshold despite multiple attempts; and Control students who improved their code and met the style threshold on their last attempt. We selected one student from each group, choosing cases where we felt most confident interpreting students' intentions and where students' actions highlighted shortcomings of AutoStyle. These case studies are from `AddUp` (session 1).

Implementation Is Hard. Case Study: Diligent Student

The Diligent Student begins with an acceptable solution: using a for-loop over all elements in `lst`, which yields an ABC score of 11.6 (slightly higher than the goal, 9). AutoStyle suggests using the Python built-in function `list`. This change would make the code better incrementally, but would not immediately lead to the best solution. The student tries to implement the suggestion, but makes a very simple typo. The student has a variable `lst`, and calls `lst` instead of `list`. This introduces a bug, and the code does not compile. It takes the student six submissions to fix this bug (see figure 3). From the code alone, it is not clear if the student misunderstood the hint, or if the student made a typo and struggled to identify it. While no students in the control condition made this exact error, 5 other students in the AutoStyle condition did (8.5% of the AutoStyle condition).

In many ways, this student exemplifies the ideal user of the system. After 45 minutes of working with the tutor, students could still get extra credit even if their code did not meet the threshold. Diligent Student continues to struggle with implementing some hints, working beyond the required 45 minutes. Ultimately, Diligent Student implements hints designed for strong clusters, and achieves an ABC score of 5.4, one of the top style solutions for the homework assignment. This case study illustrates that students cannot just write correct code and then move on to improving its style - even diligent students may regress to non-functional code in the process. For less-diligent students, the bugs they introduce may become insurmountable obstacles.

This student was not the only one who struggled to implement hints. On average, after an AutoStyle student receives a hint, it takes four tries for the student to implement the change in a functionally correct manner. The lack of hints for non-functional code is a weakness of AutoStyle.

Large Changes Are Hard. Case Study: Confused Student

AutoStyle requires students to start with working code. However, all working code is not equal. Confused Student started with a ABC score of 55.6, a big difference from Diligent Student's 11.6. Confused Student solved the homework problem by breaking the problem down into many (and sometimes redundant) edge cases instead of crafting one holistic solution. AutoStyle does not provide hints that are specific to that starting point, so Confused Student got the same hints as Diligent Student. Confused Student did not successfully implement any of AutoStyle's hints. Many attempts resulted in non-functional code, and the few submissions that were functionally correct actually made the ABC score worse.

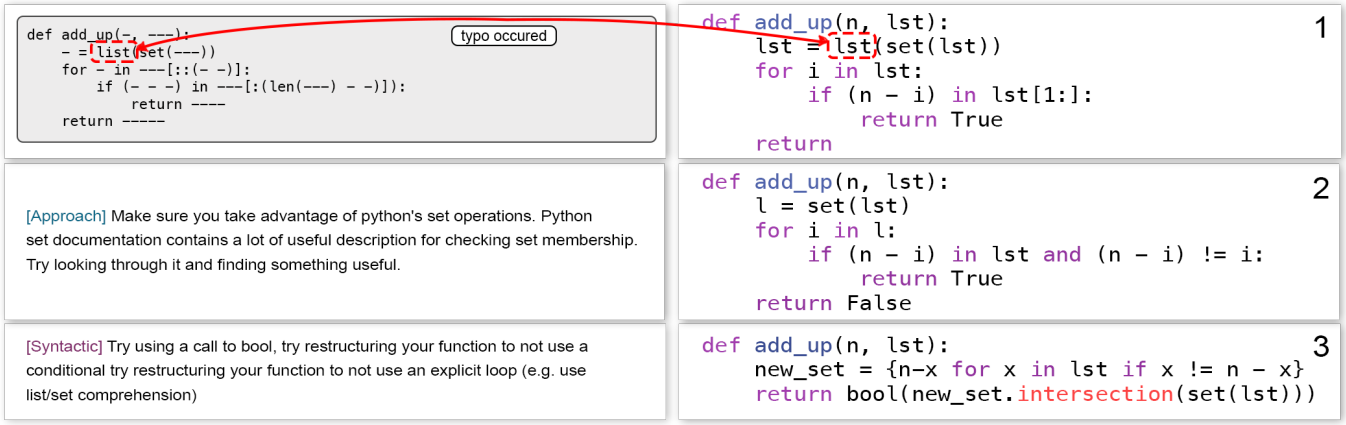

Figure 3. Diligent student gets a skeleton hint suggesting `list` (top left) and makes a typo in trying to implement it (top right). After correcting the typo, the student gets a hint to use `set`, which is implemented correctly (middle row). The student gets further hints throughout the session (bottom left) ending with a stylistically good solution (bottom right).

AutoStyle leads students down an incremental path of code improvement: suggestions point students to solutions that are a little bit better. The assumption underlying this approach is that students will not be able to implement drastic changes in one revision. Confused Student's interaction support this notion. In total, 13 students started `AddUp` with ABC scores above 20. Eight students exhausted the required 45 minutes and still had a score above 20 at the end. Two students ended between 20 and 9, and only one met the target threshold. Two students had ABC scores above 20 for the majority of their session, but achieved the best ABC score on their last submission (likely gaming the system).

The general lack of improvement for students who started with very poor code suggests that AutoStyle's current hints may only be advantageous for students who demonstrate a complete understanding of the preexisting assignment. Students may struggle when AutoStyle asks them to take a perspective that is radically different from the student's current understanding. Since starting with working code can inhibit large-scale changes [3], some students may benefit from approach hints before they start coding.

Self Assessment Is Hard. Case Study: Myopic Student

Myopic Student is a control student, and only received ABC score as feedback. Myopic Student did improve their code style, but demonstrated difficulty in assessing where and how to do so. The student's first submission is good: it solves the assignment using a for-loop and a call to Python's set operations, with an ABC score of 9.84. Myopic Student's initial submission has an identical control flow to Diligent Student's submission code block 2 in Figure 3. While Diligent Student has AutoStyle's suggestions to follow, Myopic Student demonstrates a lack of direction in attempts to improve coding style. The student makes a series of simple transformations by removing whitespace and adding redundant test cases in the for-loop body, which only serves to vary the student's ABC score between 9.84 and 10.92. At some points it seems like this student is just testing out what will make the ABC score go up or down. Shorter code is often more concise and our style metric rewarded students for using fewer lines. However, this measure is vulnerable to gaming, and several students "improved" their style by cramming the whole solution into one line. This brevity reward will not be used in future work. Myopic Student finds that shorter submissions (e.g., without blank lines) result in better ABC score, and then tries to shorten the code further re-ordering the control flow. However, Myopic Student never attempts to implement an operation that wasn't contained in the original submission. Myopic Student's submissions illustrate the difficulty in assessing what improvements can be made (step 1in our pedagogical framework).

```
def add_up(n, lst):
    for num in lst:
        if ((n - num) in lst and (n - num) != num):
            return True
    return False
```

Figure 4. Myopic Student's final submission. The Python built-in function `list` does not yield the best style solution.

Although this student started with AutoStyle's suggested function, `set`, this student replaces it with a call to `list`. A solution using `list` allows the student to achieve an ABC score better than the threshold, but does not achieve the best solution (see figure 4). As students in the introductory programming course are taught how to use `list` before `set`, the student seems to fall back on previous knowledge instead of discovering new information on how to use `set` to achieve a more stylistic solution. This illustrates a student attempting to do step 2 (Information Retrieval) from prior knowledge, without the foresight of how the change will affect the code's style.

DISCUSSION

AutoStyle helped students write and recognize beautiful code. Students in both conditions improved their style while working with a tutor. However, overall, students' submissions on the coding challenges did not demonstrate that AutoStyle promoted better independent work. While students may not have

learned enough from the tutor to make those revisions, another explanation is that students were not well-motivated to improve their style. Students received extra credit for writing any functional solution to the coding challenge, and there was no target ABC score.

Although AutoStyle and the ABC-only tutor produced similar style improvements, students' paths through the two tutors were different. Considering only the students who met the style threshold for `AddUp`, AutoStyle students were much more likely to use the target function `set` (11/22 with the control, 42/51 with AutoStyle). However, it was not the case that students were unfamiliar with `set` before working with AutoStyle. Almost all students used `set` in their independent coding challenge (where its relevance was more apparent), and students correctly identified functional solutions with `set` in the multiple-choice questions. Rather, it seems that without AutoStyle, proportionally more students did not recognize that `set` would be a good style choice for the `AddUp` problem. This underscores why the Information Retrieval step in our pedagogical framework applies to a student's prior knowledge as well as to new information.

Further, ABC score alone was not as helpful in teaching students to recognize good style. While students appeared to be at ceiling for the best-style multiple-choice question `AddUp` 1, the majority of students were not correct on `AddUp` 2 or Permute, even at post-test. Identifying good style is not trivial for novice students, likely because their conceptions of good style do not match those of experts. Students in the control condition, like Myopic Student, may have improved their ABC score through guess-and-check without uncovering the meaningful features that affect style. The easiest feature to uncover is superficial: shorter is usually better. Since AutoStyle provided text hints, students did not need to guess at which features mattered for style. This may explain why AutoStyle students improved more than the control on two of the three multiple-choice questions that asked students to select the best style solution. Recognizing good style was not cited as a goal of the original AutoStyle work [4, 3], illustrating how our pedagogical framework can improve the design of style assessments. Overall, students did not improve on their recognition of which options were functional, indicating that improvement on selecting the best style was driven by students' new ideas about style, not correctness.

Our data suggets that each step in our framework is indeed distinct. Many novices cannot easily identify good style on multiple-choice questions, or retrieve the information necessary to improve their style, even when that information resides in their own prior knowledge (e.g., Myopic Student). Finally, even when students know which function to use, implementation is not trivial (e.g., Diligent Student). These findings show initial validity for our framework, and suggest that style tutors should support all three steps.

Recommendations for CS Educators

If nothing else, show the student their style score while they're coding and provide a target score based on a reference solution. The quantitative results showed that students improved their code with ABC score alone. This suggests that simply making students aware that style can be measured, and telling them when there is room for improvement, seems to drive them to improve it. However, students are unlikely to learn what good style is with ABC feedback alone. Showing students example code of similar lengths and explaining why one has better style may dissuade students from simply thinking that shorter is better.

Future Work: Improving AutoStyle

Process measure and case studies illustrate where AutoStyle could provide more supportive feedback. First, students need style support even when their code is not functional. There is no clear distinction between getting code to work and improving the code's style. For the Diligent Student, the Confused Student, and many others, attempts to improve style introduced bugs. These bugs also show the difficulty of Implementation (step 3 in our pedagogical framework). Telling students what function to use is not always enough. In large-enrollment courses, students may make the exact same mistake when trying to implement a change (e.g., 8.5% of the AutoStyle condition tried to call a function with the typo `lst` instead of the name `list`). AutoStyle could provide specific feedback for common mistakes.

Second, when students start off with working but stylistically horrible code (like Confused Student), AutoStyle may be more effective if it tells the student to start from the beginning. Including one instructor-written approach hint for these cases may help students who otherwise wouldn't be able to start down the style-improvement path.

Third, while AutoStyle points students to documentation, it does not help them interpret the documentation. AutoStyle may be more effective if it scaffolds students in how to use online resources - this may also reduce the instance and severity of implementation errors. AutoStyle could also become more adaptive: if the student receives a hint but seems unable to successfully apply it to their own code upon resubmission, the student may need tutoring in interpreting the language documentation or in modifying an example in the documentation to apply it to their own code.

Future Work: Targeted Assessments

This study did not replicate the striking results from [4], where 70% of AutoStyle students vs. 13% of the control reached the best style solution. However, our non-replication was not because AutoStyle students did poorly, but rather because the control students did well. One explanation may be the differences between the particular problems and style thresholds in the two studies. In [4], students needed to use the AutoStyle-recommended syntax to get under the style threshold. In `AddUp`, students could reach the style threshold without using the syntax that AutoStyle recommended (though it was necessary for the best possible solution). Selecting a lower threshold may have resulted in stronger relative benefits for AutoStyle in our study. For `Permute`, the main style recommendations related to approach and control flow, not syntax. This made it more difficult to automatically detect the effect of AutoStyle in students' code. Additionally, AutoStyle's current approach hints may not be as effective as the syntax hints.

Likewise, we did not replicate the results from [3], where students demonstrated style improvements independently. The independent problems in [3] required the same syntax, used in the same way, as the corresponding tutored AutoStyle problem. This tight connection between the tutored problem and the independent problem was not present in the current study. Future work will develop a range of assessments that allow students to demonstrate both near and far transfer.

CONCLUSIONS

On course homework assignments, both AutoStyle and ABC feedback helped students improve their style. AutoStyle additionally helped students recognize code with good style. This *in situ* study shows AutoStyle's effectiveness in a large-enrollment course. Results from the style-recognition questions and from logged student interactions provide initial validation for our 3-step pedagogical framework by showing that the three steps are distinct, and students need support in each. Although AutoStyle was not designed to promote recognition of good style, students did improve on this measure, a key component of Assessment (step 1 in our framework). While an important aspect of AutoStyle is that it directs students to code documentation and online resources, this study showed the importance of helping students incorporate their own prior knowledge (part of Information Retrieval, step 2). Finally, case studies and log data show that students still struggle with Implementation (step 3) even when step 2 is complete.

Acknowledgements

This work was supported by an IBM Faculty Award, a gift from Google Inc, and the National Science Foundation (Grant No. DRL-1418423 and INT-1451604). Opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

1. Hannah Blau and J. Eliot B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM, New York, NY, USA, 15–20. DOI:http://dx.doi.org/10.1145/2729094.2742622
2. Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison-Wesley Professional.
3. Antares Chen, Eliane Wiese, HeZheng Yin, Rohan Choudhury, and Armando Fox. 2016. Preliminary evidence for learning good coding style with AutoStyle. In *Third Symposium on Learning With MOOCs (LWMOOC III)*. Philadelphia, PA.
4. Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. 2016. Scale-Driven Automatic Hint Generation for Coding Style. In *13th International Conference on Intelligent Tutoring Systems (ITS 2016)*. Zagreb, Croatia.
5. Michael Feathers. 2004. *Working Effectively with Legacy Code*. Prentice Hall.
6. J. Fitzpatrick. 2000. Applying the ABC Metric to C, C++, and Java. In *More C++ Gems*. Cambridge University Press, New York, NY, 245–264.
7. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
8. Robert L. Glass. 2002. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional.
9. Elena L. Glassman, Lyla Fischer, Jeremey Scott, and Robert C. Miller. 2015. Foobaz: Variable Name Feedback for Student Code at Scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software I& Technology*. ACM, Charlotte, NC, 609–617.
10. Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 86–93. DOI: http://dx.doi.org/10.1145/1930464.1930480
11. S. Johnson. 1977. *Lint, a C program checker*. Technical Report 65. Bell Labs.
12. D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.
13. C. E. Kulkarni, M. S. Bernstein, and S. R. Klemmer. 2015. PeerStudio: Rapid Peer Feedback Emphasizes Revision and Improves Performance. In *Proceedings of the 2nd ACM Conference on Learning@Scale*. ACM, New York, NY, 75–84.
14. Jin-Su Lim, Jeong-Hoon Ji, Yun-Jung Lee, and Gyun Woo. 2011. Style Avatar: A Visualization System for Teaching C Coding Style. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*. ACM, New York, NY, USA, 1210–1211. DOI: http://dx.doi.org/10.1145/1982185.1982451
15. M Mäntylä. 2005. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement.. In *ISESE*. 134–138. http://lib.tkk.fi/Diss/2009/isbn9789512298570/article3.pdf
16. M Mäntylä and Casper Lassenius. 2006. Drivers for software refactoring decisions. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)* (2006), 297–306. http://dl.acm.org/citation.cfm?id=1159778
17. Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
18. K McMaster, S Sambasivam, and Stuart Wolthuis. 2013. Teaching Programming Style with Ugly Code. In *Information Systems Educators Conference*. San Antonio, TX. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.400.9411&rep=rep1&type=pdf
19. Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. DOI: http://dx.doi.org/10.1109/TSE.2004.1265817