Auglets: Intelligent Tutors for Learning Good Coding Practices by Solving Refactoring Problems

Amruth N. Kumar
Computer Science
Ramapo College of New Jersey
Mahwah, NJ, USA
amruth@ramapo.edu

ABSTRACT

Code quality is of universal concern among educators. Refactoring code, i.e., revising the structure of a program without changing its behavior is one approach for improving code quality. Numerous software tools have been created to help students refactor the code they write. Only a few software tutors have been reported in literature that help students proactively learn code quality by solving refactoring problems. But they suffer false positive and false negative grading issues because they allow freehand coding. We investigated whether refactoring tutors that do not allow freehand coding could be used to help students learn about non-trivial anti-patterns. We developed and deployed two software tutors for refactoring problems that are based on the principle of "refactoring without rewriting code", and cover a subset of refactoring problems that can be solved using only deletion, duplication, reordering and token-wise editing of lines of code. We investigated whether students needed to learn the anti-patterns covered by the tutors and whether they benefited from using the tutors. In this experience report, we start by describing the tutors – the list of refactoring concepts covered, the user interface, grading, feedback and usage. We report our experience using the tutors over three semesters, which confirmed that both introductory and advanced students needed and benefited from using the tutors despite the limitations of the tutors' coverage. We reflect on what worked and what did not. The tutors currently cover C++, Java and C#. They are available for free for educational use on the web at auglets.org.

CCS CONCEPTS

• **Social and professional topics~CS1, Computational Thinking**

KEYWORDS

Problem-Solving tutor, Code quality, Refactoring, Anti-patterns, C++, Java.

ACM Reference format:

Amruth N. Kumar. 2024. Auglets: Intelligent Tutors for Learning Good Coding Practices by Solving Refactoring Problems. In *n Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1 (SIGCSE Virtual 2024), December 5–8, 2024, Virtual Event, NC, USA. ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3649165.3690119

1 Introduction

While learning to code is hard, learning to write good code is imperative [24]: the most significant cost incurred during the lifecycle of software is for maintenance and enhancement [11] which are seriously hindered by poorly written code. Good coding style leads to good software design, which in turn results in faster software development [7]. Educators are of the consensus that code quality must be more thoroughly discussed in courses [3]. Yet, they have found that it is hard to teach good coding style [12, 16, 41], and frequently advocating the use of good style in class is insufficient in and of itself [21].

While learning to code, novice students often resort to anti-patterns [4, 17, 38]: common responses to recurring problems that are counterproductive because they make the code hard to read, modify or extend. In most cases, the code is correct. The code will pass all the test cases specified by the instructor or used by automated project submission software. So, students have little or no incentive to revise the code to rid it of anti-patterns or learn good coding practices.

Several approaches have been tried to help students learn about code quality. Educators typically mention anti-patterns in class and/or recommend trade books (e.g., [7, 10, 27, 39]) as supplementary reading material. But these do not facilitate active learning. Instructors often read and critique student programs and encourage students to rewrite their programs. But this approach does not scale up to larger classes and longer projects. Some educators have proposed covering a series of lessons in class on refactoring (e.g., [34, 35, 50]). Others have found that live coding in class helps students learn anti-patterns [31, 33] and using a lab- based resource helps them learn anti-patterns pertaining to `if- else` statements [23]. These approaches place demands on class time and are therefore, resource-intensive.

Educators have proposed peer reviews in various forms: peer code reviews (PCR) [36, 40], pedagogical code reviews [12, 13] and code reviews [37] – wherein a team of students, often led by a trained moderator, reviews each other's code, and discusses and logs code quality. While these approaches have been found to be effective, they are resource-intensive. So, researchers have been developing online software to support code reviews (e.g., [11, 14, 40, 52]).

Researchers have used unsupervised machine learning to provide feedback to students on the quality of code in their program submissions. In this approach (e.g., Codex [8], CodeWebs [28], and AutoStyle [5]), code exemplars are extracted for a problem from a corpus of prior submissions, often based on minimizing Assignment-Branch-Conditional (ABC) metric [9]. The abstract syntax tree (AST) of a student's submission is compared against that of the exemplar(s) to provide style-related feedback. These approaches have been found to help students learn anti-patterns (e.g., [42]). A drawback with them is that they need a large corpus of prior solutions to be able to generate appropriate feedback. Another is that they are diagnostic, not instructional in nature, i.e., they provide style feedback while students write code, but are not designed to help students systematically learn anti-patterns.

Several style checking tools are available such as lint [15, 22], pylint (pylint.org), PMD (pmd.github.io), checkstyle, (checkstyle.org), style50 [53], Sprinter [54] and Style++ [1] to name a few. These tools deal with anti-patterns that are largely typographical (e.g., indentation, commenting, etc.) or syntactic (e.g., naming convention).

The tools developed to help students with semantic anti-patterns [32] – patterns of code that relate to the structure of the program – include Java Critiquer [30], FrenchPress [2], WebTA [38], and JDeodorant [25]. All are for Java. Some have been integrated into IDEs such as Eclipse (e.g., [49]). Similarly, Litterbox [26] has been developed for Scratch. These tools are again, diagnostic and reactive, not proactive and instructional in nature.

Prompt patterns can be used with Large Language Models (LLMs) to refactor code (e.g., [43, 51]). Similar to the tools developed to analyze student code, they are diagnostic, not instructional in nature. They provide the correct solution instead of helping students learn to solve refactoring problems. With the correct sequence of prompts, they can be coaxed to explain the correct solution. But such prompt engineering is ad hoc and not typically within the skill set of learners.

There is a need for instructional tools to help students *proactively* learn good coding style (in addition to reactively fixing anti-patterns in their code), and in particular, semantic anti-patterns. Software tutors are a scalable active-learning instructional tool - students can use them on their own time, at their own pace, and as often as they please. As per literature review and systematic literature surveys [6,18,29], software refactoring tutors have been reported for Java [19, 20, 44], C# [48] and Python [45]. All these tutors allow freehand coding and are therefore susceptible to false positive and false negative grading.

We report the development and use of intelligent tutors to help students proactively learn about anti-patterns - they do not allow freehand coding. In this experience report, we present how, even without allowing freehand coding, the tutors cover non-trivial anti-patterns that students need to learn. Our experience shows that using the tutors helps students learn about anti-patterns.

1.1 Novelty of our Approach

In contrast to earlier refactoring tutors [19, 20, 44, 45, 48], our intelligent tutors target semantic anti-patterns and allow a limited set of editing operations *not* including freehand coding. Therefore, they do not suffer false positive and false negative grading. The tutors use source code comparison instead of unit tests [19, 20] to grade student submissions. The tutors can adapt to the needs of students by presenting problems on only the anti-patterns that they do not already know [10]. They currently cover C++, Java and C#.

We will present the design of the tutors, some of the anti-patterns covered by them, their user interface, typical experience of a student solving refactoring problems with the tutors, how the tutors can be used by students and instructors, formative data collected using the tutors over three semesters, and reflection on what worked and what did not.

2 The Design of the Tutors

2.1 Topics and Concepts

The tutors cover two topics: selection statements (`if-else` and `switch`) and loops. Currently, the tutor on selection statements covers the following refactoring concepts, all of which can be solved without any freehand coding:

1. S1: Factor out code common to both if- and else- clauses
2. S2: Delete empty `else`
3. S3: Remove redundant `if` statement from else clause when the condition of `if` statement is the negation of the condition of the `if-else` statement
4. S4: Combine two or more `if-else` statements that have the same condition
5. S5: Invert control structures to minimize duplicate code
6. S6: Do not nest `if-else` statements whose conditions are independent
7. S7: Factor out code common to multiple cases in a `switch` statement
8. S8: Combine cases with the same code in a `switch` statement

The tutor on loops covers the following refactoring concepts that can be solved without freehand coding:

1. L1: Don't rig initialization of the loop variable of a `while` loop so that the loop iterates at least once
2. L2: Move the code that should be executed after the loop out of the loop

3. L3: Move computations out of validation loop (loop meant to check that the input is valid, such as 1-12 only for month)
4. L4: Eliminate redundant post-loop `if` statement whose condition is the negation of the loop condition
5. L5: Instead of a `do-while` loop inside an `if` statement, use a `while` loop
6. L6: Validate one input per validation loop

All of these are non-trivial semantic anti-patterns [7, 46] commonly observed in the program submissions of both introductory and advanced students. *Therefore, the tutors can be beneficial even though they do not allow freehand coding or cover anti-patterns that require freehand coding.*

2.2 Refactoring Problems

The tutors contain 4-5 problems per refactoring concept. Each refactoring problem contains a complete program designed to illustrate a single anti-pattern. Students are instructed to refactor the program for a specific purpose such as better readability, reduced redundancy, or improved modifiability. Each problem is designed to have a single correct solution and can be solved without freehand coding.

A refactoring problem consists of the following components: 1) a statement of the problem for which the program was written; 2) the program to be refactored; 3) the refactored version of the program; 4) drill-down instructions to clarify the purpose of refactoring; and 5) metadata including a unique problem identifier, the anti-pattern illustrated by the problem (e.g., S2: "Delete empty `else`") and the programming languages to which the problem is applicable.

In order to deter plagiarism common to software tools [47], the program to be refactored and its refactored version are encoded as templates in Backus-Naur Form (BNF), with meta-variables for data types, literal constants, variable names, etc. The tutors generate each problem as a randomized instance of a template by replacing meta-variables in the template with specific data types, literal constants and variable names. Two problems generated from a template are similar, but not identical. So, no two students see identical programs and no student sees the same program twice.

2.3 User Interface

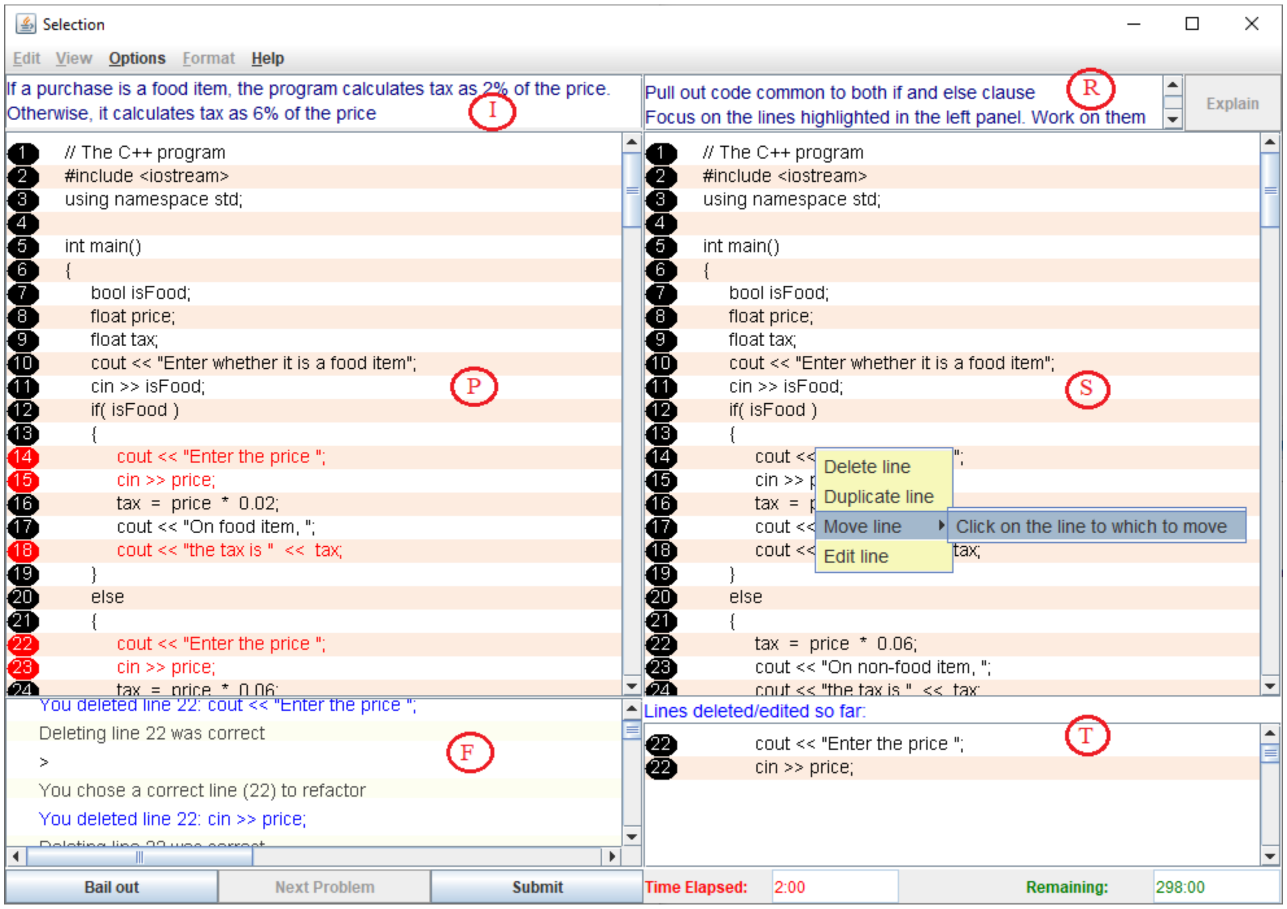
The layout of the tutors is shown in Figure 1. The layout consists of the following components:

- Instruction panel **I** where the problem statement is presented.
- Problem panel **P** where the program to be refactored is presented.
- Refactoring panel **R where** the purpose of refactoring is described. Clicking on the *Explain* button in this panel produces drill-down instructions at progressively greater levels of detail. In the final level of detail, the lines that should be refactored are highlighted in the problem panel **P** and students are asked to focus on them.
- Solution panel **S** where a copy of the program from the problem panel is shown - students refactor this copy while comparing it with the original code in problem panel.
- Feedback panel **F** where feedback is displayed after each refactoring operation and after submission of the solution. Students' refactoring operations are also summarized here.
- Trash panel **T** to which lines of code deleted in the solution panel **S** are moved. In addition, if a line of code is edited in the solution panel **S**, the original version of the line is copied to the trash panel **T**. Lines in the trash panel can be restored to the solution panel at any chosen location.
- *Submit* and *Bail out* buttons at the bottom left are enabled after students apply at least one refactoring operation to the program in the solution panel **S**. If a student bails out, the problem is marked as not attempted. If a student submits a solution, feedback is provided in panel **F** on whether the solution is correct or incorrect. In either case, the correct solution is displayed in the problem panel **P** with all the correctly refactored lines highlighted (not shown in Figure 1). After students click on *Submit* or *Bail out* button, *Next Problem* button is enabled so that students can advance to the next problem.
- Help menu at the top provides the option to get step-by-step instruction on using the tutor.
- Timer at the bottom right helps keep track of elapsed and remaining time when the tutors are used for timed assessments.

During a problem-solving session, students read the problem statement presented in panel **I** and the refactoring instructions in panel **R**. They refactor the program in panel **S** – they click on a line of code to reveal a menu of refactoring operations available for that line of code, as shown in Figure. 1 The refactoring operations currently provided by the tutors are:

- Delete a line – the deleted line is moved to the trash panel **T**. Students can undo a deletion by clicking on the line in the trash panel and selecting a line number in the solution panel **S** to which to restore it.
- Duplicate a line – a second copy appears within the solution panel **S** right after the original line.
- Move a line – students are asked to select the destination to which to move the line by clicking on a line in the solution panel **S**. This interface is shown in Figure 1.
- Edit a line – a dialog box is presented in which students can delete tokens in the line, such as punctuation characters, operators, variables, and literal constants. Figure 2 shows an example of the dialog box – students can click on any of the toggle buttons to delete/undelete the corresponding token from the line of code.

Limiting students to these four operations and no freehand coding brings home the point that refactoring is an exercise in reorganizing code rather than rewriting it. These four operations are sufficient to solve problems on the refactoring concepts S1 - S8 and L1 - L6 listed earlier.



If a purchase is a food item, the program calculates tax as 2% of the price.
Otherwise, it calculates tax as 6% of the price

Pull out code common to both if and else clause
Focus on the lines highlighted in the left panel. Work on them

```
1  // The C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      bool isFood;
8      float price;
9      float tax;
10     cout << "Enter whether it is a food item";
11     cin >> isFood;
12     if( isFood )
13     {
14         cout << "Enter the price ";
15         cin >> price;
16         tax = price * 0.02;
17         cout << "On food item, ";
18         cout << "the tax is " << tax;
19     }
20     else
21     {
22         cout << "Enter the price ";
23         cin >> price;
24         tax = price * 0.06;
```

```
1  // The C++ program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      bool isFood;
8      float price;
9      float tax;
10     cout << "Enter whether it is a food item";
11     cin >> isFood;
12     if( isFood )
13     {
14         cout <<
15         cin >> p
16         tax = p
17         cout <<
18         cout <<                   tax;
19     }
20     else
21     {
22         tax = price * 0.06;
23         cout << "On non-food item, ";
24         cout << "the tax is " << tax;
```

Delete line
Duplicate line
Move line ▸ Click on the line to which to move
Edit line

```
You deleted line 22: cout << "Enter the price ";
Deleting line 22 was correct
>
You chose a correct line (22) to refactor
You deleted line 22: cin >> price;
Deleting line 22 was correct
```

Lines deleted/edited so far:

```
22         cout << "Enter the price ";
22         cin >> price;
```

Time Elapsed: 2:00 Remaining: 298:00

Figure 1: The User Interface of the Tutors (© Amruth N. Kumar)

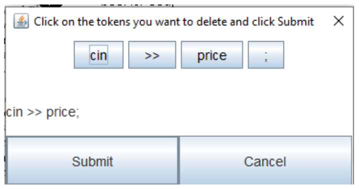
In Figure 1, a student has:

- Repeatedly clicked on *Explain* button in the refactoring panel **R** till the lines of code that must be refactored are highlighted in the problem panel **P**.
- Deleted lines 22 and 23 in the solution panel **S**. These lines appear in the trash panel **T**. (Both the lines are numbered 22 because once line 22 in the solution panel was deleted, line 23 was renumbered as line 22.) In the feedback panel **F**, feedback is provided to confirm that these deletions are correct.
- Clicked on line 14 and is currently contemplating moving the line. The student would click on line 12 to move line 14 to its correct location in the refactored code.

2.4 Grading and Feedback

After each refactoring operation, the tutors summarize the operation in the feedback panel. If the operation is unambiguously correct/incorrect, the tutors confirm its correctness. If an operation is part of a sequence of operations, it may not be possible to unambiguously determine its correctness. In such cases, no feedback is provided on the correctness of the operation.

Once students submit a solution, the tutors grade by comparing the source code of their solution with that of the correct solution. Since the tutors do not use test cases or execute code, they can grade even if students' solution is incomplete or syntactically incorrect. The tutors also present the correct solution in problem panel **P** juxtaposed with students' solution in panel **S** and highlight the refactored lines in the correct solution so that students can easily compare the two solutions.



cin >> price;

Figure 2: The user interface to edit a line one token at a time (© Amruth N. Kumar)

3 Using the Tutors

The tutors can be used for learning (with feedback turned on) or assessment (with feedback turned off). They may be used to learn anti-patterns not covered in lectures or solve problems on the anti-patterns introduced in lectures. In either case, they facilitate active learning through problem-solving.

The tutors have a built-in help menu, grade students' solution, assign credit for concepts learned, sequence problems, time themselves as necessary, and log usage data on a server. The user interface is intuitive enough that students can use the tutors without prior instruction, as has been our experience. Therefore, the tutors are self-contained and can be used unsupervised. The tutors are accessible over the web, so, students can use them on their own time after class just as easily as they can use them in class. Instructors can use them for after-class assignments just as easily as for closed-lab exercises. Their use places minimal demands on resources because instructors do not have to carve out classroom time for their use.

All of the following can be customized in the tutors: 1) the anti-patterns covered; 2) the problems presented for each anti-pattern; 3) the order in which problems are presented; 4) whether the tutors are set up for learning (with feedback) or assessment (with grading, but no feedback); and 5) whether the tutors are set up to present the same problem-solving experience to everyone or adapt to the needs of the learner.

When the tutors are configured to adapt to the learner's needs, they use a pretest-practice-post-test protocol [10]. During pretest, they present one problem per anti-pattern: the anti-patterns on which students solve the problem incorrectly are the ones that they need to learn. During practice, they present problems on only these anti-patterns until students learn to correctly solve problems on them. During post-test, they present a problem on each anti-pattern on which students solved practice problems. This adaptive mode helps students efficiently learn new anti-patterns while avoiding solving unnecessary problems on the concepts they already know [10].

A declarative representation is used in the tutors to catalog anti-patterns (e.g., S1-S8, L1-L6), so, it is easy to extend the tutors with additional anti-patterns that can be refactored without freehand coding, e.g., no need to compare against Boolean constants in conditions, class variable should be function/method variable if used in only one function.

Similarly, a declarative representation is used to encode the five components of each problem (Section 3.2), so, additional problems can be easily incorporated into the tutors for any supported anti-pattern. The tutors automatically generate feedback by comparing students' solution with the correct solution. Therefore, feedback need not be individually encoded for each new problem. This reduces the time and effort needed to add new problems to the tutors.

3.1 Formative Data

The tutors were formatively evaluated over three semesters: fall 2022 - fall 2023 in two courses taught by the developer at the host institution: CS1, the first C++ programming course and Programming Languages (PL), an advanced course taken by computer science majors in their third or fourth year. Students used the tutors twice: as an assignment in the last two weeks of the semester and again as an online exercise immediately after the final exam. Students solved two problems per concept during the assignment and one problem per concept after the final exam, all in C++ – neither contributed to the course grade. They were allowed ample time for completion - 60 minutes for each assignment and 120 minutes for the online exercise after the final exam.

Selection tutor: During the assignment, both CS1 (N=32) and PL (N=64) students scored the lowest (0.26 or less out of 1) on concepts S5, and S6. CS1 students also scored less than 0.26 on S1. Both the groups scored the highest on S3 and S4 (around 0.6 for CS1, and 0.8 for PL). Paired samples t-test yielded that the increase in score from the assignment to the final exam was significant on concepts S1, S2, S4 and S7 for CS1 and concepts S2, S6 and S8 for PL. The only significant decrease in score was for concept S3 for PL – a result that merits further investigation.

Loop tutor: The assignment scores were the lowest on concepts L1, L5 and L6 for both CS1 students (N=31, score < 0.3) and PL students (N= 64, score ≤ 0.4) Both groups scored the highest on concepts L3 and L4 (0.6 to 0.69). The score increased significantly from the assignment to the final exam on concepts L3 and L6 for CS1 and L6 for PL students. It decreased on concept L1 for CS1 students and merits further investigation. No other change from assignment to final exam was statistically significant.

Assignment scores show that both introductory and advanced students needed refactoring practice on at least some of the selection and loop concepts even though these concepts were covered in class in CS1. This confirmed the need for these refactoring tutors even when they were limited to anti-patterns that could be refactored with no freehand coding. *Improvement in score from the assignment to the final exam shows that students learned refactoring concepts by using the tutors.* This confirmed that both introductory and advanced students benefited from using the tutors. *But the learned concepts differed between the two groups,* as was to be expected.

3.2 Reflections on Usage

Students–both introductory and advanced–needed to learn about at least some of the anti-patterns covered by the tutors. They learned at least some anti-patterns by using the tutors. So, the tutors were useful even though they were limited to anti-patterns that could be refactored without any freehand coding.

At the end of the assignment, students had the opportunity to provide open-ended feedback online. The feedback they provided included suggestions for improvement of the user interface:

- The option to start over is desirable: A button should be added to the user interface to reset the solution.
- The actions needed to move a line were cumbersome: A drag-and-drop interface would be better.
- A counter showing the number of problems solved and the number of problems remaining would be useful.
- The ability to move or delete multiple lines at a time is desirable.

All of these improvements are currently being undertaken. Some students thought they could have solved the problems faster if they had been allowed to retype code in a text editor. But, allowing retyping of code suffers the same pitfalls as freehand coding – the code would have to be checked for syntax errors; code may have to be executed to verify correctness; and therefore, grading could yield false positive and false negative results.

When source code comparison is used for grading, student solution must match the correct solution exactly. Some students felt that they should have received credit when their solution was not an exact match, but was functionally equivalent instead. For example, when an input statement is factored out of an `if-else` statement because it appears in both `if` and `else` clauses, it may be moved after or before another independent input statement. Only one of those locations will be correct when exact match is used. Either location will be correct if equivalent match is used. One way to resolve this might be to favor exact match by also providing a sample run that the refactored code must produce if executed.

A related issue in designing refactoring problems involves striking the right balance between detail and pedagogic focus: a detailed setup can make the code look realistic, but it can also be distracting, especially to introductory students. Tightly cropped code can be pedagogically focused, but may make the code look contrived, especially to advanced students. One solution might be to present focused problems to introductory students and detailed problems to advanced students.

3.3 Future Work

Several additional anti-patterns that can be refactored without freehand coding are planned for inclusion in the two tutors. Additional tutors are also planned on functions, arrays and classes. Currently, the tutors cover C++, Java and C#. Extension to Python is being planned.

In addition to the four refactoring operations already implemented, a "code transformation" operation is planned that will enable inclusion of anti-patterns that would otherwise require freehand coding. Examples of such anti-patterns include transforming a parameter into a local variable in a function; and transforming a `while` loop into a `do-while` loop. Once this operation is incorporated, the tutors will be able to cover a wider range of anti-patterns.

Formative evaluation has shown that students–both introductory and advanced–learned refactoring concepts by solving problems using the tutors. In the future, we plan to investigate whether this improved knowledge translates to students writing better code in programming projects.

The tutors run on any Java-enabled computer. They are free for educational use and can be accessed at the web site auglets.org. Adopters welcome.

ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation under grant DUE-2142648.

REFERENCES

[1] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. Journal of Information Technology Education: Research 3 (2004), 245–262.

[2] 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.

[3] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. "I know it when I see it" Perceptions of Code Quality: ITiCSE '17 Working Group Report. In Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17). ACM, New York, NY, USA, 70–85.

[4] William J. Brown, Raphael C. Malveau, HaysW. "Skip" McCormick, and Thomas J. Mowbray. 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons.

[5] 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.

[6] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In Proceedings of the 20th Australasian Computing Education Conference (ACE '18). ACM, New York, NY, USA, 53–62.

[7] Fowler, M., *Refactoring: Improving the Design of Existing Code*. 2nd edition. 2019: Addison Wesley

[8] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, crowd-scale programming practice in the IDE. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2491–2500.

[9] J. Fitzpatrick. (2000) Applying the ABC Metric to C, C++, and Java. In More C++ Gems. Cambridge University Press, New York, NY, 245–264.

[10] Kumar, A. (2006). A Scalable Solution for Adaptive Problem Sequencing and Its Evaluation. In: Wade, V.P., Ashman, H., Smyth, B. (eds) Adaptive Hypermedia and Adaptive Web-Based Systems. AH 2006. LNCS vol 4018. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11768012_18.

[11] Robert L. Glass. 2002. Facts and Fallacies of Software Engineering. Addison-Wesley Professional.

[12] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan, Integrating pedagogical code reviews into a CS 1 course: an empirical study, in 40th SIGCSE Technical Symposium. 2009: Chattanooga, TN. p. 291-295.

[13] Christopher D. Hundhausen, Anukrati Agrawal, and Pawan Agarwal. 2013. Talking about code: Integrating pedagogical code reviews into early computing courses. ACM Transactions on Computing Education 13, 3, Article 14 (August 2013), 28 pages.

[14] Christopher Hundhausen, Anukrati Agrawal, and Kyle Ryan. 2010. The design of an online environment to support pedagogical code reviews. In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). ACM, New York, NY, USA, 182–186.

[15] S. Johnson. 1977. Lint, a C program checker. Technical Report 65. Bell Labs.

[16] S.-N. A. Joni and E. Soloway, "But My Program Runs! Discourse Rules for Novice Programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 95–125, 1986.

[17] Koenig, Andrew (March–April 1995). "Patterns and Antipatterns". Journal of Object-Oriented Programming. 8 (1): 46–48.

[18] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren, A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Transactions on Computing Education, 19(1), 2018

[19] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A Tutoring System to Learn Code Refactoring. In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21). ACM, New York, NY, USA, 562–568.

[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student Refactoring Behaviour in a Programming Tutor. Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research. ACM, New York, NY, USA, Article 4, 1–10.

[21] Xiaosong Li and Christine Prasad. 2005. Effectively teaching coding standards in programming. In Proceedings of the 6th conference on Information technology education. ACM, 239–244.

[22] 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.

[23] Cruz Izu, Paul Denny, and Sayoni Roy. 2022. A Resource to Support Novices Refactoring Conditional Statements. In Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1 (ITiCSE '22). ACM, New York, NY, USA, 344–350.

[24] Robert C. Martin. 2008. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

[25] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: clone refactoring. In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16). ACM, New York, NY, USA, 613–616.

[26] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. LitterBox: a linter for scratch programs. In Proceedings of the 43rd International Conference on Software Engineering: Joint Track on Software Engineering Education and Training (ICSE-JSEET '21). IEEE Press, 183–188.

[27] McConnell, S., Code Complete. 2 ed. 2004: Microsoft Press.

[28] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In Proceedings of the 23rd international conference on World wide web (WWW '14). ACM, New York, NY, USA, 491–502.

[29] Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart. 2013. A review of AI-supported tutoring approaches for learning programming. In Advanced Computational Methods for Knowledge Engineering. Springer, 267–279.

[30] Lin Qiu and Christopher Riesbeck. 2008. An incremental model for developing educational critiquing systems: experiences with the Java Critiquer. Journal of Interactive Learning Research 19, 1 (2008), 119–145.

[31] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-coding in Learning Introductory Programming. In Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18). ACM, New York, NY, USA, Article 13, 1–8.

[32] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In Proceedings of the 20th Australasian Computing Education Conference (ACE '18). ACM, New York, NY, USA, 73–82.

[33] Amy Shannon and Valerie Summet. 2015. Live coding in introductory computer science courses. J. Comput. Sci. Coll. 31, 2 (December 2015), 158–164.

[34] Suzanne Smith, Sara Stoecklin, and Catharina Serino. 2006. An innovative approach to teaching refactoring. In Proc. of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06). ACM, New York, NY, USA, 349–353.

[35] Sara Stoecklin, Suzanne Smith, and Catharina Serino. 2007. Teaching students to build well formed object-oriented methods through refactoring. In Proc. of the 38th SIGCSE technical symposium on Computer science education (SIGCSE '07). ACM, New York, NY, USA, 145–149.

[36] Deborah A. Trytten. 2005. A design for team peer code review. In Proceedings of the 36th SIGCSE technical symposium on Computer science education (SIGCSE '05). ACM, New York, NY, USA, 455–459.

[37] Scott A. Turner, Ricardo Quintana-Castillo, Manuel A. Pérez-Quiñones, and Stephen H. Edwards. 2008. Misunderstandings about object-oriented design: experiences using code reviews. In Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08). ACM, New York, NY, USA, 97–101

[38] Leo C. Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). ACM, New York, NY, USA, 738–744.

[39] Wake, W.C., Refactoring Workbook. 2004: Pearson Education, Inc.

[40] Yanqing Wang, LI Yijun, Michael Collins, and Peijie LIU. 2008. Process improvement of peer code review and behavior analysis of its participants. In Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08). ACM, New York, NY, USA, 107–111.

[41] M. Woodley and S. N. Kamin, "Programming studio: A course for improving programming skills in undergraduates," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '07. ACM, New York, NY, USA, 2007, 531–535.

[42] Eliane S. Wiese, Michael Yen, Antares Chen, Lucas A. Santos, and Armando Fox. 2017. Teaching Students to Recognize and Implement Good Coding Style. In Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17). ACM, New York, NY, USA, 41-50.

[43] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, Douglas C. Schmidt. (2023) ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. arxiv.org.

[44] Thorsten Haendler, Gustaf Neumann and Fiodor Smirnov. (2020). RefacTutor: An Interactive Tutoring System for Software Refactoring. In: Lane, H.C., Zvacek, S., Uhomoibhi, J. (eds) Computer Supported Education. CSEDU 2019. Communications in Computer and Information Science, vol 1220. Springer, Cham.

[45] Mario Levya. (2023). Refactoring Tutor: An IDE Integrated Tool for Practicing Key Techniques to Refactor Code. Master's Thesis, Massachusetts Institute of Technology. hdl.handle.net/1721.1/151544

[46] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, Yann Gael Gueheneuc. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software Volume 167. https://doi.org/10.1016/j.jss.2020.110610.

[47] Valerie Barr and Deborah Trytten. 2016. Using turing's craft codelab to support CS1 students as they learn to program. ACM Inroads 7, 2 (June 2016), 67–75.

[48] Luburić, N.; Vidaković, D.; Slivka, J.; Prokić, S.; Grujić, K.; Kovačević, A. and Sladić, G. (2022). Clean Code Tutoring: Makings of a Foundation. In Proceedings of the 14th International Conference on Computer Supported Education - Volume 1: CSEDU. SciTePress, 137-148.

[49] Sandalski, M., Stoyanova-Doycheva, A., Popchev, I. and Stoyanov, S., 2011. Development of a refactoring learning environment. Cybernetics and Information Technologies (CIT), 11(2).

[50] Eman Abdullah Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. 2024. Automating Source Code Refactoring in the Classroom. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024). ACM, New York, NY, USA, 60–66.

[51] Juliette Woodrow, Ali Malik, and Chris Piech. 2024. AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024). ACM, New York, NY, USA, 1442–1448.

[52] gerritcodereview.com

[53] github.com/cs50/style50

[54] Francisco Alfredo, André L. Santos, and Nuno Garrido. Sprinter: A Didactic Linter for Structured Programming. In Third International Computer Programming Education Conference (ICPEC 2022). Open Access Series in Informatics (OASIcs), Volume 102, pp. 2:1-2:8, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022)