Work in Progress: The RICA Project: Rich, Immediate Critique of Antipatterns in Student Code

Leo C. Ureel II
ureel@mtu.edu

Laura E. Brown
lebrown@mtu.edu

Jon Sticklen
stiklen@mtu.edu

Michelle Jarvie-Eggart
mejarvie@mtu.edu

Mary Benjamin
mbenjami@mtu.edu

Michigan Technological University
Houghton, MI 49931

ABSTRACT

Rich, relevant, and immediate student feedback is a core ingredient supporting effective student learning. Feedback is particularly important for introductory computing courses where novice programmers are still learning the basic syntax and semantics of a programming language. Our project is aimed at detecting poor solutions to common problems, termed antipatterns, in student code and providing feedback that guides the student to better solutions. This paper discusses the first year of the project, specifically, the development of a Code Critiquer to detect antipatterns in student code and generate appropriate feedback. This important first step sets-up the project to advance knowledge about novice antipatterns and their detection. The use of these antipatterns and code critiquers in future classroom interventions will help the project improve our understanding of student learning, retention, and self-efficacy.

Keywords

Antipatterns, MATLAB, Novice Programming, Code Critiquer, Engineering Fundamentals, First Year Experience

1. THE RICA PROJECT

Large introductory courses that introduce programming concepts and skills are challenging to both students and faculty. For students, there is an enormous time spent learning syntax and understanding semantics of a particular programming language while also learning general problem solving skills and paradigms. There is a need for immediate and repeated feedback when learning these fundamental skills. However, messages provided by compilers or run-time interpreters are often opaque and are geared towards experts, not novice programmers, where the experts have a clear understanding of the language and a deep model of what comprises a program and how a program is developed. For instructors, the ability to provide timely and rich feedback is taxed by the onerous time commitment required and compounded with large course sizes.

The RICA Project builds on the prior work in code critiquers by Qiu and Riesbeck [12], Brown and Pastel [5], Ureel and Wallace [15], and Walther [23]. Code critiquers analyze student code and provide rich and targeted feedback. Ureel and Wallace used a code critiquer to support learning Java programming in first-year computer science courses [16]. We aim to develop a MATLAB Critic, similar to Walther's MATLAB-TA, to assist students learning MATLAB programming in first-year engineering. The code critiquer used in this project, MATLAB Critic, is designed to detect the common mistakes, or *antipatterns*, that students make while learning MATLAB programming. By defining antipatterns, student and faculty can then converse using this vocabulary. Our project is centered on engaging and supporting student learning of programming competencies.

The project's **primary goal** is to determine if using a Code Critiquer supports effective student learning at scale. We are investigating this problem in a first-year engineering (FYE) course with approximately 1,000 students. We pose the following research questions to address this goal:

- **RQ 1**: Will the use of a Code Critiquer by first-year engineering students improve their computing skills?
- **RQ 2**: Will the use of a Code Critiquer by first-year engineering students improve student self-efficacy?
- **RQ 3**: The introduction of antipatterns to the curriculum will enable students and instructors to adapt a common pattern language. Will the common language improve student's skills in computational reasoning and communication about problems and code?

In the first year of the project we are focusing on identifying *Novice Antipatterns* and developing *Code Critiquer* software to provide students with feedback when antipatterns are detected in their code. Novice antipatterns are common code structures, found in student programs, that cause more problems than they solve. Novice antipatterns repre-

sent the kinds of coding mistakes that expert coders would never make.

Previous work identified over 200 antipatterns in Java [17]. Our goal during the first year of the project is to identify antipatterns in MATLAB and to further understanding of antipatterns generally. In this research, we will identify novice antipatterns observed in first-year engineering students' code. By arranging these antipatterns in a taxonomy, we can name common mistakes, and create a vocabulary through which both instructors and students can better discuss and address problems in programs.

If, as we hypothesize, many of these novice antipatterns transcend programming language, then we will have compiled a rich, descriptive language for communicating with novices about their code and the qualities of programming – both good and bad.

Additionally, within FYE programs, instructors are typically engineers, who may have taken only one or two programming courses within their undergraduate work. Thus, the faculty teaching young engineers programming skills often never progress much beyond a novice programming level themselves. A taxonomy of antipatterns provides these instructors with essential scaffolding for their instruction, and will advance their own reflection upon common student errors and effective teaching methods in response.

In higher educational environments with increasingly larger class sizes, the code critiquer can provide essential feedback that may be missing within the large class format.

2. NOVICE ANTIPATTERNS

In the classroom instructors will, based on experience, identify common structures in student code and call them out. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [1]. In this way, the code patterns students learn in a course form a kind of Pattern Language used to communicate efficiently on a level between verbose, plain English and terse computer code.

Instructor feedback is quite different from what a compiler tells a student. When analyzing student code, an instructor looks for *Novice Antipatterns*. Novice antipatterns are a kind of code pattern: recurring code snippets that novices use to solve problems; however, these antipatterns result in errors, bugs, and inefficient code. By defining novice antipatterns, we create a rich vocabulary for communicating about programming mistakes and what constitutes poorly designed code (and, conversely, what constitutes good code).

Novice antipatterns represent the kinds of mistakes that experts do not make. Consequently, professional tools often do not provide good feedback on these kinds of mistakes. An example of a common antipattern observed in beginning programming courses is the Empty Loop Antipattern (Figure 1).

After learning looping constructs, a student will sometimes introduce an empty loop to their code simply because "we just covered loops so they must be important!" An instructor can easily spot the empty loop and suggest to the student that it might not be needed as it contributes nothing to their solution, but could cause confusion later when the program is being debugged or modified. We have seen this antipattern in introductory courses using other languages, such as Java and Python.

Figure 1: Empty Loop Antipattern in MATLAB

```
A = [3 6 9 4 1];
% Loop serves no purpose.
for i = 1:length(A)

end
```

Examination of student code has helped us identify 200+ Java code antipatterns, which can be detected automatically. For example [17]:

- **Misplaced Code:** Inserting good code outside of any method or other appropriate enclosing structure. This will stop the compilation process, but the resulting error messages do not produce meaningful feedback that would assist a novice coder.
- **Pseudo-Implementation:** Students implement the methods called for by a Java interface, but neglect to use the reserved word *implements* in the class definition, thereby failing to enforce the contract of the interface type.
- **Localized Instance Variable:** Students declare an instance variable but only use it as if it were a local variable in a single method.
- **Repeated Resource Instantiation:** A common programming pattern is to instantiate a single resource (*e.g.,* an input or output stream) and uses that resource continually through execution. But student misunderstandings may lead to code that inadvertently instantiates multiple copies of the resource.

3. CODE CRITIQUERS

Like an instructor, a Code Critiquer analyzes student programs looking for Novice Antipatterns and responds with highly interactive and targeted feedback [2, 12, 8, 20, 9]. While approaches vary, these systems tend to make strong use of the instructor's domain knowledge to identify design issues and formulate meaningful responses (see Figure 2).

This makes them well-suited for providing novices with the kinds of feedback that an instructor might give in a classroom setting. For example, in Pseudo-Implementation (Figure 3), the student's code will compile and may execute without error. However, the instructor can configure the Java Critic to identify when the student has implemented code directly instead of using inheritance. The Java Critic can then provide the student with advice to improve their solution.

Critiquers are similar to autograders, but they focus on providing highly interactive and targeted feedback to student programmers [2]. While these systems often perform testing, they make strong use of the instructor's domain knowledge to identify patterns and formulate meaningful responses. This makes them well-suited for providing novices with the kinds of feedback we are attempting to emulate.

Figure 2: Information flow in a Code Critiquer.

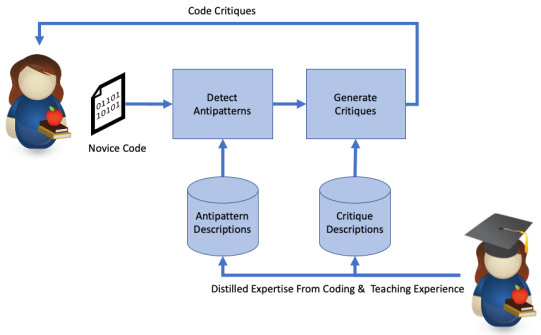


Figure 3: Example of a Java code critique.

```
public interface ReverseInterface {
    public String reverse( String s );
}

// Implements ReverseInterface
public class Reverse {
    // Method required by ReverseInterface
    public String reverse( s ) {
        if ( s.isEmpty( ) ) return s;
        return reverse( s.substr(1) ) + s.charAt( 0 );
    }
}
```

Critique: The assignment required you to implement the ReverseInterface interface. However, your class Reverse is not declared to implement the interface with the implements clause. This is important if your code is to work with other classes that expect your class to use the interface. Without it, your code will not compile into a larger system.

3.1 Related Work

Here we briefly describe some related systems before expanding on two systems, WebTA and MATLAB-TA, that we will be extending in our work.

Java Critiquer [12] provides individualized feedback and just-in-time learning opportunities to students. It performs static assessment of programming style, programming errors, and design by using regular expressions to match snippets of code with trigger patterns in instructor created rules. Students use the tool iteratively to improve their code before submission.

JUG [5, 6] provides students with fast, automated feedback. JUG (JUnit Generation) scripts to generate unit test cases and time complexity tests, which are executed to produce immediate student feedback. JUG is used as a support tool for graders, providing unit tests and reports for each student, but stopping short of automatically assigning a grade. JUG performs dynamic assessment of functionality through generated JUnit testing and efficiency through measurements of execution time.

Test My Code (TMC) [21] is an assessment system that enables instructors to build scaffolding and automated instructor-initiated feedback into programming exercises. TMC is integrated into the student's programming environment and provides tasks for the student to work on. The tool works within an environment of high instructor-student interaction, providing the ability for instructors to iteratively and precisely identify points of critique within student code.

JDeodorant [9] is an Eclipse plugin that detects several classic code smells [10] in source code, including Feature Envy, Type/State Checking, Long Method, God Class and Duplicated Code. JDeodorant is targeted for experienced programmers rather than novices.

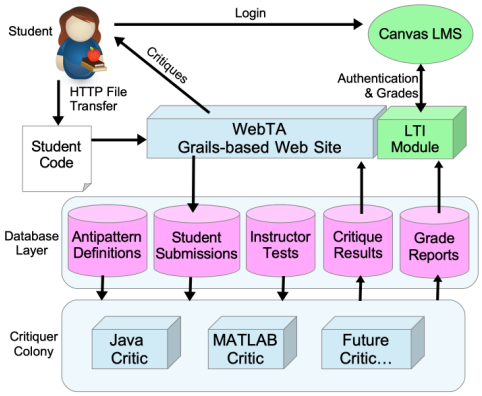
3.2 WebTA

WebTA is platform for developing and testing our Code Critiquer developed by Ureel [18]. WebTA is a federated architecture for critiquing code in the Canvas LMS [16, 19, 17].

WebTA has been used as a Java Critiquer in CS1, CS2, and CS3 courses since 2014. The system has been used by 1,421 students in 27 course instances. These students made 64,964 submissions to 119 assignments [17]. WebTA includes a library of 200+ Java-based novice antipatterns. However, this library was grown organically over time and lacks a rigorous system of classification.

WebTA Architecture: WebTA utilizes a federated architecture with a Grails-based Web Site and LTI Module comprising a front-end that handles all communication with the student through the Canvas LMS (see Figure 4) [18]. Critic modules are their own application. Inter-module communication issues are resolved through the database.

Figure 4: Integrated WebTA Critiquer System with Critic Colony Architecture



3.3 MATLAB-TA

A prototype MATLAB Critquer (MATLAB-TA), developed by Walther [22, 23], analyzes a student's MATLAB code providing error and style guidance and feedback. Similar

to WebTA, the MATLAB Critiquer compiles, tests, and analyzes code looking for antipatterns. The critiquer uses the MATLAB Java API; connecting to the MATLAB Engine that parses and executes student code submissions [11]. When an antipattern is detected, the critic generates a critique for the student. The critique covers code structure, shakedown test results, and programming style in a manner appropriate for novice coders.

Some key features of MATLAB-TA include:

1. MATLAB-TA provides feedback on syntax, runtime errors, requirements compliance via testing, and static analysis of code structures and programming style.

2. MATLAB-TA does not require configuration or test creation to provide feedback to students. Any code submitted will be analyzed for the presence of novice antipatterns in the Antipattern Library. This enables instructors to adopt MATLAB-TA immediately with no extra effort or setup.

4. SETTING: FIRST-YEAR ENGINEERING

The context of the project is a First-Year Engineering Program (FYEP) with an enrollment of approximately 1,000 students. FYEP is a common first-year engineering experience taken by all first-year students in the College of Engineering, where students are taught programming in MATLAB.

All calculus-ready students begin their two course first-year engineering sequence with the *Engineering Analysis and Problem Solving* course. This course provides students with an introduction to the engineering profession and to its various disciplines. It focuses on developing problem-solving skills and computational skills through introductory MATLAB programming. The course is taught in a flipped class format, where students watch recorded videos, read the textbook, and complete some preparatory assignments prior to class. Within class, students work on teams to apply the engineering problem-solving method to "real-world" problems through active, collaborative work.

Each *studio section* of 100-120 is composed of five *LEAP sections* of 20-24 students each. Each LEAP section has its own dedicated undergraduate TA, referred to as LEAP leaders (LEarning with Academic Partners). Within each week, students attend two 2-hour studio sessions for active learning with their large class of all 5 LEAP sections plus their entire teaching team (head instructor plus five LEAP leaders). Students also attend one weekly 1-hour LEAP session with only their section of 20-24 students and their LEAP leader. These sessions are led by the undergraduate TAs to reinforce learning the material covered in class during the week.

5. YEAR 1 OF THE RICA PROJECT

The RICA project's primary focus is on the development and implementation of a robust version of the MATLAB-TA prototype in WebTA and it's integration into the FYEP ENG1101 curricula to address our three research questions.

5.1 Detecting Antipatterns

Detecting antipatterns can be challenging for humans and computers alike. A problem is distinguishing a good solution from a bad one. Because bad solutions often manifest as bugs in the code, techniques such as debugging and testing are often used. However, because antipatterns are often subtle and some bugs are difficult to reproduce or test, different approaches are used in the literature. We currently use an ensemble of static and dynamic analysis techniques for detecting antipatterns. In future work, we are planning to extend our ensemble of methods to include the use of machine learning to detect antipatterns.

Static Analysis examines the program text. We use *Regular Expression Matching*, a form of static analysis where antipatterns are described using regular expressions. These regular expressions are matched against the program source code. A positive match indicates the presence of the antipattern in the targeted code [17, 19] Another form of static analysis that we utilize is *Abstract Syntax Tree Traversal* [17, 19] Antipatterns can be described as a arrangement of nodes in a syntax tree. Rules can be used to traverse the abstract syntax tree searching for the specified substructure. If the described arrangement of nodes is found, the antipattern has been detected.

Dynamic Analysis examines the properties of a program during execution [3]. Two types of dynamic analysis that we use for antipattern detection are *Unit Testing* and *Model Testing*. Unit testing is a common practice where programmers test of the smallest specified units of code or groups of related units in a software system [13]. In model testing, an executable model of the software system's behaviors and properties is constructed. The model is then systematically probed to find defects in the system [4].

5.2 Development of the MATLAB Code Critic

The development of the code critiquer builds on previous work by Qiu and Riesbeck [12], Brown and Pastel [5], Ureel and Wallace [15], and Walther [23].

- We have moved away from the proprietary MATLAB Java API that was used by Walther to perform syntax checking and runtime testing. Instead, we have implemented a parser and Abstract Syntax Tree (AST) based on the GNU Octave Project [7] Developing out own parser and AST results in lower memory requirements, faster performance, and opens up avenues for further customization.

- We have extended the library of regular expressions used by Walther. We added 64 new regular expressions to explain MATLAB syntax errors and police coding style. For example, The code in Figure 5 contains a tiny mistake where the student mistyped the built-in command `disp` with a leading capital letter. This produces a particularly unhelpful (especially to novices) error message: "Undefined function or variable 'Disp'". We turn that around with the feedback, "You made a call to **disp** in line # but didn't get the capitalization correct."

Initial development of our MATLAB Critic will continue through the Fall semester 2022. Using experience reports from the instructional staff (faculty and students), the system will be continuously updated and refined thereafter, for the life of the project.

5.3 The MATLAB Antipattern Database

Part of the RICA Project is to develop a taxonomy of novice antipatterns and to determine the extent of overlap between the taxonomies used to critique Java vs MATLAB code. Our hypothesis being that there will be a high degree of overlap between the types of mistakes novices make in the code of any language.

Our MATLAB Critic is designed to provide students with immediate feedback on common syntax errors, shakedown testing, and programming style. We detect antipatterns in the structure of the student code to trigger advice.

We are mining for antipatterns to store in our database in three ways:

1. First, we are mining student code submitted to the Canvas LMS during the 2021-2022 academic school year. This has helped us identify several code antipatterns, which we can then detect automatically and provide appropriate just-in-time feedback to students.

2. Second, we are mining the WebTA Java Antipattern Library. This library of over 200 antipattern definition contains many antipatterns that may be applicable across language contexts (for example, Floating-Point Comparison without Tolerance). We want to find the antipatterns that apply equally in the Java context as in the MATLAB context for teaching beginning students.

3. We are compiling a set of guidelines or reading techniques, similar to Travassis, et al. [14], for the purpose of aiding instructors in manually identifying antipatterns.

The examples given here are derived from actual student submissions, slightly modified for brevity.

5.3.1 *Capitalization Mistakes*

Beginning students spend much of their time coming to grips with the syntax of MATLAB. There are a number of ways to detect the common syntax mistakes made by novices; they can be detected by the parser, by static textual analysis, or by matching the MATLAB generated errors.

One example of a common syntax error is the mistaken capitalization of built-in MATLAB functions. Figure 5 shows a function containing an improperly capitalized call to `disp`. This produces an unfortunate MATLAB error that confuses novices, "Undefined function or variable". Our MATLAB Critic can detect this situation and provide the student with better feedback.

5.3.2 *Floating-point Loop Threshold*

Figure 5: MATLAB identifier mispelled with incorrect case.

```
function badcase(inputArg1,inputArg2)
    Disp("Hello")
end
```

MATLAB ERROR:

```
Undefined function or variable 'Disp'.

Error in badcase (line 2)
Disp("Hello")
```

Critiquer Feedback:
You made a call to **disp** in line 2 of badcase.m, but didn't get the capitalization correct. Almost all built-in MATLAB functions are named using all lowercase letters.

Students often forget that the sets of real numbers and floating-point numbers are not equivalent. This causes problems when students perform comparisons using floating-point numbers. For example, a loop using a floating-point terminating condition (see Figure 6) might result in unexpected behavior; (1) loop may fail to terminate, (2) Loop may not perform as many iterations as expected (perhaps more or less).

Figure 6: Code using a floating-point loop threshold.

```
total = 0.0;
threshold = 0.3;
while total <= threshold
    disp(total);
    total = total + 0.1;
end
```

Critiquer Feedback:
You used a floating-point comparison as the end condition for a loop. This can cause your loop to fall short of the number of expected iterations or it can result in an infinite loop. To mitigate this problem, check that the source value is within some tolerance range of the terminating value, i.e. $(total - threshold) \leq tolerance$

To mitigate this antipattern, students should implement the corresponding positive code pattern, Floating-point Loop with Tolerance, which is illustrated in Figure 7. When loop terminating condition involves floating-point values, provide a tolerance value for comparison. Our MATLAB Critic can also detect use of this positive pattern and provide positive feedback.

Figure 7: Code using a floating-point loop tolerance.

```
total = 0.0;
threshold = 0.3;
tolerance = 1E-15;
while (total - threshold) <= tolerance
    disp(total);
    total = total + 0.1;
end
```

An interesting aspect of this antipattern is that it is a special case of a more abstract antipattern: Direct Floating-point Comparison because we see similar antipatterns in any code structures that might utilize floating-point comparisons. For example in if statements,

```
if total <= threshold
```

versus

```
if (total - threshold) <= tolerance}
```

Furthermore, this antipattern appears to be a language independent antipattern appearing in any language that allows floating-point comparisons.

5.3.3 *Misuse of the* `rand` *Command*

Suppose the instructor wants the student to generate a random number in range $[0.0, 100.0)$. The instructor wants students to use a construct such as:

```
r = rand * 100.0 % r in range [0.0, 100.0)
```

However students might try

```
r = rand(100) % r is a 100 x 100 matrix
```

Here an instructor can configure the MATLAB Critic to check for this antipattern for the specific assignment where the call to `rand` could "unexpectedly" return a matrix instead of the expected floating-point result.

5.3.4 *Redefinition of the Complex Number* `i` *or* `j`

Here is a MATLAB specific antipattern. In MATLAB both i and j are built-in complex numbers that represent the imaginary number `sqrt(-1)`. MATLAB allows these identifiers to be used as variable names. However, they are still shadowed by their complex definitions and can be used as imaginary numbers resulting in confusing code.

Students most at risk of using this antipattern are those with previous coding experience who are used to using variables named `i` or `j`.

Detection of this antipattern in student code could also lead to a discussion of descriptive variable names and the avoidance of single character variable names where not mathematically motivated.

6. FUTURE WORK

Here, we have described some of the novice MATLAB antipatterns we have identified. Some are relatively easy to detect (for example, style guideline violations), while others require more subtle analysis or are conceptually tied to a particular language, context or assignment. These antipatterns do not represent the kinds of mistakes that expert programmers make. This may explain why they are not often found in professional or academic works. However instructors of intro-level programming courses, aware of the mistakes beginning students might make, can provide feedback through our MATLAB Critic that is more targeted the students' needs.

Figure 8: Redefining the Complex Number `i`.

```
i = 42;
disp(i)
disp(2i)
disp(2*i)
disp(2+2i)
disp(2i+2i)
```

MATLAB OUTPUT:

```
   42

  0.0000 + 2.0000i

   84

  2.0000 + 2.0000i

  0.0000 + 4.0000i
```

We will continue and expand our mining of submission data from the first year engineering courses in the Canvas LMS. A long term goals is to support instructors by, as much as possible, automate the process of identifying the novice antipatterns that need to be detected when examining student code. Apart from this data mining challenge, the concept of early programming antipatterns is of general interest.

Once the MATLAB Critic is ready for testing in the classroom, we plan to evaluate the ability of the MATLAB Critic to detect antipatterns by checking for antipatterns the critic fails to detect as well as correct code flagged as antipatterns.

As the project proceeds, we plan to develop and expand instructional materials to include pattern and antipattern specific material. We will create materials and training for on-boarding instructional staff. And we will assess learning outcomes on the use of our MATLAB Critic in the first year engineering courses.

A final point of distinction between the MATLAB Critic we are developing and MATLAB-TA is our plan to conduct a thorough evaluate MATLAB Critic with a large number of students. Our MATLAB Critic will be deployed for beta-testing in the classroom in Fall 2022 with broadening deployment planned for each following semester.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2142309. The authors wish to thank the following undergraduate students for their work on this project: Joe Teahen, Ricardo Nunez Cuesta, Katie Ulinski, Laura Albrant, and Daniel Masker.

8. REFERENCES

[1] C. Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[2] N. M. Ali, J. Hosking, and J. Grundy. A taxonomy and mapping of computer-based critiquing tools. *IEEE Transactions on Software Engineering*, 39(11):1494–1520, 2013.

[3] T. Ball. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234. Springer, 1999.

[4] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th international conference on software engineering companion*, pages 789–792, 2016.

[5] C. Brown, R. Pastel, B. Siever, and J. Earnest. Jug: a junit generation, time complexity analysis and reporting tool to streamline grading. In *Proceedings of the 17th ACM annual conference on Innovation and Technology in Computer Science Education*, pages 99–104, 2012.

[6] C. D. Brown. *An experience-driven pedagogy for the instruction of software testing in computer science*. PhD thesis, Michigan Technological University, 2013.

[7] J. Eaton and Et. Al. GNU Octave. https://octave.org/, 2022.

[8] S. H. Edwards and M. A. Perez-Quinones. Web-cat: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 328–328, 2008.

[9] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011.

[10] R. C. Martin. Smells and heuristics. In *Clean Code: A Handbook of Agile Software Craftsmanship*, chapter 17. Prentice Hall, 2009.

[11] Mathworks, Inc. Matlab, 2021.

[12] L. Qiu and C. Riesbeck. An incremental model for developing educational critiquing systems: experiences with the java critiquer. *Journal of Interactive Learning Research*, 19(1):119–145, 2008.

[13] P. Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[14] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM sigplan notices*, 34(10):47–56, 1999.

[15] L. C. Ureel and C. Wallace. Webta: Automated iterative critique of student programming assignments. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2015.

[16] L. C. Ureel and C. R. Wallace. Webta: Online code critique and assignment feedback. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 1111–1111, 2018.

[17] L. C. Ureel II. *Critiquing Antipatterns In Novice Code*. PhD thesis, Michigan Technological University, Houghton, MI, Aug 2020.

[18] L. C. Ureel II. Integrating a colony of code critiquers into webta. In *Seventh SPLICE Workshop at SIGCSE 2021 "CS Education Infrastructure for All III: From Ideas to Practice"*, 2021.

[19] L. C. Ureel II and C. Wallace. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 738–744, 2019.

[20] A. Vihavainen, M. Luukkainen, and M. Pärtel. Test my code: An automatic assessment service for the extreme apprenticeship method. In *2nd International Workshop on Evidence-based Technology Enhanced Learning*, pages 109–116. Springer, 2013.

[21] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, pages 117–122, 2013.

[22] M. Walther, L. Ureel, II, and C. Wallace. A prototype matlab code critiquer. In *Proceedings of the 2019 ACM Conference on innovation and technology in computer science education*, ITiCSE '19, pages 325–325. ACM, 2019.

[23] M. L. Walther. Matlabta: A style critiquer for novice engineering students. Master's thesis, Michigan Technological University, Houghton, MI, 2020.