

Board 234: Current Progress of Providing Rich Immediate Critique of Anti-patterns in Student Code

Dr. Leo C. Ureel II, Michigan Technological University

Leo C. Ureel II is an Assistant Professor in Computer Science and in Cognitive and Learning Sciences at Michigan Technological University. He has worked extensively in the field of educational software development. His research interests include intelligent learning environments, computer science education, and Artificial Intelligence

Dr. Laura E Brown, Michigan Technological University

Dr. Michelle E Jarvie-Eggart P.E., Michigan Technological University

Dr. Jarvie-Eggart is a registered professional engineer with over a decade of experience as an environmental engineer. She is an Assistant Professor of Engineering Fundamentals at Michigan Technological University. Her research interests include technology adoption, problem based and service learning, and sustainability.

Dr. Jon Sticklen, Michigan Technological University

Jon Sticklen is an Associate Professor with the Engineering Fundamentals Department (EF) and Affiliated Faculty with the Department of Cognitive and Learning Sciences (CLS). He served as Chair of EF from 2014-2020, leading a successful effort to design a

Laura Albrant, Michigan Technological University

After completing a bachelor's degree in computer science, Laura Albrant decided to challenge how she viewed software development, by switching departments. Currently working towards a PhD in human factors at Michigan Technological University, Laura pursues interests on both sides of the fence through computer science education research.

Mary Benjamin, Michigan Technological University

PhD Student in the Dept. of Civil, Environmental, & Geospatial Engineering at Michigan Technological university.

Daniel Masker, Michigan Technological University

Pradnya Pendse

Joseph Roy Teahen, Michigan Technological University

Current Progress of providing Rich Immediate Critique of Antipatterns in Student Code

Abstract: The “Rich, Immediate Critique of Antipatterns in Student Code” (RICA) project aims to provide rich, relevant, and immediate feedback to students learning to program in their first year of engineering education. This feedback is indispensable in effective student learning, particularly in introductory computing courses. Conventional classroom feedback mechanisms fall short here, partly because large-scale courses like those in First-Year Engineering (FYE) often strain the instructional team’s capacity to deliver timely feedback. Our project aims to address this challenge by developing Code Critiquers specifically tailored for First-Year Engineering (FYE).

1 The RICA Project

Our ongoing RICA project is developing a real-time Code Critiquer system, WebTA, that identifies, categorizes, and provides feedback on mistakes in MATLAB source code (antipatterns) that are commonly made by novices and unlikely to be made by experts. In programming, where the learning process is iterative and often fraught with errors, immediate feedback can serve as a critical form of scaffolding. The RICA project aligns with broader educational theory that supports the vital role of immediate feedback. However, it takes it a step further by focusing on the “richness” and “relevance” of this feedback. The project exists in the intersection of computer science, engineering, and cognitive & learning sciences. By focusing on antipatterns, it addresses the mental models that students form while learning to code. Antipatterns represent code structure, which, while usually syntactically correct, could lead to unintended consequences: errors, inefficiencies, or complexities.

The context of the project is a First-Year Engineering Program. At our institution, FYE has a typical total enrollment of approximately 1,000 students matriculating each fall into the College of Engineering. FYE is a common first-year engineering experience taken by all first-year students in the College of Engineering. During an Engineering Fundamentals course, students are taught programming in MATLAB.

The poster focuses on research conducted by our graduate students over the past year. This research includes a Human Factors analysis, development of a visual representation tool for specifying antipatterns using a visual representation of regular expressions, and work developing a Machine Learning algorithm to detect antipatterns.

2 Background

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.

The conceptual framework for RICA is informed by the foundational work of Alexander (1977) on pattern languages, which emphasizes the importance of identifying recurring problems and their solutions in a way that is universally applicable yet never redundant. This approach is adapted to the programming education context, where ‘antipatterns’—recurring coding mistakes that are counterproductive to good software design—are identified, named, and categorized. This not only facilitates a shared vocabulary between instructors and students but also allows for the development of critiquing systems that can provide specific, actionable feedback on these antipatterns.

The project builds upon the insights from pedagogical research that highlights the critical role of immediate feedback in the learning process. Studies by Shute (2008) and Narciss (2008) underscore the value of formative feedback in fostering deep learning and mastery of complex skills, such as programming. This body of work advocates for educational interventions that are responsive to student needs, providing guidance that is both timely and tailored to their specific learning contexts.

In synthesizing these perspectives, the RICA project endeavors to address the unique challenges faced by novice programmers, particularly those in first-year engineering programs. By leveraging the capabilities of the WebTA system to provide rich, immediate critique of antipatterns, the project aligns with contemporary educational theories that prioritize student engagement, personalized learning experiences, and the development of critical thinking and problem-solving skills.

2.1 The WebTA System

WebTA is a research platform for developing and testing our Code Critiquers [1]. WebTA is a federated architecture for critiquing code in the Canvas LMS [2, 3, 4, 1]. WebTA has been used since 2014. The system has been used by 1,421 students in 27 course instances. These students made 64,964 submissions to 119 assignments [4]. 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. Critic modules are their own application. Inter-module communication issues are resolved through the database.

2.2 MATLAB Critic

A prototype MATLAB Critic, developed by Walther [5, 6], analyzes a student’s MATLAB code providing error and style guidance and feedback. The MATLAB Critic parses and analyzes code looking for antipatterns. 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.

The following MATLAB antipattern example is derived from actual student submissions, slightly

modified for brevity.

Floating-point Loop Threshold: 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 1) 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).

```
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. $(total - threshold) \leq tolerance$

Figure 1: Code using a floating-point loop threshold.

3 Current Work

The following subsections highlight current work by our team of graduate students. This research includes a Human Factors analysis, development of a visual representation tool for specifying antipatterns using a visual representation of regular expressions, efforts to develop a common Abstract Syntax Tree representation for multiple languages (in particular Java, MATLAB, and Python), and work developing a Machine Learning algorithm to detect antipatterns.

3.1 Human Factors

A small study was conducted to examine the WebTA interface from a Human Factors (HF) perspective. Insights from the study reveal HF issues through the eyes of students.

We apply a blend of three HF viewpoints corrective, preventive, and prospective ergonomics. Corrective ergonomics focuses on “the problem to correct” [7]. In terms of WebTA, the ‘problem’ is the presence of antipatterns in students’ programs. Preventive ergonomics focuses on the system’s design, usually about design choices that minimize the chance of human error by accounting for human ability, or lack thereof. Prospective ergonomics is defined as “[an attempt] to anticipate human needs and activities to create new artifacts that will be useful and provide positive user

experience”. This approach helps us analyze a learning environment, training process, in our case, a critiquer system.





#	Code	Critique
000	<code>% S11 #2 - For Loops</code>	 Make sure to have a header comment at the top of every MATLAB source file. <pre> % Program Name: myprogram.m % Program Description: % What my program does and how and why. % Name: J. Doe (JDoe@mtu.edu) % Section: L00 % Team: 03 </pre>
003	<code>% 1. Use a for loop to determine the sum of integers from 1 to 5 and store the result in the variable, sum5.</code>	 For readability and printability, lines of code should be less than 80 characters. This line was over 80 characters, please break it up into two or more lines. For information about how to split up a line of code, see https://www.mathworks.com/help/matlab/matlab_prog/continue-long-statements-on-multiple-lines.html What affect do long lines have on code readability and programmer fatigue?
006	<code>for i = 1:5</code>	 Definition of i shadows a builtin function or constant.  The system already knows about the imaginary number i. What are the possible ramifications if you change the value of i? (Try it and see what happens.)

Figure 2: Current display of code critiques.

One result of the study was the determination that the way WebTA displays critiques causes excessive cognitive load for students. WebTA users view a traffic light system to portray the severity of antipattern(s) found within a student’s program. A ‘Green’ light means no antipatterns and/or a good pattern was found within the code. A ‘Yellow’ light means a non-critical antipattern or error was found. A ‘Red’ light means a critical error was encountered. The design of the individual critique pages (i.e., the page shown after a student submits their code) includes a summary section, a table that breaks down the code in the file line-by-line with corresponding critiques, and the student’s submission printed in it’s entirety. This results in a wall of text that can be confusing and frustrating to students. This is a frequent issue for user interfaces [8]. It is necessary for the application to format the information in a way that accounts for the limitation. The ability to hide/reveal extra information is a basic HF solution.

Figure 2 shows the partial display of a submission’s critique table.

Figure 3 shows a reworking of the the first critique on line 0 with a few improvements. Initially, only the student’s line of code is shown with a stoplight indicating the presence of a critique and it’s severity. The primary substance of the critique starts hidden and is only revealed by clicking the ‘Expand’ button. This is to avoid overloading the student with too much too soon and it provides an opportunity for the student to identify the problem themselves.


#	Code	Critique
0	<code>% S11 #2 - FOR LOOPS</code>	 Red Light Collapse ^ This assignment requires a specifically formatted header be at the top of every MATLAB file. Pay close attention to the spacing and characters used in the example header below to find where yours is different. <pre> % Program Name: myprogram.m % Program Description: % What my program does and how and why. % Name: J. Doe (email) % Section: L00 % Team: 03 </pre>

Figure 3: Reworked feedback revealed as needed.

color-blind users.

The use of the traffic light is well grounded as the connotations of a traffic light and its associated colors are known worldwide. The color of the traffic light is always shown as anticipation to students’ desires (i.e., prospective ergonomics). Initially showing only a simple icon with the severity of the message allows students to visually dissect the page at a glance. This, in turn, provides students with the ability to work more proficiently. Additionally, a ‘Red Light’ caption under the traffic light is added in the rework, addressing the accessibility of

Importantly, while reframing the issue as a corrective ergonomics problem, it became clear the critique messages function just as an error message does; to inform the user of a problem that requires correcting. Good error messages are, among other things, relevant, actionable, user-centered, and courteous [8]. This is motivating a review of our critique message to ensure they inform the students in a way that is actionable and user-centered.

Our HF analysis also indicated a need for training pages to educate students on antipatterns and associated terminology to assist students in the use of our critiquer system.

The analysis outlined above was an attempt to ensure that the development and design of WebTA is centered on the student (aka the human part of the system) as well as to suggest possible amendments to further the design for the better. In addition to motivating a user interface redesign effort, conducting frequent cognitive walkthroughs or more structured, in-depth usability testing will help us improve the system for students.

3.2 Regular Expression Builder

Regular expressions, or regex(ex), are a form of programming language to match patterns found within text/characters. In this day and age, regexes are a staple in the computing world. From data pack inspection [9] to text parsing to search engines, its uses are vast [10] and it is utilized frequently [11]. However, regular expressions are known to have poor readability and comprehension [12], lack generalizability to other languages in a number of scenarios [13], and differs from the standard Object Oriented Programming way of thinking [14].

Antipattern specification are represented using Regular Expressions (Regex) within WebTA. While regular expressions provide an excellent tool for text capturing, they are perceived to be unintuitive and often pose a steep learning curve for instructors when inputting new antipatterns. As part of a class project, a team of undergraduate and graduate students developed a proof-of-concept system that utilized a visual representation (a'la Blockly [15]) for entering regular expressions. This tool, WebTA's Regex Builder, aims to be a soft, visual introduction to novices of regular expressions. It takes the approach of "blocky" code, much like many popular applications (e.g. Scratch [16], NetLogo [17], Lego Mindstorms [18], etc.). The Regex Builder can allow professors or teachers regex knowledge to report new antipatterns to the database without hassle.

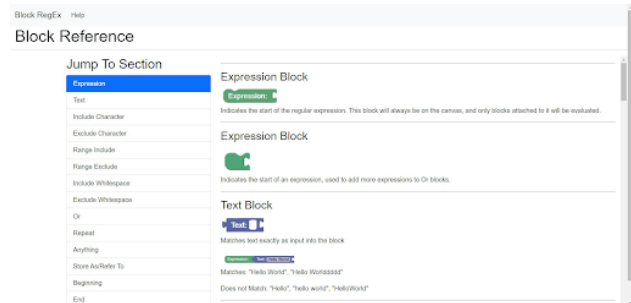


Figure 4: Regex Builder Help Page. This part of the application defines every usable block.

Overall, we found that the application shows a lot of potential. There were multiple participants who were able to create correct regular expressions having not known regex beforehand. With the recommended improvements, we strongly believe that this application will achieve its purpose(s).

3.3 Identifying Problematic Code using ML

Interest in machine learning algorithms has exploded in recent years. Their usefulness relating to identifying coding problems is therefore incredibly relevant, as outlined by a literature review of various machine learning models detecting code smells [19]. Machine learning algorithms represent a promising avenue for automating the identification of antipatterns. A proof-of-concept program was developed targeting Python as the language being critiqued. By leveraging the Random Forest, SVM, Logistic Regression, and other classification machine learning algorithms we endeavored to identify two antipatterns: *Syntax Error in Select Statement Condition*, a parsing issue, and *Crowded Operator*, a style issue. The aim was to systematically assess various classification models for each distinct antipattern, employing an ensemble approach to discern the optimal model that exhibits superior performance in identifying coding anomalies.

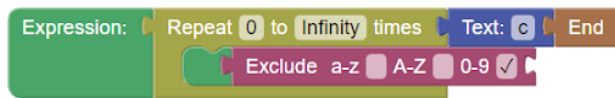


Figure 5: An example of a visual regular expression.

each code snippet underwent individual extraction, resulting in separate files for granular analysis. Subsequent meticulous labeling associated each file with specific antipatterns, establishing a robust ground truth for supervised learning. The dataset was then strategically divided into training and testing subsets, with the former serving as the foundation for model training. This bifurcation allowed for the independent evaluation of model performance on the testing set.

The project’s approach is to use different classification methods, such as Random Forest, SVM with Scikit-learn library for model implementation. All the models listed below are implemented on crowded operators and syntactical check for conditional code (if condition).

Method 1 Random Forest: The algorithm random forest [21] was selected for its robustness in handling diverse and complex datasets, making it well-suited for our task of classifying various antipatterns. Leveraging the labeled dataset, the model was trained on a training set, allowing it to discern patterns associated with different coding anomalies. Subsequently, the model’s performance was assessed on a dedicated testing set, drawn from the same dataset, to evaluate its ability to generalize to unseen

Data: The dataset utilized in this study consists of 900 Python code snippets in a dataset downloaded from Kaggle [20]. Some of these snippets come injected with antipatterns, while others adhere to correct coding practices. Preprocessing efforts were undertaken to enhance the dataset’s suitability for machine learning analysis. Initially consolidated into a unified file,

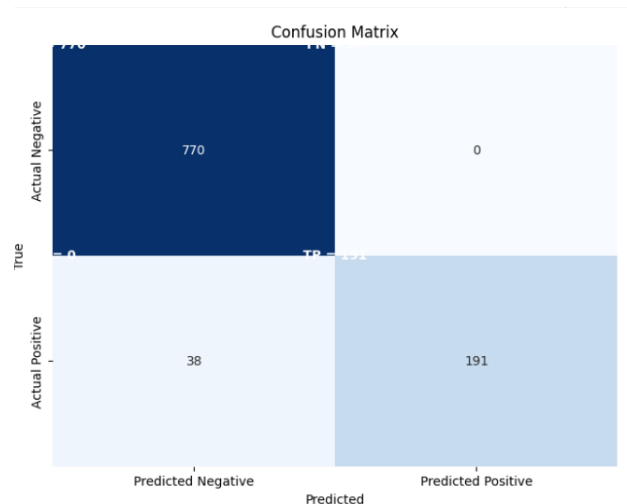


Figure 6: Confusion Matrix for Crowded Operators SVM Model

data. For the crowded operator antipattern, the hyper-parameter chosen for depth of the Random Forest is 15 and the accuracy achieved for this model is 81%. The accuracy is noted after running the same model multiple times. For the syntactical error with conditional code, the hyper-parameter chosen for depth of the Random Forest is 1 and the accuracy achieved for this model is 93%. The accuracy is noted after running the same model multiple times.

Method 2 SVM: Support Vector Machines (SVMs) are chosen for their effectiveness in high-dimensional spaces, making them well-suited for tasks like identifying antipatterns in code snippets. SVMs exhibit robustness to overfitting, particularly in scenarios with numerous features, preventing a loss of generalization. Their versatility in employing various kernel functions allows for capturing non-linear relationships within code structures. The clear objective of maximizing margins between different classes enhances generalization, while their consistent performance across domains, including text classification, suggests applicability to diverse coding patterns. With a focus on small to medium-sized datasets and a proven track record in classification tasks, SVMs present a compelling choice for discerning complex relationships inherent in code analysis. The SVM model was built on train data and then validated on validation dataset and then tested for various statistics on test data.

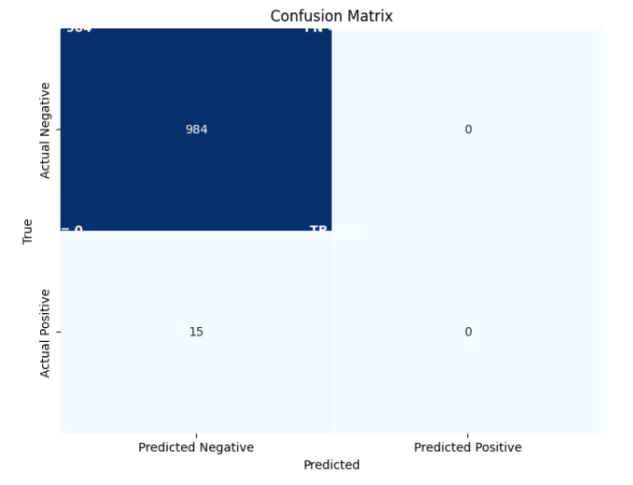


Figure 7: Confusion Matrix for syntactical errors in conditional code SVM Model

For crowded operators the SVM model performed the best as compared to Random Forest and other models with highest accuracy rate of 96/For the syntactical error with conditional code, the accuracy achieved for this model is 98%. The accuracy is noted after running the same model multiple times. Although, the accuracy seems high, it might not be the best matrix to consider here, as specificity and sensitivity are not being considered, which may impact the performance of the model. The confusion matrices in Figures 6 and 7 can help looking in to this missed aspects. below is the confusion matrix for both the antipatterns with SVM model.

Logistic Regression: Logistic regression is a statistical method that is used for binary classification problems. Despite its name, logistic regression is primarily used for classification tasks, not regression tasks. It's a type of generalized linear model that uses the logistic function to model the probability that a given instance belongs to a particular category. As for antipattern classification, the dataset code snippets either have antipattern of the particular kind or it doesn't, hence this could be a good data to try out logestic regression model. For the crowded operator antipattern, the accuracy achieved for this model is 98%. The accuracy is noted after running the same model multiple times. For the syntactical error with conditional code antipattern, the accuracy achieved for this model is 97%. The accuracy is noted after running the same model multiple times.

Ensemble Model: An ensemble model combines the predictions of multiple individual models to

improve overall performance and robustness. Ensemble methods are particularly useful as individual models have different strengths and weaknesses or when dealing with noisy or uncertain data. We used ensemble method with random forest and gradient boost models.

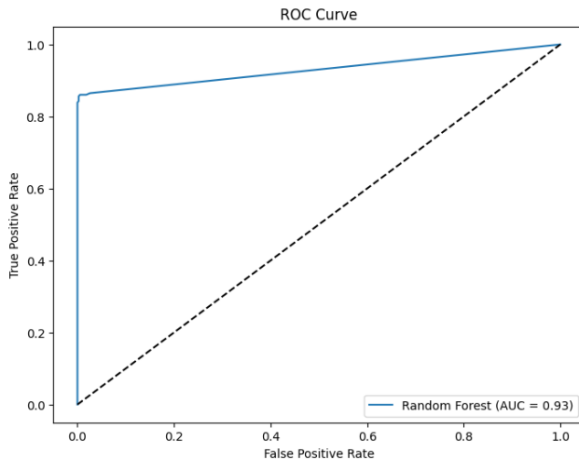


Figure 8: ROC curve for crowded operator antipattern's ensemble model

The process of parsing the Kaggle dataset into individual Python scripts using a custom algorithm might have introduced errors. Additionally, the automated labeling process for ground truth may not have been completely accurate, impacting the model's performance. Future work will focus on refining the parsing process and thoroughly checking ground truth labels for accuracy.

A major limitation was the relatively small dataset. To enhance model generalizability, obtaining larger and more diverse datasets, such as code written by novice students, could provide a more realistic scenario where code quality is often a concern. Larger datasets may lead to more accurate and robust antipattern detection models.

Lastly, as we show from the ROC curve, and confusion matrix, we can see that the data is imbalanced with true positive and true negative values. This is due to the imbalanced data and small size of original dataset. To rectify this, we need to use a larger dataset and expose the models with more balanced samples to have accurate training and prediction mechanism.

For the crowded operators antipatterns, the ensembles method with above mentioned models picked random forest as the best model with accuracy 96 % and another statistics that gives insights in to models overall performance with consideration of sensitivity and specificity is AUC (area under curve) represented with the help of ROC graph. Figure 8 shows the ROC plot for the crowded operator antipattern for the ensemble model.

The best performing model for the syntactical error with conditional code antipatterns was also Random Forest model with accuracy of 98%. To see overall performance of the model, we need to look at ROC curve if Figure 9.

While this project showed a preliminary method for identifying antipatterns in Python code, there are areas that could be enhanced.

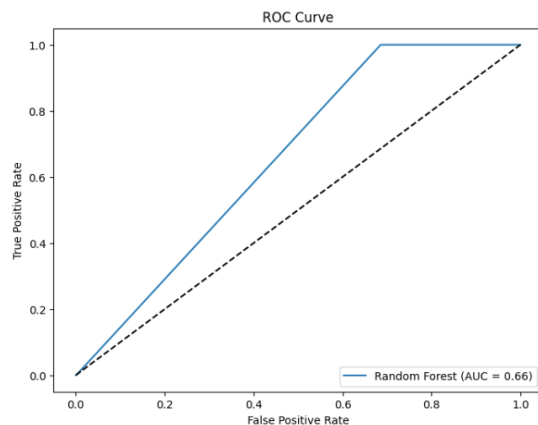


Figure 9: ROC plot for syntactical errors in conditional code antipattern's ensemble Model

4 Future Work

The RICA project embodies the significant potential of immediate feedback systems in enhancing learning outcomes. The project underscores the transformative power of automated code critique tools in addressing common antipatterns and fostering a deeper understanding of programming concepts among novice learners.

Looking ahead, the project will extend its scope to include a broader range of programming languages beyond Java and MATLAB, with Python identified as an immediate next step. This expansion aims to validate the system's effectiveness across different programming paradigms and educational contexts. Additionally, future iterations of WebTA will incorporate AI-based mechanisms to enhance the tool's adaptability to various pedagogical strategies.

The ongoing development and research will continue to be guided by educational theories on feedback and learning, with an emphasis on creating inclusive, supportive, and engaging learning environments for all students. The ultimate goal is to contribute to the broader discourse on educational technology and programming education, providing insights and tools that can be adopted and adapted by educators worldwide.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2142309. The authors wish to thank the following students for their work on this project: Pradnya Pendse, and Katie Ulinski.

References

- [1] 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.
- [2] 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*, pp. 1111–1111, 2018.
- [3] 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*, pp. 738–744, 2019.
- [4] L. C. Ureel II, *Critiquing Antipatterns In Novice Code*. PhD thesis, Michigan Technological University, Houghton, MI, Aug 2020.
- [5] M. Walther, L. C. 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*, pp. 325–325, 2019.
- [6] M. L. Walther, "Matlabta: A style critiquer for novice engineering students," Master's thesis, Michigan Technological University, Houghton, MI, 2020.
- [7] J.-M. Robert and E. Brangier, "What is prospective ergonomics? a reflection and a position on the future of ergonomics," in *Ergonomics and Health Aspects of Work with Computers: International Conference, EHAWC 2009, Held as Part of HCI International 2009, San Diego, CA, USA, July 19-24, 2009. Proceedings*, pp. 162–169, Springer, 2009.
- [8] R. Oshana, "Human factors and user interface design for embedded systems," in *Software Engineering for Embedded Systems*, pp. 417–440, Elsevier, 2013.

- [9] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *SIGCOMM Comput. Commun. Rev.*, vol. 36, p. 339–350, aug 2006.
- [10] G. R. Bai, B. Clee, N. Shrestha, C. Chapman, C. Wright, and K. T. Stolee, "Exploring tools and strategies used during regular expression composition tasks," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 197–208, IEEE, 2019.
- [11] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 282–293, 2016.
- [12] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 405–416, IEEE, 2017.
- [13] J. C. Davis, D. Moyer, A. M. Kazerouni, and D. Lee, "Testing regex generalizability and its implications: A large-scale many-language measurement study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 427–439, IEEE, 2019.
- [14] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 415–426, IEEE, 2019.
- [15] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with blockly," in *2017 IEEE Blocks and Beyond Workshop (B&B)*, pp. 21–24, 2017.
- [16] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.
- [17] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *International conference on complex systems*, vol. 21, pp. 16–21, Citeseer, 2004.
- [18] F. Klassner and S. D. Anderson, "Lego mindstorms: Not just for k-12 anymore," *IEEE robotics & automation magazine*, vol. 10, no. 2, pp. 12–18, 2003.
- [19] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad smell detection using machine learning techniques: A systematic literature review," *Arabian Journal for Science and Engineering*, vol. 45, pp. 2341–2369, 2020.
- [20] Kaggle Inc., "Kaggle," 2024. <https://www.kaggle.com>.
- [21] L. Breiman and R. Schapire, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2001.