Automated Critique of Early Programming Antipatterns*

Leo C. Ureel II

Michigan Technological University
Houghton, Michigan
ureel@mtu.edu

Charles Wallace

Michigan Technological University
Houghton, Michigan
wallace@mtu.edu

ABSTRACT

The introductory programming lab, with small cycles of teaching, coding, testing, and critique from instructors, is an extraordinarily productive learning experience for novice programmers. We wish to extend the availability of such critique through automation, capturing the essence of interaction between student and instructor as closely as possible. Integrated Development Environments and Automated Grading Systems provide constant feedback through static analysis and unit testing. But we also wish to tailor automated feedback to acknowledge commonly recurring issues with novice programmers, in keeping with the practice of a human instructor. We argue that the kinds of mistakes that novice programmers make, and the way they are reported to the novices, deserve special care.

In this paper we provide examples of early programming antipatterns that have arisen from our teaching experience, and describe different ways of identifying and dealing with them automatically through our tool WebTA. Novice students may produce code that is close to a correct solution but contains syntactic errors; WebTA attempts to salvage the promising portions of the student's submission and suggest repairs that are more meaningful than typical compiler error messages. Alternatively, a student misunderstanding may result in well-formed code that passes unit tests yet contains clear design flaws; through additional analysis, WebTA can identify and flag them. Finally, certain types of antipattern can be anticipated and flagged by the instructor, based on the context of the course and the programming exercise; WebTA allows for customizable critique triggers and messages.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; CS1; • **Applied computing** → **Computer-assisted instruction**; • **Software and its engineering** → **Patterns**; *Object oriented languages*;

KEYWORDS

CS1, design patterns, autograder, critiquing systems

*This work is supported by the National Science Foundation under grant DUE-1504860.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCSE '19, February 27–March 2, 2019, Minneapolis, MN, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5890-3/19/02...\$15.00

<https://doi.org/10.1145/3287324.3287463>

ACM Reference Format:

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), February 27–March 2, 2019, Minneapolis, MN, USA*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3287324.3287463>

1 INTRODUCTION

Many automatic approaches have been developed for evaluating student programs [6, 7, 13]. Often these tools function as auto-graders, supporting the instructor in testing and grading programs submitted by large numbers of students. However, automated assessment can also be used to provide students with feedback before submitting code for grading. Program characteristics assessed dynamically include functionality, efficiency, and testing coverage. In addition, static analysis tools can be used to assess program characteristics such as coding style, programming errors, software metrics, and adherence to good design principles.

Our automated code critique tool *WebTA* falls into this latter category. WebTA acts as an assistant to the student, commenting on intermediate code iterations during development, and then as a traditional autograder for final submissions. Our inspiration is the computer lab experience, where students engage in tight loops of coding, testing, and critique, with human lab instructors providing continuous feedback. We wish to simulate as closely as possible the experience of interacting with a human mentor.

With this aim, we wish to detect and comment on programming and design flaws early in the process. Through our experience as instructors, we can anticipate *antipatterns*, commonly occurring practices that reflect misunderstandings or poor design choices [5], and plan interventions accordingly. Many studies of novice programmers point out how student misconceptions, particularly concerning the object-oriented paradigm, are manifested as antipatterns [4, 11, 17, 21].

Some early programming antipatterns cut across traditional boundaries of syntactic wellformedness or behavioral correctness, and therefore require special effort to detect. WebTA constructs an Abstract Syntax Tree for student code and performs analysis on this structure to detect certain kinds of anomalous code—some syntactically well-formed, others not.

Furthermore, we want to tailor our feedback to be most effective for learning. Messages that may be appropriate for a general purpose Integrated Development Environment like Eclipse may be confusing, misleading, or even intimidating for novices, and this can have detrimental effects on their persistence in studying computing [2]. Another risk of cryptic error messages is that they may inoculate students against carefully reading such feedback.

From our experience teaching introductory programming, the following categories of antipattern have emerged:

- Some antipatterns result in syntactically ill-formed code, but we do not wish to report them with a standard compile-time error message; the standard message may be quite opaque and confusing, whereas we can say something more relevant and meaningful to a novice student.
- Other antipatterns do not result in compile time errors and have no effect on behavior, thus are not detectable through testing. A human instructor looking at student code would note such “code smells” [10, 15] and call them out. This provides an important teaching moment, highlighting the importance of good design beyond simple syntactic well-formedness and behavioral correctness.
- Still other antipatterns may defy easy static analysis but can be detected through “insider knowledge”. In our position as instructor, we can fine-tune our feedback to the particulars of the assignment, allowing us to give comments that would be outside the scope of a general analysis.

We begin our paper with an overview of WebTA and related work on automated grading and critique (§2). We then give examples of common antipatterns in student code and explain how we detect them and respond to them through WebTA (§3). We conclude with some ideas for future development (§4).

2 TOOLS FOR AUTOMATED GRADING AND CRITIQUE

A wide array of tools provide assistance to programmers, both experienced and novice, in the form of continuous feedback on static and dynamic properties of their code. We start with a discussion of our tool, WebTA, and compare it to related products in the domains of Integrated Development Environments (IDEs) and Automated Grading Systems.

2.1 WebTA

WebTA is a web-based tool that critiques student programs in introductory Computer Science courses. An optional Learning Tools Interoperability (LTI) Module provides services required for authentication with, and grade reporting back to, the Canvas Learning Management System (LMS). Although Canvas is our platform of choice, in principle the module can plug in to any other LMS implementing the LTI standard, including Blackboard and Moodle.

Data sources for the system include Student Code, Instructor Tests, Patterns Database, and Grade Reports. The Patterns Database contains four kinds of rules for identifying patterns and dispensing advice to the students. A basic rule contains a description of a code pattern that the instructor may recognize manually. Error rules are used to provide feedback and hints for compile-time and runtime errors. A regular expression rule can be used to identify patterns in the source code. Rules can be used to traverse the abstract syntax tree to identify more subtle patterns.

Every time a student submits code to WebTA, their code is subjected to all tests designed for the project. Students learn to develop code to pass each test and immediately discover if new code has side-effects that cause other tests to fail. Students are provided with immediate feedback on their programs, just when they need it during the development process. Faculty encode guidance and feedback into a database of common mistakes. When student code

Listing 1: Instructor Tests

```

1 @Test
2 @CanvasTaTest( points = 5,
3               name = "fibRecursive_Edge_Case",
4               description = "Checks_for_n=-1" )
5 public void fibRecursiveEdgeCaseTest( ) {
6     Fibonacci prog = new Fibonacci( );
7     long result = prog.fibRecursive( -1 );
8     if ( result != -1 ) {
9         fail( "fibRecursive(" + -1 + ") = " + -1
10            + ". Your method returned " + result );
11     }
12 }
```

exhibits symptoms of a problem, the database guidance is triggered and students immediately receive the instructor’s advice. This kind of guidance covers situations arising from compile and runtime errors, as well as more general coding style mistakes.

When students connect with WebTA through Canvas, a startup screen that explains the current problem and tells them which files they should upload to receive a code critique. After clicking on the *Critique My Code* button, students receive an online report which includes a Critique Summary. Under the hood, the system has compiled the student’s code and run it through a series of shake-down tests. Students can scroll down from the critique summary to view details of the critique, including errors and warnings generated both at compile-time and run-time. Scrolling further down the code critique, students find a listing of each code file submitted, with advice generated via textual analysis of the code.

The instructor can configure the system to run both public and secret tests, run the student’s own test code against their program, assess the student’s unit test cases to determine their ability to generate edge cases, produce feedback by matching rules run against the source code, or matching rules against output text.

As an example, consider the following assignment:

Develop a class named Fibonacci that contains the method fibRecursive which given an integer n returns the long nth element of the Fibonacci Sequence. If n is out of range, return -1.

In addition to testing common Fibonacci values, the instructor wants to ensure that the students have conformed to the range specification. The instructor sets up a secret test, with the result of the test shared but not the test inputs themselves. This test, using JUnit test conventions, is shown in Listing 1. Tests can be toggled as secret or shared dynamically through the WebTA interface.

The instructor also creates rules for the static analyzer, or reuses existing ones. Rules consist of a regular expression matching a potential problematic code snippet, coupled with appropriate advice. For example, anticipating a Knee-Jerk pattern (§3.4) where students add a vacuous base case, the instructor could set up the rule shown in Listing 2. In this way, the instructor provides the student with better feedback than the error message (stack overflow) triggered by failing a test.

Listing 2: Instructor rule capturing an empty base case.

```

1 if match( " if _ * \ ( . * \ ) \ s * \ { \ s * \ } " )
2 " It _ looks _ like _ you _ have _ an _ empty _ base _ case . "

```

2.2 Related tools

We include a brief survey of tools related to WebTA, highlighting certain tools that have potential to detect and critique antipatterns for novice programmers.

2.2.1 Autograders. Autograders have a long history dating back to at least 1960 [12]. These systems often include a testing component that provides a rigorous shakedown of the student work to expose compilation and execution errors.

Some autograder systems, like *Marmoset* [18] and *JUG* [3], as well as professional analysis tools like *JExercise* [19], are entirely test-driven and do not examine source code. A sole focus on program behavior addresses our antipattern concerns only partly. First, error messages from black-box test results do not necessarily give students insight into what is wrong with their code. Second, some antipatterns may manifest themselves via improper programming practices but do not have any effect on program behavior; these “code smells” [10, 15] cannot be detected through unit testing. To address these issues, instructors need to draw on their pedagogical and domain knowledge to guide novices in a meaningful way.

The following autograder systems include facilities for analysis and feedback on source code, and therefore show promise in providing the kind of feedback on antipatterns that WebTA currently provides.

- *Web-CAT* [8]. Web-CAT is highly configurable due to its modular architecture. Plugins for common IDEs allow its use while students are coding. It works with a variety of programming languages. Web-CAT grades assignments and provides feedback to the student based on both instructor and student provided test cases. Instructors can plug in their own constraints and triggers for feedback. This can be very helpful in addressing the kinds of antipatterns we are discussing in this paper.
- *BOSS* [14] is a course management system that originated as an autograder. BOSS includes among its features, a database and file-independent storage system, a submission and testing framework, and a student front-end. BOSS checks runtime behavior, code style, and detects plagiarism.

2.2.2 Critiquers. Critiquers are similar to autograders, but they focus on providing highly interactive and targeted feedback to student programmers [1]. While these systems often perform testing, they make strong use of the instructor’s domain knowledge to identify patterns and formulated meaningful responses. This makes them well-suited for providing novices with the kinds of feedback we are discussing in this paper. WebTA falls into the category of automated critiquer, along with the following:

- *Test My Code (TMC)* [20] 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

```

❗ Error:(4, 4) java: illegal start of type
❗ Error:(4, 13) java: ')' expected
❗ Error:(4, 14) java: illegal start of type
❗ Error:(4, 15) java: <identifier> expected
❗ Error:(4, 16) java: ':' expected
❗ Error:(4, 22) java: > expected
❗ Error:(4, 34) java: <identifier> expected
❗ Error:(4, 36) java: illegal start of type
❗ Error:(4, 37) java: <identifier> expected
❗ Error:(4, 39) java: ':' expected
❗ Error:(5, 14) java: illegal start of type
❗ Error:(5, 17) java: ':' expected
❗ Error:(5, 18) java: invalid method declaration; return type required
❗ Error:(5, 26) java: <identifier> expected
❗ Error:(5, 27) java: ':' expected
❗ Error:(7, 4) java: class, interface, or enum expected
❗ Error:(8, 1) java: class, interface, or enum expected

```

Figure 1: Errors normally associated with misplaced code.

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.

- *Java Critiquer* [16] 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.
- *JDeodorant* [9] is an Eclipse plugin that detects several classic code smells [15] 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 ANTIPATTERNS: CHARACTERISTICS, DETECTION, AND RESPONSE

Examination of student code submitted to WebTA has helped us identify several code antipatterns, which we can then detect automatically and provide appropriate just-in-time feedback to students. The examples given here are derived from actual student submissions, slightly modified for brevity.

3.1 Misplaced Code

While learning to program in Java, beginning students are focused at the level of procedural code, working on individual expressions and statements; the larger object-oriented structures of classes and methods mean little or nothing to them, leading them to neglect the necessary enclosing “boilerplate”. This leads to the *Misplaced Code* antipattern: 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 (Fig. 1).

We have tailored our syntactic analysis to bundle up such illegal code fragments in such a way as to allow further analysis to continue. Thus we can provide feedback about the code even if it is misplaced. For example, in Listing 3 there is a misplaced sequence of statements; the student has plunged into the details of implementing the assignment without declaring an enclosing

Listing 3: Misplaced Code

```

1 public class Hello {
2     String s = "Hello_World";
3     String result = "";
4     for(int i=0; i < s.length(); i++) {
5         result = s.charAt(i) + result;
6     }
7     return result;
8 }

```

method. WebTA detects the code fragment, bundles it as a unit separate from the rest of the abstract syntax tree, and determines that if the bundled code were contained within a method it would be syntactically correct. So we can suggest to the student:

Your code statements follow the rules of the Java language, but they're not in the right place. In Java, all statements must belong to a method, but your statements are not inside any method declaration. (For tips on how to make a method declaration, [click here](#).)

3.2 Pseudo-Implementation

The *Pseudo-Implementation* antipattern occurs when 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. A similar issue arises for abstract classes wherein students develop code in a new class, neglecting to extend the abstract class. In some cases, students only implement a subset of the methods contained in the abstract class. In other cases, students copy the encapsulated code into their own class and avoid inheritance.

An example is shown in Listing 4; the student has declared a class `Reverse` that implements the single method `reverseString` required by the interface `ReverseInterface`, but has failed to explicitly tag the class as implementing the interface.

This is a particularly vexing problem for students as the code will often run correctly on their machine where they create an instance of class `Reverse` and store it in a variable of the same type. However, this often breaks in instructor test cases where an instance of `Reverse` is likely to be stored in a variable of type `ReverseInterface`.

Detecting this pattern can be achieved in a couple of ways:

- **Statically:** By traversing the Abstract Syntax Tree, the pseudo-implemented class can be inspected to determine if it matches the method signatures declared in the interface.
- **Dynamically:** Instructors can include a simple `instanceof` test case in their test suite, to ensure that an instance of the class is indeed also an instance of the interface. (Listing 5).

3.3 Localized Instance Variable

In the *Localized Instance Variable* antipattern, students declare an instance variable but only use it as if it were a local variable in a single method. Often, as in Listing 6, these variables are both modified and used within the method.

Listing 4: Interface Pseudo-Implementation

```

1 public interface ReverseInterface {
2     public String reverseString( String s );
3 }
4
5 // Reverse implements ReverseInterface
6 public class Reverse {
7     // required by ReverseInterface
8     public String reverseString( String s ) {
9         String result = "";
10        for(int i = 0; i < s.length(); i++) {
11            result = s.charAt( i ) + result;
12        }
13        return result;
14    }
15 }

```

Listing 5: one way to detect the Interface Pseudo-Implementation pattern.

```

1 @Test
2 public void testImplementsInterface( )
3     ↪ throws FileNotFoundException {
4     Reverse test = new Reverse( );
5     if ( !(test instanceof
6         ↪ ReverseInterface) ) {
7         fail( "Your_Reverse_class_does_not_
8             ↪ implement_ReverseInterface."
9             ↪ );
10    }
11 }

```

Listing 6: Localized Instance-Variable

```

1 public class Reverse {
2     String result = "";
3     public String reverseString( String s ) {
4         for(int i = 0; i < s.length(); i++) {
5             result = s.charAt( i ) + result;
6         }
7         return result;
8     }
9 }

```

We detect this by slicing the instance variable within the Abstract Syntax Tree. If the variable is not public and is only used in a single method, WebTA suggests that the student make the variable local to the method.

Consider, however, the case where a student defines an instance variable and only supplies a reader method, a writer method, and/or initializes the variable in a constructor (Listing 7). In such cases, where there are no defined behaviors operating on the variable, one has to question whether the data member belongs in the class or if it is perhaps a property of another class.

Listing 7: Instance Variable with no Behavior

```

1 public class Elephant {
2     private int numberPoached;
3     public int getNumberPoached() {
4         return numberPoached;
5     }
6     public void setNumberPoached(int i) {
7         numberPoached = i;
8     }
9 }

```

Listing 8: Empty Knee-Jerk Code

```

1 public double abs( double d ) {
2     double result = d;
3     if ( d < 0 ) {
4         for( int i = 0; i < 10; i++ ) {
5             }
6         result = -d;
7     }
8     return result;
9 }

```

This case can stymie automatic detection because of notable exceptions, such as key-value pairs or the value slot in a linked-list node. This is where the instructor's knowledge of the problem domain is required to whether or not WebTA should report a problem to the student.

3.4 Knee-Jerk

Students in an introductory programming course are bombarded with new terms and programming constructs, and are asked to write code before a deep understanding of the new constructs has settled in. Consequently, it is common for first year students to utilize the *Knee-Jerk* antipattern: implementing a language construct in a vacuous way, simply because it was recently studied in class. Listing 8 shows a simplified example: working on an assignment to implement a method that returns the absolute value of a number, a student includes a for-loop (lines 4-5) because the code structure was covered recently in class.

Often, the body of such knee-jerk code is left empty. This makes it easy to detect using either a regular expression (in the case of static analysis of the code) or by walking the abstract syntax tree representing this method. Upon detecting empty knee-jerk code, we highlight the fragment and ask the student why it is included in their code.

Somewhat more challenging is the case when code is provided within the body of the knee-jerk fragment (Listing 9 lines 4-6). In the special case where the body code does not contribute to the outcome of the method, this pattern may be detected by slicing the dependency graph of the outcome of the method. If the code does not affect the result of the method, the fragment is highlighted and the student is advised that the code may be unnecessary.

Listing 9: Independent Knee-Jerk Code

```

1 public double abs( double d ) {
2     double result = d;
3     if ( d < 0 ) {
4         for( int i = 0; i < 10; i++ ) {
5             System.out.println( i );
6         }
7         result = -d;
8     }
9     return result;
10 }

```

Listing 10: Entangled Knee-Jerk Code

```

1 public double abs( double d ) {
2     double result = d;
3     for( int i = 0; i < 10; i++ ) {
4         if ( d < 0 ) {
5             result = -d;
6         }
7     }
8     return result;
9 }

```

An open problem for us, can occur when the knee-jerk code fragment includes valid code that impacts the result of the method (Listing 10 lines 3-7). In this example, the student write a for-loop around the internal logic of the method. The loop has no real effect and method still produces a valid result. A human reading this code can easily identify the loop as unnecessary. Here the instructor can use specific knowledge about the assignment to identify and flag code that matches this pattern.

In general, instructors can be mindful of knee-jerk coding behavior and anticipate places where new concepts may be used inappropriately. For instance, methods that can be easily implemented by straight-line code can be tagged so as to trigger a critique when conditional or iterative constructs appear.

3.5 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. Consider the code in Listing 11, where the variable declaration that instantiates a new Scanner object is inserted within the body of a loop. Execution of this loop results in multiple instantiations of Scanner objects, contrary to the intentions of the student. A more appropriate solution is shown in Listing 12, with the Scanner declaration and initialization outside of the loop.

The Repeated Resource Instantiation antipattern can be seen as a form of Knee-Jerk antipattern. Students are taught how to read keyboard input with the Scanner class in their first lab, before they know about conditional statements or loops. Until they encounter an example with piped input or file I/O, students have no reason to

Listing 11: Repeated Resource Instantiation

```

1 for (int i; i < 10; i++) {
2     Scanner input = new Scanner( System.in );
3     System.out.println( input.next( ) );
4 }

```

Listing 12: Repeated Resource Instantiation (solution)

```

1 Scanner input = new Scanner( System.in );
2 for (int i; i < 10; i++) {
3     System.out.println( input.next( ) );
4 }

```

consider issues beyond manual keyboard input. Thus the Scanner declarations that they see in their initial straightline code becomes in essence a rote formula that they replicate without any reflection, and the consequences of the placement of the resource instantiation are not clear to them.

This example also exhibits the *Variable Declared In Loop* antipattern, shown more simply in Listing 13. Interestingly, this status of this antipattern is sensitive to the programming language in question. A Java expert would typically declare a variable in the smallest scope; this is legitimate because the JVM Specification dictates that the compiler declare stack entries prior to entering a loop; thus implying a move of the declarations outside the loop. But this is not true for all languages, and keeping the variable declaration within the loop may amount to an unnecessary repetition of resource allocation. In general, reallocation of space for declared variables upon each iteration is inefficient and wasteful—although easy for a compiler to optimize away. Since we want to teach our students the best general practices, we characterize the code in Listing 13 as an antipattern and advocate for a solution in Listing 14.

The effects of Repeated Resource Instantiation can be subtle. The code in Listing 11 is syntactically correct and, when tested with keyboard input, appears to execute as intended. Yet problems occur when the input is piped to standard input from another source. When this happens, the piped data is queued in the buffer of the instantiation of the first iteration—only to be overwritten by the instantiation of the second iteration. Thus the code only retrieves one iteration worth of data, then blocks, waiting for keyboard input.

Detecting Repeated Resource Instantiation is sometimes context dependent. An instructor can specify a particular resource to track, then WebTA looks for sequences of code containing multiple copies of code that instantiates the resource. Alternatively, WebTA can traverse the abstract syntax tree looking for instantiation of the resource within the body of a loop. Some open questions include: How deeply should we traverse the abstract syntax tree? If the resource instantiation is contained within a method and the method is called within a loop, does the code match the pattern? When do we consider the expertise of the student? A novice will be surprised by the side-effects of Repeated Resource Instantiation, whereas an expert could capitalize upon them. For now, we rely on the instructor’s knowledge of the students and assignment to guide the detection of this pattern.

Listing 13: Variable Declared in Loop

```

1 for (int i; i < 10; i++) {
2     int x = i * i;
3     System.out.println( x );
4 }

```

Listing 14: Variable Declared in Loop (solution)

```

1 int x;
2 for (int i; i < 10; i++) {
3     x = i * i;
4     System.out.println( x );
5 }

```

4 CONCLUSION

WebTA currently detects and reports on approximately 200 antipatterns, including those described in §3. Many, such as violations of style guidelines, are relatively simple to detect, while others require more subtle analysis and in some cases knowledge of the particulars of the assignment. These antipatterns typically do not appear in professional or academic software engineering texts because they are not patterns that seasoned developers would produce. Aware of the misconceptions or practices they can fall into, we can provide students feedback through WebTA that is more focused, more appropriate to their level of understanding, and less intimidating than what they would encounter in a more traditional development environment.

Apart from the challenge of automated detection, the concept of early programming antipatterns is of interest in its own right, and we are continuing to mine student submissions for new entries in our antipatterns library. We plan to use WebTA to track the frequency of antipatterns and determine if antipattern production declines with use of WebTA, comparing to earlier work. We can then bundle our library of known antipatterns and provide instructors with instructions on how to detect and respond to them with appropriate feedback for their students. Continued and broader use of WebTA will provide us with the data needed to confirm the effectiveness of our automated critique method.

Moreover, we have designed WebTA to be extensible, so that other instructors can define their own antipatterns and means of detection and feedback. Our library of antipatterns will be made publicly available as part of our WebTA distribution. We hope that this perspective on novice programming can inspire more meaningful and appropriate critique for students in this high-risk period of their computing careers.

REFERENCES

- [1] N. M. Ali, J. Hosking, and J. Grundy. 2013. A taxonomy and mapping of computer-based critiquing tools. *IEEE Transactions on Software Engineering* 39, 11 (2013), 1494–1520.
- [2] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective compiler error message enhancement for novice programming students. *Computer Science Education* 26 (2016), 148–175. Issue 2-3.
- [3] Christopher Brown, Robert Pastel, Bill Siever, and John Earnest. 2012. 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*. New York, NY, USA, 99–104.
- [4] Neil Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education* 17 (2017). Issue 2.
 - [5] William J. Brown, Raphael C. Malveau, Hays W. “Skip” McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.
 - [6] John DeNero, Sumukh Sridhara, Manuel Pérez-Quinones, Aatish Nayak, and Ben Leong. 2017. Beyond autograding: Advances in student feedback platforms. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, ACM, NY, NY, 651–652.
 - [7] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4.
 - [8] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. 328–328.
 - [9] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: Identification and application of extract class refactorings. *Proceedings of the 33rd International Conference on Software Engineering* (2011).
 - [10] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
 - [11] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the ACM Conference on International Computing Education Research*. 160–168.
 - [12] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (Oct 1960), 528–529.
 - [13] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 86–93.
 - [14] M. Joy and N. Griffiths. 2004. Online submission of coursework—A technological perspective. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*. 430–434.
 - [15] Robert C. Martin. 2009. Smells and heuristics. In *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Chapter 17.
 - [16] L. Qiu and C.K. 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.
 - [17] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15 (2005), 203–221. Issue 3.
 - [18] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proceedings of the 11th Annual ACM Conference on Innovation and Technology in Computer Science Education*. 13–17.
 - [19] Hallvard Trætteberg and Trond Aalberg. 2006. JExercise: A specification-based and test-driven exercise support plugin for Eclipse. *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange* (2006).
 - [20] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students’ learning using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. 117–122.
 - [21] Stelios Xinogalos. 2015. Object-oriented design and programming: An investigation of novices’ conceptions on objects and classes. *ACM Transactions on Computing Education* 15 (2015). Issue 3.