Critiquing Antipatterns In Novice Code

Leo C. Ureel II
Michigan Technological University, ureel@mtu.edu

CRITIQUING ANTIPATTERNS IN NOVICE CODE

By

Leo C. Ureel II

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computational Science and Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2020

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computational Science and Engineering.

Department of Computer Science

Dissertation Advisor:	*Dr. Charles Wallace*
Committee Member:	*Dr. Linda Ott*
Committee Member:	*Dr. Robert Pastel*
Committee Member:	*Dr. Adam Feltz*
Department Chair:	*Dr. Linda Ott*

Dedication

To Dad

who knew I would make it this far.

Contents

List of Figures

Listings

List of Tables

Acknowledgments

No one can succeed alone. I have been blessed to have been helped along the way by many people. Elaine Eikenberry, our offspring Miriam and Neil were more than supportive and patient through a process that has spanned more than a decade. My parents. My Dad would have been very proud to see me graduated.

I have had the good fortune to have convinced Dr. Charles Wallace to be my advisor. I have been mentored by many academics through the years: Dr. Linda Ott, Dr. Robert Pastel, Dr. Adam Feltz on my committee. Others include Dr. John Lowther, Dr. Anne Wysocki, Dr. Ken Forbus, Dr. Chris Riesbeck, Dr. Larry Birnbaum, Dr. Laura Brown, Dr. Jean Mayo.

My close compatriot Briana Bettin and many friends John Welch, Chris and Kyle Pellar Kosbar, Madeleine Usher, Mike Sobocinski. And the many who have worked with me through the years, especially James Rudlaff who put long hours into the AST code, Scott Pomerville, Madeleine Howard, Sam Wallace, and Sarah Larkin who joined Briana and myself for lively summer research discussions.

This work builds the works of Dr. Robert Pastel and Dr. Chris Brown on his JUnit Generation (JUG) and Autograder tool at Michigan Technological University [9, 11], and the work by Dr. Chris Riesbeck and Dr. Lin QiU at Northwestern University [45, 46] Undergraduate researchers James Rudlaff and Sam Wallace contributed original ideas toward the development of the Abstract Syntax Tree and the Pseudocode Representation components of WebTA.

This research has been supported by a Jackson Blended Learning Grant, NSF IUSE Grant, and an ICC Seed Grant.

Abstract

Students in introductory computer science courses, are learning to program. Indeed, most students perceive that learning to code is the central topic explored in the courses. Students spend an enormous amount of time struggling to learn the syntax and understand semantics of a particular language. Instructors spend a similar amount of time reading student code and explaining the meaning of the cryptic error messages displayed by compilers.

Messages provided by compilers are intended to give feedback on the adherence of one's code to the language specification and conventions. Unfortunately, these message are geared towards experts who have a clear understanding of the language syntax and semantics and a deep model of what comprises a program and how a program is developed. These students are novices who lack fundamental understanding of the structure of a program and have no basic mental model of how a program works. Novices make different kinds of mistakes than experts. Instructors need to spend a lot of time simply assisting novices in using compilers and understanding their output.

In addition to mastering the syntax and semantics of their first programming language, novices are exposed to the question of what constitutes good design. Instructors can identify virtuous design choices and articulate areas of improvement. But contact time with students is limited, and waiting for in-person feedback or replies to personal messages can be a critical delay.

Novices, still struggling to use the compiler, have not yet developed the sophisticated analytical processes employed by experts and this is reflected in their design choices and the kinds of mistakes they make. When a novice approaches an instructor with a question, the instructor must often provide a balanced critique that assists the student with understanding both the structure and the design aspects of their own code.

My research has focused on whether we can identify examples of early programming antipatterns that have arisen from our teaching experience, and describe different ways of detecting them automatically. Novice students may produce code that is close to a correct solution but contains syntactic errors; code critiquers attempt to salvage the promising portions of the students submission and suggest repairs in ways 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, code critiquers can detect and flag these flaws. Finally, certain types of antipatterns can be anticipated and flagged by the instructor, based on the context of the course and the programming activity; code critiquers allow for customizable critique triggers and messages.

This dissertation presents several key contributions to our understanding of novice misconceptions and their representation, diagnosis and repair using antipatterns. My research focuses on identifying antipatterns and detecting them in novice code, then using this information to provide the student with a meaningful critique of their work. I have developed WebTA, a tool to critique student programs in introductory computer science courses. WebTA is used to teach students test-driven agile development methods through small cycles of teaching, coding integrated with testing, and immediate feedback.Through the use of WebTA in introductory computer science courses since 2014, I have amassed a significant corpus of novice programmer submission data. Lastly, I have compiled a library of antipatterns found in novice code.

Chapter 1

Introduction

Many automatic approaches have been developed for evaluating student programs [18, 20, 30]. Often these automated assessment tools take the form of testing scripts or unit tests designed to support the instructor in testing and grading programs submitted by large numbers of students. Such tools are generally called *Autograders*.

Autograder technology may also be applied to support agile design processes and provide students with critical feedback on their code and design processes. Software that focuses on identifying structures and behaviors that indicate working code and good design then providing feedback to the student are known as code critiquers. According to agile design principles, the primary measure of a design is working software [27]. Timely communication is critical to the agile design process. Code critiquers provide this communication in the absence of the instructor. Furthermore, an essential element of agile design is reflection with an eye towards learning and improvement. Code critiquers provide feedback to students when they need it so they can reflect on the patterns found in their code in order to learn about programming and develop better programs.

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.

The aim of our automated code critique tool WebTA is to provide feedback to students during development. WebTA acts as an assistant to the student, commenting on intermediate code iterations during development, and then as a traditional automated grader for final submissions. 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 student practices that reflect some misunderstanding, early in the process. Through our experience as instructors, we can anticipate recurring practices that indicate misunderstanding or lack of care (antipatterns [13]) and plan accordingly.

Some antipatterns cut across traditional boundaries of syntactic well-formedness or behavioral correctness. For instance, 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. Also, some antipatterns do not result in compile-time errors and have no effect on behavior, thus are not detectable through testing. Furthermore, 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 (§I) an overview of related work, (§II) a working description of our strategies for identifying antipatterns and generating code critiques for different aspects of programming, (§III) an examination of a production code critiquer, WebTA, the tool used to conduct this research and a discussion of the corpus of student submissions that was compilerd through the use of WebTA, (§IV) we then present a library of common antipatterns in student code, and finally, (§V) we conclude with a summary of contributions and a look to future work.

Part I

Background

Chapter 2

Situating the Work

2.1 Motivation

Code critiquers provide critique student code in much the same way that instructors and their teaching assistants (TAs) respond to student programs.

What does an Instructor/TA really do? Based on experience gained over time, the instructor/TA looks for antipatterns in student work, recognizes them, and calls them out. These antipatterns are commonly occurring practices that reflect misunderstandings or poor design choices made by beginning programmers.

What makes the Instructor/TA different from existing programming tools? The instructor combines pedagogical experience with expertise in computing to provide comments on coding issues unique to novices. Most development software is designed for experts. The support and assistance provided by these tools operate on a higher level than students in introductory computer science courses

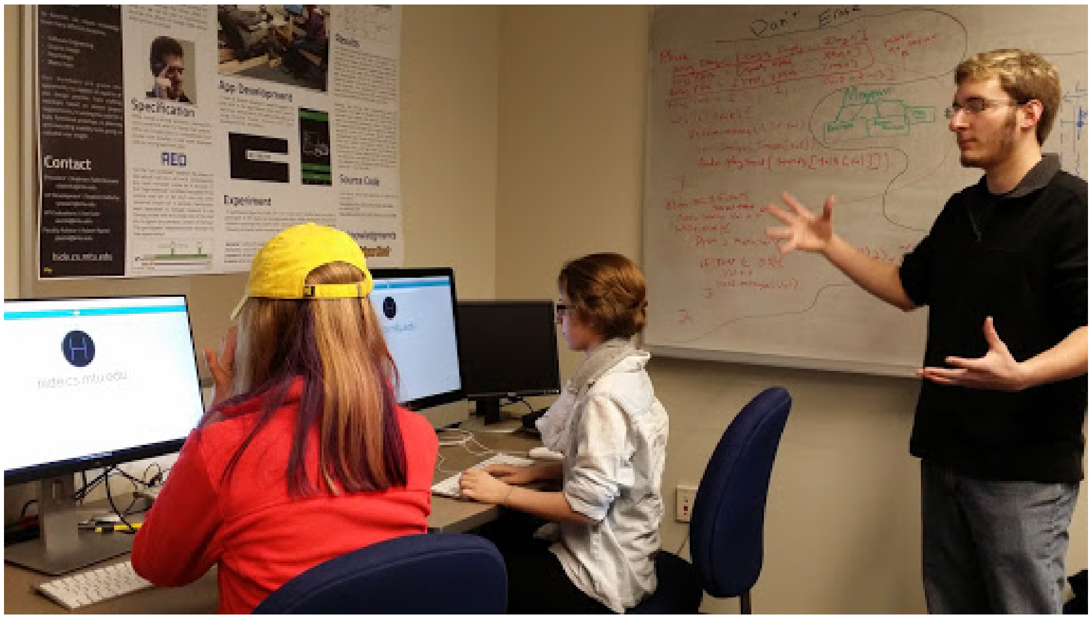


Figure 2.1: What does an Instructor/TA really do?.

can reach. Instructor comments, however, are quite different from what a compiler would tell the student. The instructor meets the students at their level to provide meaningful critiques on coding mistakes that experts would never make.

Figure 2.2: Instructors can't provide individual attention in large classrooms.

Unfortunately, instructors cannot critique every student's code in large classroom settings and instructors are unavailable during late-night study sessions to help students develop good coding practices. Code critiquers automate the critiquing process; capturing the essence of instructor-student interaction and focusing on novice antipatterns. This motivates the need for software that can provide feedback on student code when they meed it most, during development.

2.1.1 Cognitive Apprenticeship

The Cognitive Apprenticeship model [14, 15, 16, 17] is a constructivist approach to learning that focuses on teaching concepts and practices utilized by experts to solve problems in realistic environments.

Constructivism is a theory for teaching and learning whereing learners construct knowledge rather than passively absorbing information. It is the student's ability to experience and reflect upon the world that enables them to construct mental models and incorporate new information into their previous knowledge. Instructors and code critiquers support constructivist learning by providing feedback for reflection while the student is engages in the experience of programming.

Cognitive Apprenticeship has special relevance in the context of software development because it emphasizes making implicit processes explicit to the learner. In typical computer science or software engineering educational settings, topics like design are often deemphasized in favor of more technical topics, such as syntax; in the workplace, the design-related knowledge that experienced developers possess is internalized, complicating their ability to pass it along to new employees.

Vihavainen and Luukkainen [58, 59] utilize Extreme Apprenticeship (XA), which, in the context of software engineering education, builds on cognitive apprenticeship. The focus of XA in the classroom is transforming the student into an expert by emphasizing learning-by-doing and starting early. This is accomplished primarily through modeling activities and scaffolding. Scaffolds are correctly-timed hints and feedback. Instructors emphasize *deliberate practice* and students complete many small programs during the course. The deliberate effort of programming on a day-to-day basis emphasizes that students are becoming professionals.

Using the cognitive apprenticeship model, code critiquers provide the same kinds of critiques that instructors do. The student receives the the critiques while engaged in authentic programming tasks. The critiques call out antipatterns in the student's code and indicate best practices that can mitigate or repair the issues. Students can then reflect upon the code critiques and incorporate this new information into their own mental models of programming.

Chapter 3

Automated Assessment of Programming Assignments

3.1 Automated Assessment of Programming Assignments

Many automatic approaches have been developed for grading student programs. Often these automated assessment tools take the form of testing scripts designed to support the instructor in testing and grading programs submitted by large numbers of students. However, automated assessment can also be used to support agile design processes and provide students with feedback.

Ala-Mutka [1] describes the advantages of automated assessment as “speed, availability, consistency and objectivity” but warns that “automated tools emphasize the need for careful pedagogical design of the assessment solutions”.

Ala-Mutka discusses many features of automated assessment approaches. Program characteristics assessed dynamically include functionality and efficiency. In addition, static analysis tools can be used to assess program characteristics such as adherence to coding conventions, coupling of modules, and cyclomatic complexity.

There are three main categories of automated assessment tools that have influenced the design of WebTA; Autograders, Submission and Grading Systems, and Critiqer Systems. These are described in the following sections.

3.1.1 Autograders

By definition autograder systems provide students with feedback on their code; albeit sometimes this is only a score with little explanation. As the name indicates, autograders are primarily instructor-facing; the underlying model is that students use their own methods to test and debug their code, then the autograder passes judgment on their final effort. The primary motivation underlying autograder systems is saving instructors time in grading, rather than providing feedback to students, which is our goal in developing code critiquers. Autograders have a long history dating back to at least 1960 [29]. Some autograder systems, like *Marmoset* [55] and *JUG* [10], as well as professional analysis tools like *JExercise* [56], are entirely test-driven and do not examine source code. Such systems can only detect behavior-based antipatterns.

The Marmoset Project [54]. A submission and testing system, students can use Marmoset from either the command line or an IDE plugin. Student submission are subjected to four different kinds of tests (Student tests, Public tests, Release tests, and Secret tests), then given feedback based on the test results.

JUnit Generation (JUG) Autograder [9, 11]: One of the primary influences on the work provided here is the JUnit Generation (JUG) Autograder [10, 11]. JUG was developed to support instructors in grading and providing feedback on homework assignments in a data structures course. It provides supplemental feedback to improve student learning, and ease of use for the assignment developer and grader. The JUG system combines unit testing, evaluation and reporting to fulfill those goals. JUG performs dynamic assessment of functionality through generated JUnit testing and efficiency through measurements of execution time. JUG influenced the dynamic assessment portion of our application that evaluates student code by applying a battery of instructor developed JUnit tests to the student's code.

3.1.2 Grading with analysis and feedback

More relevant to our work are autograder systems that include facilities for analysis and feedback on source code.These include the following:

Web-CAT [21]. 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 [**32**] 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.

3.1.3 Critiquer Systems

Critiquers are similar to autograders, but they focus on providing highly interactive and targeted feedback to student programmers [4]. 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.

Java Critiquer [**46**]**:** Another major influence on our work is Qiu and Reisbeck's Java Critiquer [45, 47, 48, 49], developed to teach students how to write clean, maintainable and efficient Java code. Java Critiquer provides individualized feedback and just-in-time learning opportunities to students. The Java Critiquer 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. When a match is found in the student's code, the rule is fired and the advice encoded therein is dispensed to the student. Students use the tool iteratively to improve their code before submission. Ali, Hosking, and Grundy [3, 5] categorize the Java Critiquer as an analytic critic, using text-based pattern matching through instructor authored rules to provide explanations and suggestions. The Java Critiquer influenced the static assessment portion of our application that evaluates students' programming style.

Test My Code (TMC) [60]: Test My Code (TMC) is an automated assessment tool developed by Vihavainen *et al.* [60] as a tool for their Extreme Apprenticeship methodology. TMC is an assessment system that enables instructors to build scaffolding into programming exercises. TMC is integrated into the student's programming environment and provides tasks for the student to work on. TMC allows for scaffolding and automated instructor-initiated feedback.

Test My Code is very close to the overall goals and functionality of our project. One important difference in focus is that TMC 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. In contrast, the personnel constraints of our teaching environment offer less intensive interplay between student and instructor. We appreciate TMC's ability to facilitate direct communication between student and instructor, but our focus is on providing automated feedback to students that are detectable through dynamic or static methods, as anticipated by the instructor. Consequently, we use automated methods for style and design feedback in addition to test-related feedback.

JDeodorant [26] is an Eclipse plugin that detects several classic code smells [37] 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.

Chapter 4

Patterns and Antipatterns

4.1 Patterns and Antipatterns

4.1.1 Patterns

Experienced programmers focus on solving problems with algorithms, interfaces, and inheritance hierarchies. They know that a good design "should be specific to the problem at hand but also general enough to address future problems and requirements." [28] They reflect on their solutions; revisiting and refactoring the code many times even as the software is in use. Expert programmers build a library of reusable code over time that forms the foundation for future work.

Patterns are common solutions, reified in code, that are used repeatedly in expert programs. Use of these patterns makes our programs flexible, elegant, and reusable. The original inspiration for the use of design patterns in computer science comes from a treaties on architecture (designing buildings) by Christopher Alexander. He wrote, “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.” [2] Although they are reified as code when applied, the pattern represents a general approach that can be implemented in a variety of ways, dependent on the context of use.

Gamma et. al. describe patterns as “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” [28] Gamma describes patterns using four elements: the pattern name, the problem, the solution, and the consequences.

An example of a good programming pattern is the implementation of *Visitor*, which represents an operation to be performed on the elements of a data structure. The Visitor lets you change the behavior applied to the elements without changing the solution code.

For example, in Listing 4.1, we depict the code for an inorder traversal of a binary tree. In the algorithm for inorder traversal, the method recurses to the left child of the current node, effectively processing all of the nodes in the entire left subtree. Then the current node is processed, i.e. visited. Finally, the algorithms recursively processes the right child of the current node, effectively processing the entire right subtree.

It is the processing of the current node that defines the Visitor Pattern. A novice programmer might hard-code the way the node is processed into the inorder traversal itself. However, doing so would result in re-implementing the inorder traversal algorithm for every problem solution that required a different way of processing the nodes in a tree.

In the Visitor Pattern, the inorder traversal algorithm is coded once and the code snippet used to process each node is passed in as an argument (bundled as a lambda expression.) This allows the programmer to make use of the general traversal code while tailoring the way each node is processed to the required solution.

Listing 4.1: The Visitor Pattern

```java
// The Visitor pattern
public interface Visitor<E> {
    public void visit( E element );
}

// Application of the Visitor pattern
// to perform an operation on each element
// in a binary tree.
public <E> void inorderTraversal( Node<E> node,
                                   Visitor<E> visitor ) {
    if ( node == null ) return;
    inorderTraversal( node.getLeftChild( ), visitor );
    visitor.visit( node.getElement( ) );
    inorderTraversal( node.getRightChild( ), visitor );
}

// Use of the visitor to print all elements
inorderTraversal( rootNode, element -> {
    System.out.println( element )
} );

// Use of the visitor to add all elements to a list.
ArrayList<E> list = new ArrayList<>( );
inorderTraversal( rootNode, element -> {
    list.add( element )
});
```

We want to be able to identify good patterns in student code so we can highlight them and reinforce good design habits. At the same time, we want to deemphasize compiler errors and warnings, while still maintaining the compiler as gatekeeper, to place the focus on good code design.

4.1.2 Antipatterns

Antipatterns are commonly seen problem solutions that generate negative consequences [12]. Antipatterns are code structures that look good, and maybe are good within a narrow context, but produce generally bad results. [12] Andrew Koenig described them as "An Antipattern is just like a pattern, except that instead of a solution it gives something that looks superficially like a solution, but isn't one." [35]

While we are concerned with Antipatterns in general, we are more concerned with *novice Antipatterns*; i.e., poorly conceived or erroneous code structures commonly created by novice programmers. Often, these novice Antipatterns represent bad solutions that would never be seen in expert code. For this reason, tools designed to assist expert programmers rarely provide good feedback on these kinds of mistakes.

For example, introductory computer science students often want to create a new long-lived resource, such as a Scanner, for each data item that might obtain its value from user input.(See Listing 4.2)

A Scanner is a resource for reading values from some input source, such as the keyboard, a file, or a website. Every time a programmer creates a Scanner, Java allocates a buffer space in memory to store the data as quickly as it can be read from the input source. This can consume a tremendous amount of memory when a Scanner is created for every single data element. Furthermore, because the Scanners all share the same input source, the first Scanner will gobble-up all the waiting keyboard strokes, leaving the others with no data. The solution to this antipattern is to use a singleton. (Listing 4.3

Listing 4.2: The Scanner Overuse Antipattern

```java
// The Accidental Rebirth Antipattern
// Student creates a new resource (Scanner) for ↩
    ↪ every node in a tree.
public class LinkedBinaryTreeNode<E> implements ↩
    ↪ BinaryTreeNode<E> {
    private Question root;
    private Scanner scan;

    public LinkedBinaryTreeNode() {
        root = new Question("cat");
        scan = new Scanner(System.in);
    }
}
```

Listing 4.3: Correcting the Scanner Overuse Antipattern

```java
// Correcting the Accidental Rebirth Antipattern
// Student retrieves a singleton resource (Scanner↩
    ↪ ) instead of creating new for every node in ↩
    ↪ a tree.
public class LinkedBinaryTreeNode<E> implements ↩
    ↪ BinaryTreeNode<E> {
    private Question root;

    public LinkedBinaryTreeNode() {
        root = new Question("cat");
        // retrieves singleton resource instead of ↩
            ↪ creating new for each node.
        scan = Singleton.Scanner(System.in);
    }
}
```

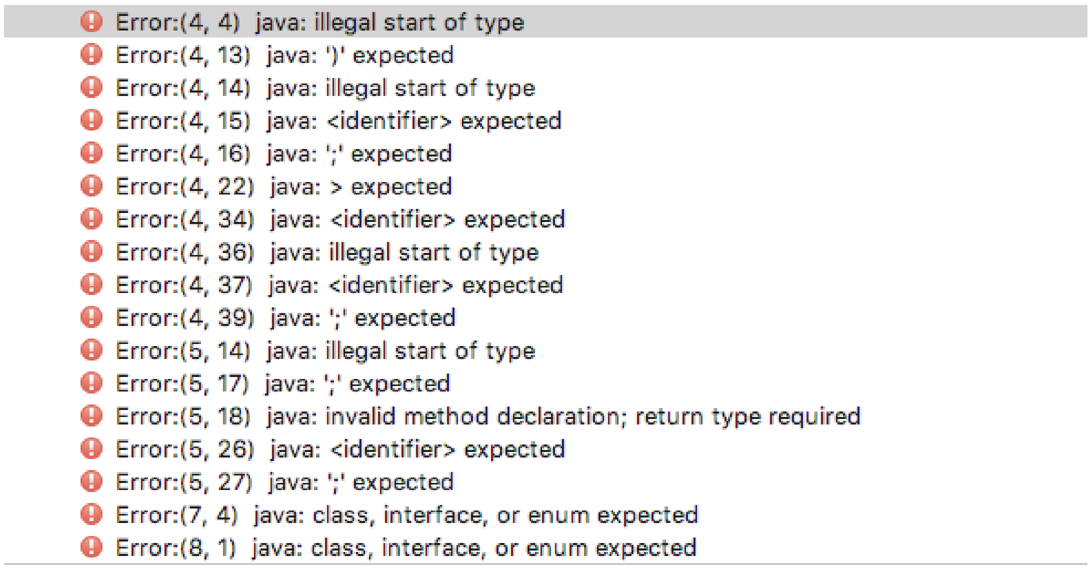
Our goal with WebTA is to identify novice Antipatterns in the student code and provide them with feedback that helps them develop better coding habits. The critiques generated from detected Antipattern tell the students "why the bad solution looks attractive (e.g. it actually works in some narrow context), why it turns out to be bad, and what positive patterns are applicable in its stead." [7]

4.2 Antipatterns: Characteristics, Detection, and Response

WebTA analyzes code snippets and larger programming projects, providing pseudocode translations, feedback on compilation and execution, shakedown testing, and style critique. During analysis, code patterns trigger advice for the students. 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.

4.2.1 Misplaced Code

While learning to program in Java, beginning students are focused at the level of individual expressions and statements, rather than the broader organization of the code. This often leads them 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. 4.1).



```
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 4.1: Errors normally associated with misplaced code.

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 4.4 there is a large code fragment outside of any 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: "*The only code allowed at the class level are variable and method declarations. This code looks good, but needs to be moved into a method.*"

Listing 4.4: Misplaced Code

```java
public class Hello {
   String s = "Hello World";
   String result = "";
   for(int i=0; i < s.length(); i++) {
      result = s.charAt(i) + result;
   }
   return result;
}
```

4.2.2 Interface Pseudo-Implementation

When developing a library to be used by other programmers, Java provides a mechanism for specifying the promise of behavior as a contract between programmers.

The *Interface 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. The following simplified example shows how a student might implement a class in such a scenario. 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`.

It is important to detect this pattern early on to mitigate student frustration. Detecting this pattern can be achieved by traversing the Abstract Syntax Tree for the class to determine if it meets the specification. Alternatively, and perhaps more easily, instructors can include a simple `instanceof` test case in their test suite (Listing 4.6).

Listing 4.5: Interface Pseudo-Implementation

```java
public interface ReverseInterface {
   public String reverseString( String s );
}

// Reverse implements ReverseInterface
public class Reverse {
   // required by ReverseInterface
```

```
  public String reverseString( String s ) {
    String result = "";
    for(int i = 0; i < s.length(); i++) {
      result = s.charAt( i ) + result;
    }
    return result;
  }
}
```

Listing 4.6: one way to detect the Interface Pseudo-Implementation pattern.

```
  @Test
  public void testImplementsInterface( ) throws ↩
    ↪ FileNotFoundException {
    Reverse test = new Reverse( );
    if ( !(test instanceof ReverseInterface) ) {
      fail( "Your Reverse class does not ↩
        ↪ implement ReverseInterface." );
    }
  }
```

4.2.3 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 4.7, these variables are both modified and used within the method.

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.

Listing 4.7: Localized Instance-Variable

```java
public class Reverse {
  String result = "";
  public String reverseString( String s ) {
    for(int i = 0; i < s.length(); i++) {
      result = s.charAt( i ) + result;
    }
    return result;
  }
}
```

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 4.8). 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 4.8: Instance Variable with no Behavior

```java
public class Elephant {
  private int numberPoached;
  public int getNumberPoached() {
     return numberPoached;
  }
  public void setNumberPoached(int i) {
     numberPoached = i;
  }
}
```

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 determine whether or not WebTA should report a problem to the student.

4.2.4 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 4.9 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.

Listing 4.9: 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 }
```

Somewhat more challenging is the case when code is provided within the body of the knee-jerk fragment (Listing 4.10 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 [43]. 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 4.10: 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 }
```

An open problem for us, can occur when the knee-jerk code fragment includes valid code that impacts the result of the method (Listing 4.11 lines 3-7). In this example, the student writes 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.

Listing 4.11: Entangled Knee-Jerk Code

```
public double abs( double  d ) {
   double result = d;
   for( int i = 0; i < 10; i++ ) {
      if ( d < 0 ) {
         result = -d;
      }
   }
   return result;
}
```

Part II

Constructing Code Critiquers

Chapter 5

Basic Critiquer Design

5.1 A Bit of Design

Part II is presented as the heart of a introductory course in Code Critiquers. Here we discuss the basics of constructing code critiquers. We define code critiquers as software that facilitates learning through the automatic critique of student source code by identifying and responding to antipatterns commonly found in novice source code. Larger systems, such as Web-CAT [22] or BOSS [33], incorporate aspects of Curriculum Management Systems (CMS) including assignment management, online submission system, student grade book, etc. Smaller systems, such as FindSmells [53] are narrowly focused on identify a specific issues within student code.

We are interested in constructing code critiquers with the ability to identify different kinds of antipatterns found in novice code and provide feedback to the student. This chapter outlines the design of several rudimentary tools useful doing just this.

5.1.1 User Story

Each of the critiquer tools we develop will be based on the simple design (Figure 5.1) that the instructor distills their pedagogical and computing experience into descriptions of antipatterns, which the software will use to identify antipatterns in novice code, and critique descriptions, used to formulate feedback for the student.

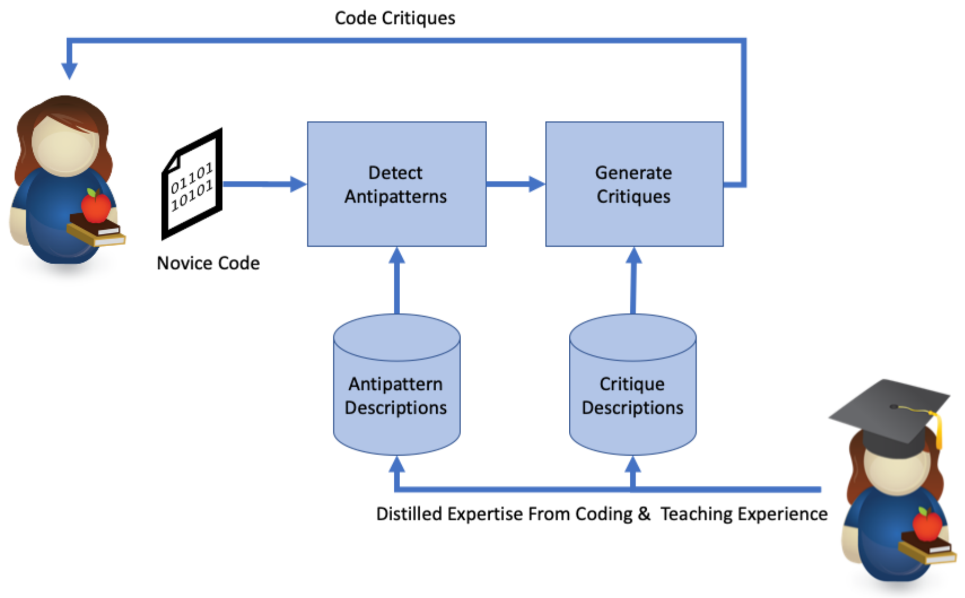


Figure 5.1: Basic design of a Code Critiquer.

Students will supply source code, which the critiquer system will examine in some way to identify antipatterns based on the instructor's antipattern descriptions. Once the antipatterns have been identified, the critiquer system will generate critiques based on the instructor's critique descriptions. These critique descriptions are canned feedback; distilled responses that the instructor would have given the student if the instructor had reviewed the code.

The critiques generated by the critiquer system are then communicated to the student, who reflects upon them and incorporates their advice into the next revision of their source code.

5.1.2 Aspects of Coding

In the following chapters, we will seek to identify antipatterns within five different aspects of coding: Structure, Behavior, Style, Testing, and Design. (Figure 5.2)

- Structure involves the mechanics of the language; the syntax of a program.
- Behavior encompasses the meaning and execution of a program.
- Style invokes community-based standards enabling efficient communication between programmers.
- Testing ensures the program is robust and developed according to spec.
- Design refers to high-level best practices that produce good code.

We will represent these coding aspects as an enumerated data type. (Listing: 5.1)

Listing 5.1: CodingAspect.java - Enumerating Coding Aspects

```java
package edu.mtu.cs.webta.critiquer;

public enum CodingAspect {
  STRUCTURE("Structure" ),
  BEHAVIOR("Behavior" ),
  STYLE("Style" ),
  TESTING( "Testing" ),
  DESIGN("Design" ),
  TBD( "To Be Determined" );
```

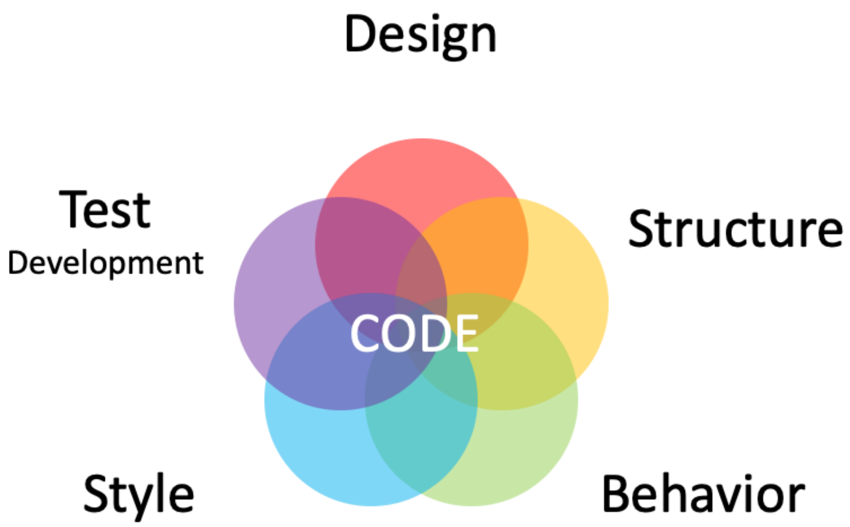


Figure 5.2: Five aspects of coding.

```

  public final String label;

  private CodingAspect( String label ) {
    this.label = label;
  }
}
```

In general, although not exclusively, each of these coding aspects suggests the implementation for a code critiquer. For example, structural aspects of coding, especially at the novice level, are deeply tied to the syntax of the programming language. This suggest that patterns can be identified through parsing or compilation. Whereas the behavior or meaning of code is best captured at runtime. We will look at a different source for identifying antipatterns based on the coding aspect. We can enumerate the different sources in a type (Listing 5.2).

Simple code critiquers for each of these aspects of programming will be explored in the following chapters.

Listing 5.2: AntipatternSource.java - Sources for Identifying Antipatterns.

```java
package edu.mtu.cs.webta.critiquer;

public enum AntipatternSource {
  AST("AST"),
  CODE("Code"),
  DIAGNOSTIC("Diagnostic"),
  EXCEPTION("Exception"),
  TEST("Test"),
  MANUAL("Manual");

  public final String label;
  private AntipatternSource( String label ) {
    this.label = label;
  }
}
```

5.1.3 Describing Antipatterns and Critiques

In our user story, the instructor provides the critiquer system with a description of the antipatterns to be identified. One way to describe patterns is to use *regular expressions*. Regular expressions are equivalent to finite automata and can be used to represent patterns based on common characteristics. Regular expressions are particularly useful for describing patterns that appear in text or, in our case, source code.

Regular expressions can be used to match specific text. For example, we could find all of the print statements in a program using the regular expression `System.out.print`. The power of regular expressions comes from the ability to match patterns where differences between the text exist. For example, we could find every case where a primitive integer variable is declared by using the following regular expression: `^\s*int\s+[a-z_][A-Za-z0-9_$]*\s+=`. In this example, `^` represents the start of a line of text, `\s` represents a single character of whitespace, the `*` means to match the preceding pattern zero or more times and a `+` means the preceding pattern must appear one or more times, the rather long expression `[a-z_][A-Za-z0-9_$]*` describes a variable name in Java (a lowercase letter followed by any combination of letters, digits, underscores, or dollar signs).

So we can find where all of the integer variables are declared in the code, what if we wanted a list of the variables? If we add parentheses around the regular expression for variable names, `^\s*int\s+([a-z_][A-Za-z0-9_$]*)\s+=`, the regular expression engine will remember all of the variable names.

That is interesting, but we want to identify common antipatterns in student code. We can use the regular expression to find all the integer variable declarations where the variable name did not comply with the community standard that variable names must begin with a lowercase letter (Antipattern: §12.4.31): `^\s*int\s+([A-Z0-9_$][A-Za-z0-9_$]*)\s+=`. Here we are looking for variable names that begin with an uppercase letter, a digit, an underscore, or a dollar sign. Note that some of the variable names found using this pattern are legal Java identifiers, for example: this would match the code `int LoanAmout = 5;`. The variable name `LoanAmount` is a legal variable name, but it does not comply with community standard for naming variables in Java.

Table 5.1 lists some of the common meta-characters used in regular expressions and their meaning.

We can now use regular expressions to describe antipatterns (Listing 5.3).

Table 5.1
Common Regular Expression Meta-Characters.

`abc`	Strings of characters `abc` to be matched exactly.
`[abc]`	Match any single character `a`, `b`, or `c`.
`[âbc]`	Match any single character except `a`, `b`, or `c`.
`[a-z]`	Match any single lowercase letter a through z.
`[A-Z]`	Match any single lowercase letter A through Z.
`[0-9]`	Match any single digit 0 through 9.
`.`	(dot) matches any character accept newline.
`^`	Matches The beginning of a string or line.
`$`	Matches EOL or EOF.
`*`	Match zero or more occurrences of the preceding expression.
`+`	Matches zero or more occurrences of the preceding expression.
`?`	Matches zero or one occurrences of the preceding expression.
`( )`	Parentheses are used for grouping information.
`\s`	A single character of whitespace (but not EOL).
`\S`	A single character; not whitespace or EOL.
`\w`	Match a word.
`\W`	Not a word.

Listing 5.3: AntipatternDescription.java - Interface for Antipattern Descriptions

```java
package edu.mtu.cs.webta.critiquer;

import java.util.regex.Pattern;

public interface AntipatternDescription {
  public String getName( );
  public String getDescription( );

  public AntipatternSource getSource( );

  public String getRegexString( );
  public Pattern getRegexPattern( );
}
```

Chapter 6

Critiquing Structure

6.1 The Structure of a Program

The structure of a program is a broad topic that begins, for novices, with mastering the syntax of a language. In addition to syntax, the structure of a program encompasses Structured Programming. Structured programming is a programming paradigm that includes code structures and coding methodology that aims to improve the clarity, quality, development speed, and maintainability of programs. These are important concepts for students to learn. However, we will not be able to identify antipatterns relating to structured programming until we have developed tools for analyzing source code in §8. For now, we will focus on identifying novice antipatterns related to syntax.

Syntax is language dependent and defines the rules for correctly combining symbols into statements and expressions when composing a program. Novices are confused by the rules of syntax and struggle with resolving syntax errors in their code [50]. One reason code critiquers are important tools for addressing novice confusion about syntax is because syntax errors manifest themselves in the moment, while students are coding. Rodrigo et. al. say "*One approach to aiding students in learning to deal with syntactic errors could be to focus more on the process they are learning, helping students to become more self-aware and reflective in their work. However, the frustration and confusion that students experience when programming are emotions and confusion in the moment. To this end, pedagogic approaches that directly support students in meaningful ways while they are engaged in programming (and, therefore, learning) are, we believe, critical.*"

We can use a language parser, either separate from or embedded in a compiler/interpreter to reveal syntax-related structural errors in student code. Our goal is to explain syntactic errors in easy to understand terms, provided debugging techniques that help repair syntax errors, and assist students in developing the knowledge and best practices that help them avoid syntax errors in the future.

One common example of a syntax error that plagues novice programmers learning the Java programming language is the missing semicolon. Java uses semicolons as separators between statements. Indeed, the missing semicolon is part of a more general antipattern MISSING-SEPARATOR (Antipattern §12.2.23). Java has 12 separators (Table §6.1) that can derail the parsing process and produce cryptic error meesages that are difficult for novices to understand.

In this chapter we will develop a critiquer to provide advice to students who encounter a missing separator or other syntax issue in their code.

Table 6.1
Java Separators.

`(`	Left parenthesis - indicates the beginning of an expression, argument list, or parameter list.
`)`	Right parenthesis - indicated the end of an expression, argument list, or parameter list.
{	Left curly-bracket - indicates the beginning of a code block or static initialization environment.
}	Right curly-bracket - indicates the end of a code block or static initialization environment.
`[`	Left square bracket - indicates the beginning of an indexing expression or size expression of an array.
`]`	Right square bracket - indicates the end of an indexing expression or size expression of an array.
;	Semicolon - indicates the end of a statement.
,	Comma - used to separate elements in a static array initialization and to separate assignment operations in variable declarations and for-loops.
.	The dot is used to separate an instance and a variable or method identifier or the packages in a package hierarchy.
...	Ellipsis - used to separate a data type and identifier for use in declaring a method that accepts multiple arguments.
`@`	The commercial-at symbol is used to indicate annotations in Java.
`::`	Double-colon - separates class name from methods name in a lambda expression.

6.2 Detecting Structure Antipatterns

Java provides a convenient API for accessing the compiler at runtime. This allows us to utilize the Java compiler to detect syntax-based structure antipatterns in student code. Listing 6.2 contains the source code to compile a Java source file. The code is relatively straight-forward, but bears examination.

The `compile()` method on line 15 returns true if the compilation is successful or false if compilation failed. There are three arguments required to call the `compile()` method. The first is a `File` object that describes the name and location of the source file. The second is a list of strings representing compiler options. Listing 6.1 provides an example of common compiler options. The third argument is a *Listener* that processes `Diagnostic` objects as they are generated by the compilation process. A `Diagnostic` contains information about any errors or warnings discovered in the code by the compiler. We will collect this diagnostic information and use it to identify syntax-based antipatterns in the code.

Listing 6.1: ArrayList of Compiler Options

```
Iterable< String > options = Arrays.asList(
   "-Xlint:all", // Enable all warnings
   "-Xdoclint:all", // Enable all checks for ↩
      ↪ problems in javadoc comments
   "-Xmaxerrs", "1000", // Set the maximum number ↩
      ↪ of errors to print
   "-Xmaxwarns", "1000", // Set the maximum number↩
      ↪  of warnings to print
   "-Xdiags:verbose", // Select verbose diagnostic↩
      ↪  mode
   "-deprecation", // Output source locations ↩
      ↪ where deprecated APIs are used
   "-source","11", // Provide source compatibility↩
      ↪  with the specified Java SE release.
   "-target","11", // Generate class files ↩
      ↪ suitable for the specified Java SE ↩
      ↪ release.
   "-g", // Generate all debugging info
   "-d", folder, // Specify where to place ↩
      ↪ generated class files
   "-cp", folder, // Specify where to find user ↩
      ↪ class files and annotation processors
   "-sourcepath", "." // Specify where to find ↩
      ↪ input source files
);
```

Digging into the code in Listing 6.2, line 19 retrieves the system compiler object. From this we obtain the standard file manager in line 21. This is used to manage the source file dependencies and compilation order. We seed the file manager with the initial java source file to compile, which was passed to the `compile()` method as the first argument. Beginning in line 31, the code iterates through the files provided by the manager. These files are compiled in a separate thread by compiler on line 32. Finally, if all of the compilation tasks complete successfully, line 42 will return true.

Listing 6.2: CompilerTools.java - Detecting Structure Antipatterns

```
 1 package edu.mtu.cs.webta.util.compile;
 2
 3 import javax.tools.Diagnostic;
 4 import javax.tools.DiagnosticListener;
 5 import javax.tools.JavaCompiler;
 6 import javax.tools.JavaFileObject;
 7 import javax.tools.StandardJavaFileManager;
 8 import javax.tools.ToolProvider;
 9 import java.io.File;
10 import java.util.ArrayList;
11 import java.util.Arrays;
12 import java.util.Collections;
13 import java.util.List;
14 import java.util.Locale;
15
16 public class CompilerTools {
17    public static boolean compile (
18          File sourcefile,
19          Iterable< String > options,
20          DiagnosticListener< JavaFileObject > ↩
             ↪ diagnosticListener
21       ) {
22       JavaCompiler compiler = ToolProvider.↩
          ↪ getSystemJavaCompiler( );
23
24       StandardJavaFileManager manager = compiler.↩
          ↪ getStandardFileManager(
25            diagnosticListener,
26            Locale.ENGLISH,
27            null
28           );
29
30       Iterable< ? extends JavaFileObject > files =
31            manager.getJavaFileObjectsFromFiles(
                ( List< File > ) Arrays.asList( ↩
                   ↪ sourcefile ) );

        boolean result = true;
        for( JavaFileObject file : files ) {
            JavaCompiler.CompilationTask task = ↩
                ↪ compiler.getTask(
                null,
                manager,
                diagnosticListener,
                options,
                null,
                Arrays.asList( file )
              );
            result = task.call( ) && result;
        }
        return result;
    }

    public static List< Diagnostic< ? extends ↩
       ↪ JavaFileObject > > generateDiagnostics(
        String folder,
                String javaClassname,
                        ArrayList<String> options ) {
      File javaSourceFile = new File( folder + "/" +↩
         ↪  javaClassname + ".java" );
      List< Diagnostic< ? extends JavaFileObject > >↩
         ↪  diagnosticsList =
          Collections.synchronizedList( new ↩
             ↪ ArrayList<>( ) );

      options.addAll( Arrays.asList(
          "-Xlint:all",
          "-Xmaxerrs", "1000",
          "-Xmaxwarns", "1000",
          "-Xdiags:verbose",
          "-source","11",
          "-target","11",
          "-g",
          "-d", folder,
          "-sourcepath", folder
       ));

      compile(
          javaSourceFile,
          options,
          diagnosticsList::add
      );

      return diagnosticsList;
    }
  }
```

Continuing with our MISSING-SEPARATOR antipattern (§12.2.23), the code in Listing 6.3 is missing three separators. Can you see where?

Listing 6.3: MissingSeparators.java - Example Containing Antipatterns

```java
public class MissingSeparators {
  private Double volume(String nameOfCurvedSolid, double height, double base, double radius) {
    Double volume = null;
    int [] foo = new int[10
    switch nameOfCurvedSolid.toUpperCase()) {
      case "SPHERE":
        volume = 4 * Math.PI * Math.pow( radius, 2 );
        break;
      case "CYLINDER":
        volume = 2 * Math.PI * Math.pow( radius, 2 ) +
                 2 * Math.PI * radius * height
        break;
      case "CONE":
        volume = Math.PI * radius * ( radius + Math.sqrt(
          Math.pow( height, 2 ) + Math.pow( radius, 2 ) ) );
        break;
    }
    return volume;
  }
}
```

We are now ready to collect diagnostics by compiling *MissingSeparators.java*. Add the `generateDiagnostics( )` method (Listing 6.4) to *CompilerTools.java*. Then call the method with the path and filename for *MissingSeparators.java*.

Listing 6.4: CompilerTools.java - Generate Diagnostics

```java
public static List< Diagnostic< ? extends JavaFileObject > > generateDiagnostics( String folder, String javaClassname ) {
   File javaSourceFile = new File( folder + "/" + javaClassname + ".java" );
   List< Diagnostic< ? extends JavaFileObject > > diagnosticsList =
      Collections.synchronizedList( new ArrayList<>( ) );

   Iterable< String > options = Arrays.asList(
      "-Xlint:all",
      "-Xdoclint:all",
      "-Xmaxerrs", "1000",
      "-Xmaxwarns", "1000",
      "-Xdiags:verbose",
      "-deprecation",
      "-source","11",
      "-target","11",
      "-g",
      "-d", folder,
      "-cp", folder,
      "-sourcepath", "."
   );

   compile(
      javaSourceFile,
      options,
      diagnosic -> {
         diagnosticsList.add( diagnosic );
       }
   );

   return diagnosticsList;
}
```

You should now have a list containing the `Diagnostic` objects (Figure 6.1).

Figure 6.1: Example Diagnostics for MissingSeparators.java.

```
data/library/MissingSeparators.java:4: error: ']' expected
int [] foo = new int[10
                       ^
data/library/MissingSeparators.java:5: error: '(' expected
switch nameOfCurvedSolid.toUpperCase()) {
      ^
data/library/MissingSeparators.java:11: error: ';' expected
         2 * Math.PI * radius * height
                                      ^
```

6.3 Making Critiques from Diagnostics

Several studies have indicated that Novices have trouble understanding the messages produced by a compiler [8, 19, 41, 44, 57]. We use regular expressions to extract salient details from Diagnostics that indicate an antipattern and apply them to critique templates provided by the instructor. The instructor provides the antipattern and critique information in a `Description` (Listing 6.5).

Listing 6.5: Description.java - Combined Antipattern and Critique Description

```java
package edu.mtu.cs.webta.critiquer;

import java.util.regex.Pattern;

public class Description implements ↩
    ↪ AntipatternDescription, CritiqueDescription ↩
    ↪ {
  private String name;
  private String description;

  private CodingAspect type = CodingAspect.TBD;
  private AntipatternSource source;  // AST, Code,↩
      ↪  Diagnostic, Exception
  private boolean useAltSource = false;
  private String regexString = null;
  private Pattern regexPattern = null;

  private String textTemplate = "";
  private String altTextTemplate = "";


public Description( String name,
              CodingAspect type,
              AntipatternSource source,
              boolean useAltSource,
              String description,
              String regexString,
              String textTemplate,
              String altTextTemplate
            ) {
    this.name = name;
    this.type = type;
    this.source = source;
    this.useAltSource = useAltSource;
    this.description = description;
    this.regexString = regexString;
    regexPattern = Pattern.compile( regexString, ↩
       ↪ Pattern.MULTILINE );
    this.textTemplate = textTemplate;
    this.altTextTemplate = altTextTemplate;
  }

  @Override
  public String getName( ) {
    return name;
  }

  @Override
  public String getDescription( ) {
    return description;
  }

  @Override
  public CodingAspect getType( ) {
    return type;
  }

  @Override
  public AntipatternSource getSource( ) {
    return source;
  }

  public boolean useAltSource ( ) {
    return useAltSource;
  }

  @Override
  public String getRegexString( ) {
    return regexString;
  }

  @Override
  public Pattern getRegexPattern( ) {
    return regexPattern;
  }

  @Override
  public String getTextTemplate( ) {
    return textTemplate;
  }

  @Override
  public String getAltTextTemplate( ) {
    return altTextTemplate;
  }
}
```

The result of matching up a `Diagnostic` with a `Description` is a `Critique` (Listing 6.6). The `Critique` pulls together the name (same as the matched `Description`), the matched `Description`, the source (in this case a `Diagnostic`), the matched text (which is group(0) from the regular expression match, the text of the critique and alternate text.

Listing 6.6: Critique.java - Critique Class

```
package edu.mtu.cs.webta.critiquer;

import java.util.regex.MatchResult;
```

```java

public class Critique< T > {
   private String name = "";
   private String description = "";

   private Description matchedDescription = null; //?
   private MatchResult trigger = null; //?

   private T source = null; // e.g., Diagnostic
   private String sourceFile = ""; // If relevant

   private String matchText = "";
   private String text = "";
   private String altText = "";


   public Critique( Description matchedDescription, MatchResult trigger, T source,
                    String sourceFile, String text, String altText ) {
      name = matchedDescription.getName( );
      description = matchedDescription.getDescription( );
      this.matchedDescription = matchedDescription;
      this.trigger = trigger;
      this.source = source;
      this.sourceFile = sourceFile;
      this.text = text;
      this.altText = altText;
   }

   public String toString( ) {
      return toString( 0 );
   }

   public String toString( int textChoice ) {
      String result = "ANTIPATTERN: " + matchedDescription.getName( ) + "\n";
      if ( textChoice == 0 ) {
         result += text + "\n";
      } else {
         result += altText + "\n";
      }
```

```
        return result;
    }

    // GETTER METHODS
    public String getName( ) {
        return name;
    }

    public String getDescription( ) {
        return description;
    }

    public Description getMatchedDescription( ) {
        return matchedDescription;
    }

    public MatchResult getTrigger( ) {
        return trigger;
    }

    public T getSource( ) {
        return source;
    }

    public String getSourceFile( ) {
        return sourceFile;
    }

    public String getText( ) {
        return text;
    }

    public String getAltText( ) {
        return altText;
    }
}
```

Some tools are required for generating the critiques. *CritiquerTools.java* (Listing 6.7) contains a default structure critique that is used to alert the student when a `Diagnostic` does not match any antipattern descriptions. The method `getMatchResults( )` matches a regular expression to supplied text producing a list of match results. Finally, there is a factory method, `makeCritique` specifically for structure antipattern descriptions. This method takes a `Diagnostic`, a filename, and an array of instructor provided antipattern descriptions. From these inputs the factory method generates a `Critique`.

Listing 6.7: CritiquerTools.java - Critique Class

```
package edu.mtu.cs.webta.critiquer;

import javax.tools.Diagnostic;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.MatchResult;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class CritiquerTools {
   public static final Description ↩
      ↪ DEFAULT_STRUCTURE_CRITIQUE = new ↩
      ↪ Description(
         "BOGUS_ERROR", CodingAspect.STRUCTURE, ↩
            ↪ AntipatternSource.DIAGNOSTIC, false,
         "Triggered when code an unrecognized error↩
            ↪  occurs.",
         "^[\\S\\s$]*$",
         "ERROR:\n{0}\n",
         "ERROR:\n{0}\n"
   );

   private static List< MatchResult > ↩
      ↪ getMatchResults(
         String regexString,
         String text ) throws ↩
            ↪ PatternSyntaxException {
      List< MatchResult > matches = new ArrayList↩
         ↪ <>( );
      if ( regexString != null && !regexString.↩
         ↪ isEmpty( ) ) {
         Pattern regexPattern = Pattern.compile( ↩
            ↪ regexString, Pattern.MULTILINE );
```

```java
         Matcher matcher = regexPattern.matcher( text );
         while ( matcher.find( ) ) {
            matches.add( matcher.toMatchResult( ) );
         }
      }
      return matches;
   }

   public static < S extends Diagnostic > Critique< S > makeCritique(
        S source,
        String sourceFile,
        Description[] descriptions ) {
      Critique< S > critique = null;
      boolean critiqueFound = false;
      for ( Description description : descriptions ) {
         if ( description.getSource( ) != AntipatternSource.DIAGNOSTIC ) { continue; }
         List< MatchResult > matchResults = getMatchResults(
              description.getRegexString( ),
              source.toString( ) );
         if ( !matchResults.isEmpty( ) ) {
            critiqueFound = true;
            MatchResult matchResult = matchResults.get( 0 );
            String[] matchedGroups = new String[ matchResult.groupCount( ) + 1 ];
            for ( int i = 0; i < matchedGroups.length; i++ ) {
               matchedGroups[ i ] = matchResult.group( i );
            }
            String matchedText = matchedGroups[0];
            String critiqueText =
               MessageFormat.format(
                  description.getTextTemplate( ),
                  matchedGroups );
            String altCritiqueText =
               MessageFormat.format(
                  description.getAltTextTemplate( ),
                  matchedGroups );
            critique = new Critique<>(
                 description,
                 matchResult,
                 source,
                 sourceFile,
                 critiqueText,
                 altCritiqueText
            );
         }
      }
      if ( !critiqueFound ) {
         Description defaultDescription = DEFAULT_STRUCTURE_CRITIQUE;
         List< MatchResult > matchResults = getMatchResults(
              defaultDescription.getRegexString( ),
              source.toString( ) );
         for ( MatchResult critiqueMatch : matchResults ) {
            critique = new Critique< S >(
                 defaultDescription,
                 critiqueMatch,
                 source,
                 sourceFile,
                 defaultDescription.getTextTemplate( ),
                 defaultDescription.getAltTextTemplate( )
            );
         }
      }
      return critique;
   }
}
```

All that remains is to build a Structure Critiquer Application (Listing 6.8. The application needs access to an array of antipattern descriptions. For simplicity, we added this as a constant, but one can imagine the data being extracted from a database. There is a single instance method, `generateStructureCritiques( )`. This method takes as arguments a filepath, a filename and the array of antipattern descriptions. From these it produces a list of critiques for the student.

Listing 6.8: StructureCritiquer.java - A Rudimentary Structure Critiquer

```java
package edu.mtu.cs.webta.critiquer;

import edu.mtu.cs.webta.util.compile.CompilerTools;

import javax.tools.Diagnostic;
import javax.tools.JavaFileObject;
import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class StructureCritiquer {
   public static final Description[] ANTIPATTERN_CRITIQUE_DESCRIPTIONS = {
      new Description(
            "MISSING_SEPARATOR", CodingAspect.STRUCTURE, AntipatternSource.DIAGNOSTIC,
            false,
            "Triggered when code is missing a something, such as a semicolon or a parenthesis.",
            "^([[\\w-\\s]*/?]*)\\b([^/]+\\.\\S+)?\\:[\\s]*([\\d][\\d]*)[\\s]*:[\\s]*(error)\\:[\\s]*(\\'(.)\\'[\\s]*expected)$?([\\s][\\S]*[^$]*)$?([\\s]*[\\S])$",
          "ERROR:\n{0}\nThe compiler is looking↩
             ↪  for a missing ''{6}'' ↩
             ↪ somewhere near line {3} in file↩
             ↪  {2}.\nIf you don''t see the ↩
             ↪ problem at that location,\↩
             ↪ ncarefully read backwards ↩
             ↪ through the code\nlooking for ↩
             ↪ the missing character or some ↩
             ↪ other problem.\n",
          "ERROR: The compiler is looking for a↩
             ↪  missing ''{6}'' somewhere ↩
             ↪ around line {3}.\n"
     )
};

public List< Critique< Diagnostic< ? extends ↩
   ↪ JavaFileObject > > >
generateStructureCritiques(
     String folder,
     String javaClassname,
     Description[] descriptions
                          ) {
   File javaSourceFile = new File( folder + "/"↩
      ↪  + javaClassname + ".java" );
   List< Critique< Diagnostic< ? extends ↩
      ↪ JavaFileObject > > > critiques =
        Collections.synchronizedList( new ↩
           ↪ ArrayList<>( ) );

   ArrayList< String > options = new ArrayList<↩
      ↪  String> ( Arrays.asList(
        "-Xdoclint:all",
        "-deprecation",
        "-cp", folder ) );

   CompilerTools.generateDiagnostics(
        folder,
        javaClassname,
        options).forEach(
        diagnostic -> {
           critiques.add(
                CritiquerTools.< Diagnostic< ? ↩
                   ↪ extends JavaFileObject > ↩
                   ↪ >
```

```
                                       makeCritique(
                        diagnostic,
                        diagnostic.getSource( ).getName( ),
                        descriptions
              ) );
           } );
      return critiques;
   }

   public static void main( String[] args ) {
      String folder = args[ 0 ]; // E.g., "data/library"
      String javaClassname = args[ 1 ]; // E.g., "MissingSeparators"
      StructureCritiquer critiquer = new StructureCritiquer( );
      System.out.println( "Diagnostic Critiques for " + javaClassname + ".java" );
      critiquer
           .generateStructureCritiques( folder, javaClassname, ANTIPATTERN_CRITIQUE_DESCRIPTIONS )
           .forEach( System.out::println );
   }
}
```

Running `StructureCritiquer` and supplying arguments for the MissingSeparators.java code, we get the following code critiques (Figure 6.2):

Making structure-based critiquers is straight-forward. For the most part, time is spent eliciting antipattern descriptions from the instructor.

Figure 6.2: Example Critiques for MissingSeparators.java.

```
Diagnostic Critiques for MissingSeparators.java
ANTIPATTERN: MISSING_SEPARATOR
ERROR:
data/library/MissingSeparators.java:4: error: ']' expected
int [] foo = new int[10
                       ^
The compiler is looking for a missing ']' somewhere near line 4
in file MissingSeparators.java.
If you don't see the problem at that location,
carefully read backwards through the code
looking for the missing character or some other problem.


ANTIPATTERN: MISSING_SEPARATOR
ERROR:
data/library/MissingSeparators.java:5: error: '(' expected
switch nameOfCurvedSolid.toUpperCase()) {
      ^
The compiler is looking for a missing '(' somewhere near line 5
in file MissingSeparators.java.
If you don't see the problem at that location,
carefully read backwards through the code
looking for the missing character or some other problem.


ANTIPATTERN: MISSING_SEPARATOR
ERROR:
data/library/MissingSeparators.java:11: error: ';' expected
         2 * Math.PI * radius * height
                                      ^
The compiler is looking for a missing ';' somewhere near line 11
in file MissingSeparators.java.
If you don't see the problem at that location,
carefully read backwards through the code
looking for the missing character or some other problem.
```

Chapter 7

Critiquing Behavior

7.1 The Behavior of a Program

Behavior is related to the execution, logic, and semantics (meaning) of a program. Nothing is more frustrating to a novice programmer than writing code that runs, but produces incorrect results due to logic errors. In one study, Ettles, Luxton-Reilly and Denny, found the sources of novice logic errors to be "algorithmic errors, misinterpretations of the problem, and fundamental misconceptions" [24]. One of the most common fundamental misconceptions is indexing into strings, arrays, and lists. It is confusing to students that indexing begins at zero and the last element is positioned at the length - 1. Listing 7.1 illustrates a couple antipatterns that produce a string index out of bounds error (Antipatterns §12.3.13, §12.3.14).

Listing 7.1: StringIndexOutOfBounds.java - Common string antipatterns.

```
public class StringIndexOutOfBounds {
```

```java
  public void getLastCharacter( String str ) {
    char ch = str.charAt( str.length( ));
  }

  public void printSubstrings( String str ) {
    for( int index = str.length(); index >= 0 ; index-- ) {
      System.out.println( str.substring( index - 1, str.length( ) ) );
    }
  }
}
```

There are many different ways to identify behavioral antipatterns in code. For example, under the right conditions, a compiler can detect integer division by zero (Antipattern §12.2.12; e.g. when an integer variable is divided by an integer constant whose value is zero. Another way to identify behavior antipatterns is to rely on the instructor's knowledge of both the assignment and programming to test the student's code.

7.2 Testing with JUnit

An efficient way to find behavior antipatterns is to develop a battery of tests for common logic error that trip up students for any given assignment. This is called shakedown testing. Testing reveals logic errors and behavioral antipatterns. Most languages have an associated testing platform or framework that instructors can use to test student code. JUnit is the most popular testing framework for Java. Listing 7.2 is a Junit test suite containing two test methods. Test methods are any method immediately preceded by the `@Test` annotation. Methods not preceded by `@Test` are not executed by JUnit, but can be called as helper methods by the instructor's test code.

Listing 7.2: StringIndexOutOfBoundsTest.java - JUnit tests for antipatterns.

```java
import org.junit.Test;
import org.junit.Ignore;
import java.io.FileNotFoundException;

import static org.junit.Assert.*;

public class StringIndexOutOfBoundsTest {
  @Test( case_name="↩
     ↪ STRING_INDEX_OUT_OF_BOUNDS_EXCEPTION",
    point_value= 25.0,
    hint="Java strings are zero-indexed. That ↩
       ↪ means the valid index range is [0, ↩
       ↪ length), i.e. zero through the length of↩
       ↪  the string MINUS ONE.\n\n")
  public void upperStringIndexOutOfBoundsTest ( ) ↩
     ↪ {
    StringIndexOutOfBounds object = new ↩
       ↪ StringIndexOutOfBounds ();
    String str = "It's not black magic; it's just ↩
       ↪ Java code!";
    char result = object.getLastCharacter( str );
    char expected = '!';
    if ( result != expected ) {
      fail( String.format("Method ↩
         ↪ StringIndexOutOfBounds() returned an ↩
         ↪ expected value.\nINPUT: \"%s\"\nOUTPUT↩
         ↪ : '%s'\nEXPECTED: '%s'\n",
          str, result, expected));
    }
  }

  @Test( case_name="↩
     ↪ SUBSTRING_INDEX_OUT_OF_BOUNDS_EXCEPTION",
    point_value= 25.0,
    hint="The substring( start, end ) returns the ↩
       ↪ characters from the start index up to, ↩
       ↪ but not including, the end index. The ↩
       ↪ first argument to substring, the ↩
       ↪ starting index, must be in the range [0,↩
       ↪ length). The second argument, the ending↩
       ↪  index, must be in the range [0, length↩
       ↪ ].\n\n")
  public void substringStringIndexOutOfBoundsTest
    ( ) {
    StringIndexOutOfBounds object = new
      StringIndexOutOfBounds ();
    String stooge = "Shemp";
    object.printSubstrings( stooge );
  }
}
```

The test suite method `upperStringIndexOutOfBoundsTest()` tests the student's `getLastCharacter()` method in two ways: 1. it may trigger a runtime exception and 2. if no runtime exception is triggered, it verifies that the value returned is the expected return value based on the inputs provided. In the latter case, the instructor calls the JUnit `fail()` method with a message indicating what happened.

7.3 Using JUnit in a Behavior Critiquer

To use JUnit in our behavior critiquer, we need to register a listener object that will record the results of the JUnit test methods. The JUnit API provides `RunListener`, a listener class we can subclass for this purpose. There are several methods in `RunListener`, but we only need to override `testRunFinished()` to record the final test results (Listing 7.3). To accomplish this, we add the `finalResult` data field and associated getter method.

Listing 7.3: JUnitRunListener.java - Listening for Failed Tests.

```
package edu.mtu.cs.webta.util.junit;

import org.junit.runner.Result;
import org.junit.runner.notification.RunListener;

```

```
public class JUnitRunListener extends RunListener ↩
  ↪ {
  private Result finalResult = null;

  public Result getFinalResult( ) {
    return finalResult;
  }

  @Override
  public void testRunFinished ( Result result ) ↩
    ↪ throws Exception {
    finalResult = result;
  }

}
```

Listing 7.4 contains some methods that we will use to work with JUnit. In particular, `runTests()` is passed the test class, e.g. `StringIndexOutOfBoundsTest`, registers the listener, executes its tests, and returns the results.

Listing 7.4: UnitTestTools.java - Listening for Failed Tests.

```
package edu.mtu.cs.webta.util.junit;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.net.URL;
import java.util.List;

public class UnitTestTools {

  public static List< URL > urlClassPath( ↩
    ↪ ClassLoader classLoader, List< URL > lst )↩
    ↪  {
    if ( classLoader.getParent( ) != null ) {
      urlClassPath( classLoader.getParent( ), lst ↩
        ↪ );
    }
    return lst;
  }

  public static String getTrimmedTrace( Failure failure ) {
    StringWriter stringWriter = new StringWriter( );
    PrintWriter writer = new PrintWriter( stringWriter );
    Throwable e = failure.getException( );
    StackTraceElement[] stackTraceElements = e.getStackTrace( );
    writer.println( e );
    for ( StackTraceElement stackTraceElement : stackTraceElements ) {
      if ( stackTraceElement.getClassName( )
                            .equals( failure.getDescription( )
                              .getTestClass( )
                              .getName( ) ) ) {
        break;
      }
      writer.println( stackTraceElement.toString( ) );
    }
    return stringWriter.toString( );
  }

  public static String getFilteredTrace( Failure failure ) {
    StringWriter stringWriter = new StringWriter( );
    PrintWriter writer = new PrintWriter( stringWriter );
    Throwable e = failure.getException( );
    StackTraceElement[] stackTraceElements = e.getStackTrace( );
    StackTraceElement lastStackTraceElement = stackTraceElements[0];
    for ( StackTraceElement stackTraceElement : stackTraceElements ) {
      if ( stackTraceElement.getClassName( )
                            .equals( failure.↩
                               ↪ getDescription( ↩
                               ↪ )
                              .getTestClass( )
                              .getName( ) ) ) {
        break;
      }
      lastStackTraceElement = stackTraceElement;
    }
    writer.println( e );
    for ( StackTraceElement stackTraceElement : ↩
       ↪ stackTraceElements ) {
      if ( stackTraceElement.getClassName( ).↩
         ↪ equals( lastStackTraceElement.↩
         ↪ getClassName() ) ) {
        writer.println( stackTraceElement.toString↩
           ↪ ( ) );
      }
    }
    return stringWriter.toString( );
  }

  public static Result runTests( Class testClass )↩
     ↪  {
    JUnitCore core = new JUnitCore( );
    JUnitRunListener jUnitRunListener =
      new JUnitRunListener( );
    core.addListener( jUnitRunListener );
    core.run( testClass );
    return jUnitRunListener.getFinalResult();
  }
 }
```

The behavior critiquer is developed in Listing 7.5. The constant `ANTIPATTERN_CRITIQUE_DESCRIPTIONS` contains a `Description` of the array index out of bounds antipattern and response text for the critique. Domain knowledge is used to provide advice when tests fail and is encoded in the Description. Given the source file path and JUnit test filename, the `generateCritique()` method sets-up a java class path that encompasses the JUnit libraries, the instructor's test suite, and the compiled student source files.

This information is passed to the compiler via the `generateDiagnostics()` method, described in an earlier chapter. If there are no diagnostics returned, then compilation was successful so we load the compiled JUnit test class and pass it to `UnitTestTools.runTests()`. This returns the JUnit results, which contains information about failed tests. We convert that information into critiques. Figure 7.1 shows the output critique generated when `substringStringIndexOutOfBoundsTest()` failed even though there isn't an antipattern description for it.

Listing 7.5: BehaviorCritiquer.java - A Behavior Critiquer.

```java
package edu.mtu.cs.webta.critiquer;//import static  edu.mtu.cs.webta.util.junit.UnitTestTools.urlClassPath;

import edu.mtu.cs.webta.util.compile.CompilerTools;
import edu.mtu.cs.webta.util.compile.MyCompilerTools;
import edu.mtu.cs.webta.util.junit.JUnitRunListener;
import edu.mtu.cs.webta.util.junit.UnitTestTools;
import org.junit.runner.notification.Failure;

import javax.tools.Diagnostic;
import javax.tools.JavaFileObject;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.StringJoiner;
import java.util.stream.Collectors;

import static edu.mtu.cs.webta.util.junit.UnitTestTools.urlClassPath;

```

```java
public class BehaviorCritiquer {
    public static final Description[] ↩
        ↪ ANTIPATTERN_CRITIQUE_DESCRIPTIONS = {
            new Description( "↩
                ↪ STRING_INDEX_OUT_OF_BOUNDS", ↩
                ↪ CodingAspect.BEHAVIOR,
                AntipatternSource.EXCEPTION, false,
                "Trigger at runtime by calling a String↩
                    ↪  method with an index that is out↩
                    ↪  of the range [0, length).",
                "\\A(java.lang.(↩
                    ↪ StringIndexOutOfBoundsException))↩
                    ↪ :[\\s]+(String\\sindex\\sout\\sof↩
                    ↪ \\srange:\\s([-]*[0-9]+))[\\s]*$↩
                    ↪ ?[\\s\\S$]*\\.(String)\\.([A-Za-↩
                    ↪ z_][A-Za-z_\\$]*)\\(([\\S]*\\)↩
                    ↪ (^?((((([^\\.\\(\\)\\:$]*)\\.)+)↩
                    ↪ ([^\\(\\)$]*))\\(([^:\\(\\)$]*)↩
                    ↪ :([^\\)$]*)\\))$",
                "ERROR:\n{0}\n\nA call was made to the ↩
                    ↪ String method {6}() with an index↩
                    ↪  of {4} at line {14} in {13}.\↩
                    ↪ nThe index must be in the range ↩
                    ↪ [0, length).\nNote that the last ↩
                    ↪ element in the string is at ↩
                    ↪ position length - 1.\nThis ↩
                    ↪ problem often occurs in for-loops↩
                    ↪  and is resolved by using < ↩
                    ↪ instead of <= in the end ↩
                    ↪ condition of the loop.\n",
                "ERROR: The index in a call to {6}({4})↩
                    ↪  in {11}.{12}() is out of bounds↩
                    ↪ .\n"
            )
    };

    public List< Critique< Failure > > ↩
        ↪ generateCritiques( String sourceFolder, ↩
        ↪ String junitFilename )
            throws ClassNotFoundException, ↩
                ↪ MalformedURLException {
        String libFolder = System.getProperty( "user↩
            ↪ .dir" ) + "/lib";
        StringJoiner classpathJoiner = new ↩
            ↪ StringJoiner( ":" );
        classpathJoiner.add( sourceFolder );
        classpathJoiner.add( libFolder );
        classpathJoiner.add( libFolder + "/lib/org/junit/Test.class" );
        classpathJoiner.add( libFolder + "/hamcrest-core-1.3.jar" );
        classpathJoiner.add( libFolder + "/junit-4.12.jar" );

        ArrayList< String > options = new ArrayList<>( ( List< String > ) Arrays.asList(
            // Specify where to find user class files and annotation processors
            "-classpath", classpathJoiner.toString( ) ) );

        List< Diagnostic< ? extends JavaFileObject > > diagnosticsList =
            CompilerTools.generateDiagnostics(
                sourceFolder, junitFilename, options );

        if ( diagnosticsList.isEmpty( ) ) {

            URL url1 = new File( sourceFolder ).toURL( );
            URL url2 = new File( libFolder + "/junit-4.12.jar" ).toURL( );
            URL url3 = new File( libFolder + "/lib/org/junit/Test.class" ).toURL( );
            List< URL > urls = Arrays.asList( url3, url2, url1 );
            urlClassPath( BehaviorCritiquer.class.getClassLoader( ), urls );

            URL[] urlArray = new URL[ urls.size( ) ];
            for ( int i = 0; i < urls.size( ); i++ ) {
                urlArray[ i ] = urls.get( i );
            }

            ClassLoader cl = new URLClassLoader( urlArray );
            Class annotationClass = cl.loadClass( "org.junit.Test" );
```

```
         Class junitClass = cl.loadClass( junitFilename );


         System.out.println( "Behavior Critiques" );
         UnitTestTools.runTests( junitClass )
            .getFailures( )
            .stream( )
            .map( failure ->
               CritiquerTools.< Failure > makeCritique(
                  failure,
                  failure.getDescription( )
                  .getClassName( ),
                  ANTIPATTERN_CRITIQUE_DESCRIPTIONS ) )
            .collect( Collectors.toList( ) );
      } else {
         System.err.println( "COMPILER ERRORS & WARNINGS" );
         for ( Diagnostic< ? extends JavaFileObject > diagnostic : diagnosticsList ) {
            System.err.println( diagnostic + "\n" );
         }
      }
      return new ArrayList<>( );
   }

   public static void main( String[] args ) throws ClassNotFoundException, MalformedURLException {
      String sourceFolder = args[ 0 ];
      String junitTestClassname = args[ 1 ];
      BehaviorCritiquer behaviorCritiquer = new BehaviorCritiquer( );
      behaviorCritiquer.generateCritiques( sourceFolder, junitTestClassname )
                       .forEach( System.out::println );
   }
}
```

Figure 7.1: Example Critiques for StringIndexOutOfBounds.getLastCharacter( ).

```
ANTIPATTERN: STRING_INDEX_OUT_OF_BOUNDS
ERROR:
java.lang.StringIndexOutOfBoundsException:
   String index out of range: 42
java.base/java.lang.StringLatin1.charAt(StringLatin1.java:47)
java.base/java.lang.String.charAt(String.java:693)
StringIndexOutOfBounds.getLastCharacter(
   StringIndexOutOfBounds.java:9)

A call was made to the String method charAt() with an index of 42
at line 9 in StringIndexOutOfBounds.java.
The index must be in the range [0, length).
This most often occurs in for-loops and is resolved by
using < instead of <= in the end condition of the loop.
```

Listing 7.6: Test.java - Overriding the test annotation.

```java
package org.junit;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD})
public @interface Test {
  static class None extends Throwable {
    private static final long serialVersionUID = 1↩
       ↪ L;

    private None() {
    }
  }

  Class<? extends Throwable> expected( ) default ↩
     ↪ org.junit.Test.None.class;

  // Changed default from 0L. We want a timeout ↩
     ↪ for student code. 5min should be enough.
  long timeout( ) default 300000L;

  String case_name( ) default "";
  String user_story( ) default "";
  String test_case( ) default "";
  String hint( ) default "";
  double point_value( ) default 1.0d;
}
```

Astute JUnit programmers will have noticed that the `Test` annotation has some unusual arguments. We replaced the annotation with our own (Listing 7.6). This enables the instructor to provide a hint, based on their deep domain knowledge, when an unanticipated error occurs. Figure 7.2 shows a critique generated in this way.

Figure 7.2: Example Instructor Critique for Logic Error.

```
ANTIPATTERN: BEHAVIOR_ERROR
java.lang.StringIndexOutOfBoundsException:
   begin 0, end -1, length 5
java.base/java.lang.String.checkBoundsBeginEnd(String.java:3319)
java.base/java.lang.String.substring(String.java:1874)
StringIndexOutOfBounds.printSubstrings(
   StringIndexOutOfBounds.java:14)

The substring( start, end ) returns the characters from the start
index up to, but not including, the end index. The first argument
to substring, the starting index, must be in the
range [0,length). The second argument, the ending index, must be
in the range [0, length].
```

Chapter 8

Critiquing Style

8.1 Style

Our culture views style as a highly personal characteristic to be developed by individuals. We see this taken to extremes in fashion and the arts. Students are encouraged through high school to develop their own unique writing style and authors are studied and analyzed based on theirs. Yet in computer science we find that experts conform to community standards for style and the adoption of good programming guidelines is a critical aspect of gaining mastery of a programming language. Good programming style helps us communicate our solutions efficiently and makes our code readable and maintainable by others [42, 61]. Developing good coding style prevents bugs.

8.2 Static Analysis

Our approach in developing a simple style critiquer is to perform static analysis of student code. Scanning the code reveals style mistakes and can be done even if the code fails to compile. As before, we will rely on regular expressions to match style violations and trigger guidance.

Listing 8.1: ImportsOwnPackage.java - Example Style Antipattern.

```java
package edu.mtu.cs;

import edu.mtu.cs.*;

public class ImportsOwnPackage {

}
```

Listing 8.1 illustrates an example of poor style where a student places their code in a package, then imports code from that package (Antipattern §12.2.18). This is unnecessary in Java and makes the student's code look amateurish. A common novice mistake, importing one's own package may be due to a misconception that any code, outside the core `java.lang` package, must be imported to be available for use.

Listing 8.2: StyleCritiquer.java - Static Code Analysis.

```java
package edu.mtu.cs.webta.critiquer;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.MatchResult;

public class StyleCritiquer {
   public static final Description[] ↩
      ↪ ANTIPATTERN_CRITIQUE_DESCRIPTIONS = {
         new Description( "IMPORT_OWN_PACKAGE",
                          CodingAspect.STRUCTURE,
                          AntipatternSource.CODE,
                          false,
                          "Triggered when code ↩
                             ↪ imports the package↩
                             ↪  it resides within.↩
                             ↪ ",
                          "(?s)package\\s+([\\w↩
                             ↪ \\.]*);.*import\\s↩
                             ↪ +\\1\\.[^\\.;]*\\s↩
                             ↪ *;",
                          "You automatically have ↩
                             ↪ access to the ↩
                             ↪ classes in your own↩
                             ↪  package -- no need↩
                             ↪  to import them.",
                          "Do not import the ↩
                             ↪ package the code is↩
                             ↪  defined within.\n"
         )
   };

   public ArrayList<Critique<String>>  ↩
      ↪ generateCritiques( String sourcepath, ↩
      ↪ String sourcefile ) throws IOException {
      String filename = sourcepath + "/" + ↩
         ↪ sourcefile;
      Path filepath = Path.of( filename );
      String sourceText = Files.readString( ↩
         ↪ filepath );
      ArrayList<Critique<String>> critiqueList = ↩
         ↪ new ArrayList<>(  );
      for ( Description description : ↩
         ↪ ANTIPATTERN_CRITIQUE_DESCRIPTIONS ) {
         if ( description.getSource( ) == ↩
            ↪ AntipatternSource.CODE ) {
            List< MatchResult > matchResults =
                 CritiquerTools.getMatchResults(
                      description.getRegexString( ),
                      sourceText );
            if ( !matchResults.isEmpty() ) {
               for( MatchResult matchResult : matchResults ) {
                  String[ ] matchedGroups = new String[ matchResult.groupCount() + 1 ];
                  for ( int i = 0; i < matchedGroups.length; i++ ) {
                     matchedGroups[ i ] = matchResult.group( i );
                  }            String matchedText = matchedGroups[0];
                  String critiqueText = MessageFormat.format( description.getTextTemplate( ), matchedGroups );
                  String altCritiqueText = MessageFormat.format( description.getAltTextTemplate( ), matchedGroups );
                  CodeCritique<String> critique = new CodeCritique<>(
                       description,
                       matchResult,
                       sourceText,
                       filename,
                       matchedText,
                       critiqueText,
                       altCritiqueText
                  );
                  critiqueList.add( critique );
               }
            }
         }
      }
      return critiqueList;
   }

```

```
    public static void main( String[] args ) throws
        IOException {
        String pathname = args[ 0 ];
        String filename = args[ 1 ];
        StyleCritiquer styleCritiquer = new
            StyleCritiquer( );
        styleCritiquer.generateCritiques( pathname,
            filename )
                .forEach( System.out::println );
    }
}
```

Our style critiquer, Listing 8.2, begins by declaring a constant `ANTIPATTERN_CRITIQUE_DESCRIPTION` containing a list of antipattern descriptions. Again, this is for convenience, in a production application this list would likely be loaded from a database and may vary between assignments as the instructor draws upon pedagogical knowledge to tailor the critiques to their courses.

The heart of the style critique is the `generateCritiques()` method. Here we see the familiar regular expression code. What is different is that we are matching against the entire text of the source code, a process producing potentially many matches throughout the code. Because students may repeat antipatterns several times through the code, we ask instructors to provide two text templates in their critique descriptions. A lengthy tutorial critique message for the first time the code triggers a particular critique and a shorter alternative critique for every match thereafter.

Applying the style critiquer to our example code results in the following critique (Figure 8.1).

Figure 8.1: Example Style Critique.

```
ANTIPATTERN: IMPORT_OWN_PACKAGE
Found at Line:Column (1:1) - (3:21) in
data/library/ImportsOwnPackage.java

package edu.mtu.cs;

import edu.mtu.cs.*;

You automatically have access to the classes in your own package -
no need to import them.
```

Another style issue with the code in Listing 8.1, is that it imports everything in the package (Antipattern §12.4.10). Wildcard imports are generally not used by experts who are very deliberate and specific about the code they import. When developing a `Description` for a style antipattern, it is often useful to include a link to a relevant style guideline. E.g. a link to the Google Java Style Guide, `https://google.github.io/styleguide/javaguide.html#s3.3.1-wildcard-imports`.

Chapter 9

Critiquing Design

9.1 Illuminating Patterns and Antipatterns

Design is the process of deciding how to model the world in code. Learning to design programs calls for reflection and practice [25]. Critiquers provide feedback based on patterns and antipatterns present in student code. The presence of positive patterns indicates the student has well-designed code that intentionally models the world toward creating robust solutions. The presence of antipatterns in the code indicates poor design choices containing code that negatively impacts robustness and accuracy. The byproducts of virtuous or poor design, patterns and antipatterns provide a window into the student's design process and an opportunity for instructor's to support the student in their journey from novice to master coder.

We have developed code critiquers that combine an instructor's knowledge of teaching and programming with software that compiles, executes, tests, and searches code for antipatterns, which are then used to provide design feedback for the students to reflect upon. In this chapter, we discuss some additional critique techniques for prompting students to reflect on their design process.

9.2 Identifying Patterns with an Abstract Syntax Tree

One very powerful tool at our for identifying patters in code is an Abstract Syntax Tree (AST). An AST is a tree representation of the structure of code. Trees can help us analyze properties of the code, such as coupling and cohesion, that it is difficult for novices to understand otherwise.

Traversing an AST, we can identify patterns that might be difficult to detect by testing or static analysis. For example, the code in Listing 4.9 is fully listed as a Java class in Listing 9.1. This code can be difficult to detect using regular expressions. Most regular expression engines come with two modes for matching text: lazy and greedy. Balancing curly brackets can be problematic in either mode allowing a bracket match too soon or too late respectively.

Listing 9.1: EmptyForLoop.java - Example Design Antipattern.

```java
/**
 * Class exhibits the EMPTY-KNEE_JERK-CODE ↩
   ↪ Antipattern
 */
public class EmptyForLoop {
   /**
    * @param num is an integer value
    * @return two times the specified number
    */
   public int mult2x ( int num ) {
        // Loop does not affect solution, but serves
            no purpose.
        for(int i=0; i<10; i++){

        }
        return 2 * num;
    }
}
```

In order to parse this code and construct an AST, we execute the following code (Listing 9.2).

Listing 9.2: Code Snippet to Construct an AST.

```
Lex lex = new Lex("data/library/EmptyForLoop.java"
    );
ArrayList<Token> tokenList = lex.lex("data/library
    /formatted_output");
BuildAST ast = new BuildAST(tokenList);
Root root = ast.build();
```

We can print out the tokens that were parsed to construct the AST. The code in Listing 9.3 produces the output in Figure 9.1.

Listing 9.3: Printing the tokens parsed by the lexer.

```
for(Token t : tokenList) {
  System.out.printf("[TOKEN:%s:%s, %s, %s]\n",
                    t.getLine(),
                    t.getIndex(),
                    t.getId(),
                    t.getWord());
}
```

We can also print the nodes in the AST (Listing 9.4 produces output in Figure 9.2).

Figure 9.1: Tokenization of Listing 9.1.

```
[TOKEN:4:0, modifier, public]
[TOKEN:4:1, declare, class]
[TOKEN:4:2, class, EmptyForLoop]
[TOKEN:4:3, Lbrace, {]
[TOKEN:9:4, modifier, public]
[TOKEN:9:5, type, double]
[TOKEN:9:6, method, abs]
[TOKEN:9:7, Lparen, (]
[TOKEN:9:8, type, double]
[TOKEN:9:9, pvar, d]
[TOKEN:9:10, Rparen, )]
[TOKEN:9:11, Lbrace, {]
[TOKEN:10:12, type, double]
[TOKEN:10:13, var, result]
[TOKEN:10:14, assign_op, =]
[TOKEN:10:15, pvar, d]
[TOKEN:10:16, semi_colon, ;]
[TOKEN:11:17, conditional, if]
[TOKEN:11:18, Lparen, (]
[TOKEN:11:19, pvar, d]
[TOKEN:11:20, compare_op, <]
[TOKEN:11:21, const, 0]
[TOKEN:11:22, Rparen, )]
[TOKEN:11:23, Lbrace, {]
[TOKEN:12:24, loop, for]
[TOKEN:12:25, Lparen, (]
[TOKEN:12:26, type, int]
[TOKEN:12:27, var, i]
[TOKEN:12:28, assign_op, =]
[TOKEN:12:29, const, 0]
[TOKEN:12:30, semi_colon, ;]
[TOKEN:12:31, var, i]
[TOKEN:12:32, compare_op, <]
[TOKEN:12:33, const, 10]
[TOKEN:12:34, semi_colon, ;]
[TOKEN:12:35, var, i]
[TOKEN:12:36, post-unary_op, ++]
[TOKEN:12:37, Rparen, )]
[TOKEN:12:38, Lbrace, {]
[TOKEN:13:39, Rbrace, }]
[TOKEN:14:40, var, result]
[TOKEN:14:41, assign_op, =]
[TOKEN:14:42, unary_op, -]
[TOKEN:14:43, pvar, d]
[TOKEN:14:44, semi_colon, ;]
[TOKEN:15:45, Rbrace, }]
[TOKEN:16:46, branch, return]
[TOKEN:16:47, var, result]
[TOKEN:16:48, semi_colon, ;]
[TOKEN:17:49, Rbrace, }]
[TOKEN:18:50, Rbrace, }]
```

Listing 9.4: Printing the nodes int the AST.

```java
for( Node node : root.ALL_NODES ) {
  System.out.printf( "[%s:%s]\n",
                 node.getClass().getSimpleName(),
                 node.getToken().getWord() );
}
```

More interesting is that we can traverse the tree or search through the nodes to identify patterns. Listing 9.5 searches through AST and prints all methods

Figure 9.2: Nodes in the AST for Listing 9.1.

```
[Root:]
[Other:public]
[Other:class]
[ClassDecl:EmptyForLoop]
[ClassBody:{]
[MethodDecl:abs]
[Parameter:d]
[Body:{]
[VarDecl:result]
[VarDecl:result]
[Variable:d]
[IfStmt:if]
[Variable:d]
[Constant:0]
[BinaryOp:<]
[Body:{]
[ForStmt:for]
[VarDecl:int]
[VarDecl:int]
[Constant:0]
[Variable:i]
[Constant:10]
[BinaryOp:<]
[UnaryOp:i]
[Variable:i]
[Assign:result]
[Variable:result]
[Variable:d]
[UnaryOp:-]
[ExprStmt:result]
[BranchStmt:return]
[Variable:result]
```

containing empty loops. See Figure 9.3 for the output.

Listing 9.5: Idenifying methods exhibiting the EMPTY-KNEE-JERK Antipattern.

```java
root.ALL_NODES.stream()
    .filter( node -> node.getToken().getId().contains("loop") && (( ForStmt ) node).getBody() == null )
    .forEach( node -> {
    Node n = node;
    while( n != null && !n.getToken().getId().contains( "method" ) ) {
      n = n.getParent();
    }
    System.out.println("Method containing Knee-Jerk Code\n");
    System.out.println(n);
    });
```

Figure 9.3: Nodes in the AST for Listing 9.1.

```
The following method contains Knee-Jerk Code

public  double  abs( double  d ) {
 double result = d;
if (d < 0) {
for(int = 0;i < 10;[i++]){
}
result = d-;
}
return result;
}
```

9.3 Identifying Antipatterns in Bad Code

Novice programmers sometimes write code that will not compile. Instructors can look at bad code and still offer advice to students. Our code critiquers are be able to do the same. An AST can be a powerful tool for identifying antipatterns in the presence of code that normally will not parse and compile because it violates the rules of the language. Listing 9.6 contains good code that the student has placed at the top-level in the class body. This is a fairly common mistake as students develop a mental model of how to organize their code.

Listing 9.6: CodeOutsideMethod.java - CODE-OUTSIDE-METHOD Antipattern.

```java
public class CodeOutsideMethod {
   public long fubar = 0;

   for(int i=0;i<10;i++){
      System.out.println("Hello World");
   }

   public void methodName ( ) {
        int a = 7;
    }

    int b = 0;

    System.exit(-1);
}
```

Our code critiquer navigates to the class body and accesses the bad code directly, but we could have traversed the tree looking for "garbage" nodes. When the parser is creating the AST, instead of stopping when an error is discovered, it creates a "garbage" node and continues parsing. In this way we identify valid code that is in the wrong place and can advise the student where to place it. Parsers and ASTs do not normally operate in this manner. This is a significant contribution to the development of code critiquers by this research. (See Listing 9.7 and Figure 9.4.)

Listing 9.7: Idenifying Antipatterns in Bad Code.

```
System.out.println( root.getClassDecl().↩
    ↪ getClassBody().getGarbage() );
```

Figure 9.4: Bad code found in the AST for 9.6.

```
!!Garbage-0:
for(int = 0;i < 10;[i++]){
System.out.println("Hello World");
}
!!
!!Garbage-1:
System.exit([1-]);
!!
```

9.4 Using the AST to Prevent False Positives

Sometimes our regular expression driven Style Critiquer will produce false positives. For example, it could identify a pattern not in the code, but present in comments or strings. Listing 9.8 contains an example. Applying the Style Critiquer to this code identifies Antipattern §12.4.7. See output in Figure 9.5.

Listing 9.8: FalsePositive.java - Causes a Style Critiquer False Positive.

```
1  public class FalsePositive {
2     /**
3      * @param a - a number
4      * @param b - another number
5      * @return (a+b)*(a+b)
6      */
7     public double addAndSquare ( double a, double b ) {
8        return Math.pow( a + b , 2 )
9     }
10 }
```

Figure 9.5: Style Critique for Listing 9.8.

```
ANTIPATTERN: CRAMMED_OPERATORS
Found at Line:Column (5:16) - (5:25)
in data/library/FalsePositive.java

    * @return (a+b)*(a+b)

Don't cram operators. Put a space on both sides.
```

The AST can identify comments and strings in the code (Figure 9.6). A critiquer can use this information to check the antipatterns it identifies. In this case, the antipattern was identified on Line 5 in the program and the AST indicates that matched text on Line 5 in contained in a comment. Thus the critiquer should not report the critique to the student.

Figure 9.6: AST identified comments in 9.8.

```
There are 5 comments declarations in this code
[comment: Line 2, // ------------------------]
[comment: Line 3, //@param a - a number]
[comment: Line 4, //@param b - another number]
[comment: Line 5, //@return (a+b)*(a+b)]
[comment: Line 6, // ------------------------]
```

Part III

A Critiquer for Introductory Computer Science

Chapter 10

WebTA: A Tool for Automated Code Critique

10.1 WebTA

Traditional methods for teaching computer science — lecturing on abstract concepts, assigning a programming project related to the lectures, then grading the students' submitted finished products — resemble the outdated waterfall model of software development [51] in many ways:

- An instructor writes a specification and hands it off to students as an assignment.
- Students toil in isolation, without the benefit of instructor feedback or team communication.
- When they run out of time, students submit the assignment and hope for the best – not entirely sure that they interpreted the assignment in the

same way as the instructor.

- Lastly, the instructor applies secret tests to the student work and assigns a grade, then moves on to the next topic, regardless of whether students have successfully constructed mental models sufficient to understand the current topic.

With WebTA, we employ an authentic approach for today's software world, teaching students test-driven agile development methods through small cycles of teaching, coding integrated with testing, and immediate feedback. We focus on this Learning Cycle [36] by providing students just-in-time code critiques for them to reflect on and feedback into a continuous development process (Fig. 10.1). This exposure prepares them better for today's software industry and reduces the frustration that students often experience in early programming projects, mitigating the risk of student burnout and helping with retention in computing-related majors.

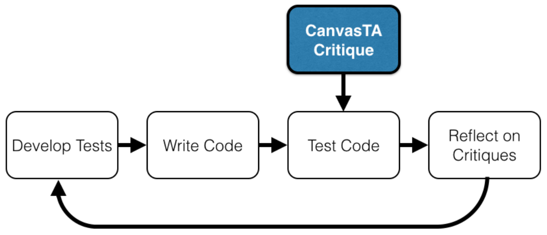


Figure 10.1: WebTA development cycle.

Students in our introductory course work on 7–10 programming assignments per semester; of these, 5–7 are large, multi-week projects. Within these strict time bounds, students must parse the project specification, extract from this a suitable design, and implement it in code. The pace can sometimes outdo students, and they often feel they submit their assignment without a deep understanding of the project, its requirements, or a sense of the grade they will receive. In this context, the WebTA tool provides students with immediate testing and feedback while they complete the assignment so there are no surprises at the end.

The following features are central to the use of WebTA in the classroom:

Supports Agile Development. Students using WebTA are learning to develop software using modern techniques. WebTA supports small cycles of specification, coding integrated with testing, and reflection, in the spirit of agile development.

Continuous Shakedown Testing. 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.

Progressive Code Scalability. WebTA grows with the student. It will evaluate expressions and code fragments submitted by beginning students, classes submitted by intermediate students, and multiclass projects submitted by advanced students.

Pseudocode Support. WebTA can translate back and forth between code and pseudocode. Students can submit pseudocode for evaluation. Java code can be translated to pseudocode, providing students with means to check their logic and learn to fluidly switch between code and English.

Immediate Feedback. 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 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.

Preliminary and Automatic Grade Assessment. After submitting code, students see a summary screen that uses a stop-light metaphor to communicate how well they are doing. A red light indicates they have not passed all the tests, yellow indicates they received some warnings or unresolved guidance, and green means they passed all tests and no further changes are necessary. Behind the scenes, a preliminary score is generated for approval by instructors or instructors can configure the system to automatically assign grades.

10.1.1 Architecture

WebTA is comprised of several modules, data sources, components, and interaction modes. Figure 10.2 shows an overview of the WebTA architecture. A detachable LTI Module provides services required for authentication with, and grade reporting back to, the Canvas Learning Management System (LMS). 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, Critique Database, and Grade Reports. Student Code and Instructor Tests are comprised of source files uploaded to WebTA. The Critique Database contains rules for identifying patterns and dispensing advice to the students.

Our introductory computer science sequence utilizes the Java programming language. Thus our first pass at creating a Critiquer System focuses on providing feedback to students on their Java code.

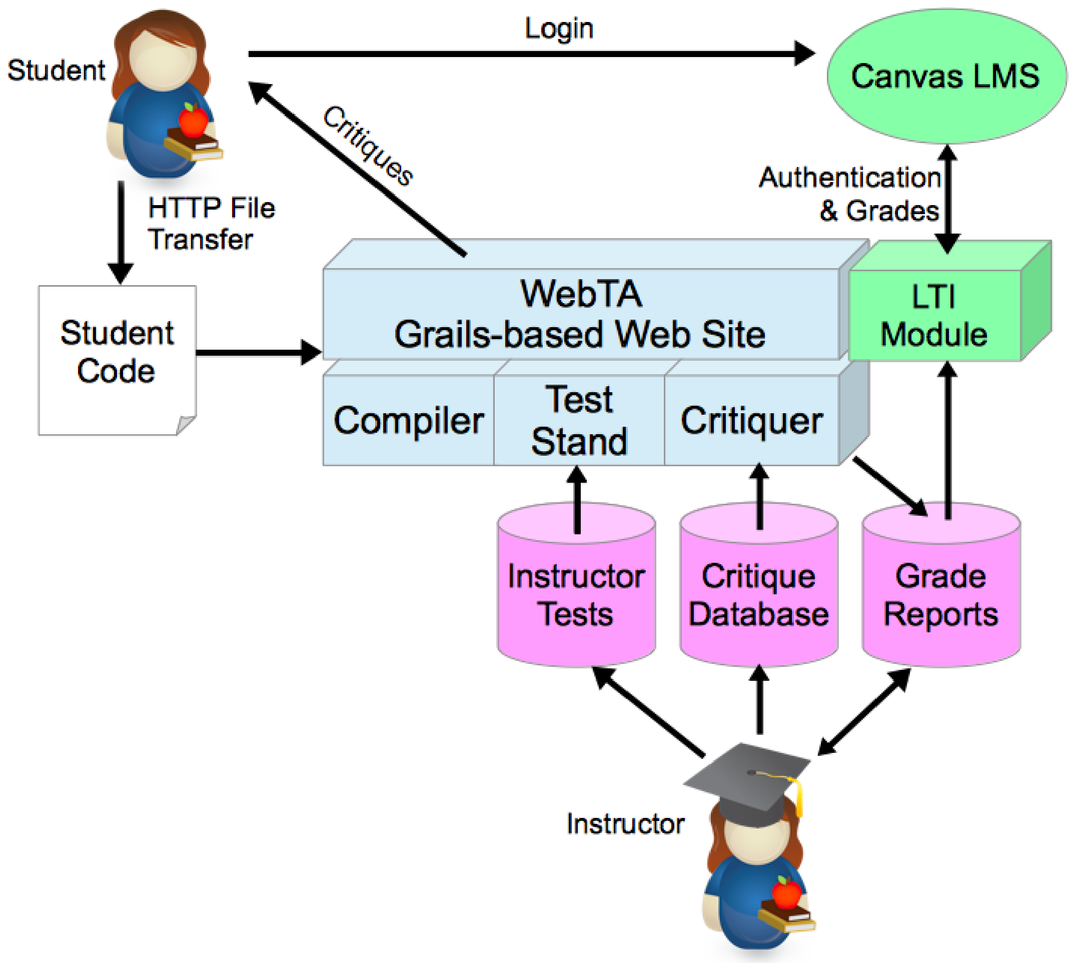


Figure 10.2: WebTA System Architecture

There are four modes of interacting with the system:

1. *Snippet*: Students enter code snippets in a text-area that may come seeded with partial code to get students started. The code snippet entered by the student can be as simple as an expression or as sophisticated as an entire class. The snippet is analyzed and tested according to the configuration set up by the instructor. Possible analysis include English translation, compilation, test execution, and style analysis.
2. *Project*: A drag-and-drop field is provided for uploading project files. Uploaded files are compiled, tested, and critiqued. The student receives feedback in their browser.

3. *Test Coverage*: Students supply their own test cases. These are run against an instructor supplied project and feedback is provided on test coverage.
4. *Instructor*: Canvas is also used to authenticate faculty and graduate TAs, enabling them to access WebTA in their browser. Instructors use WebTA to setup code tests, enter rules into the critique database, and view/modify grade reports before assigning final grades.

These modes of interaction are built-into the web application and are not dictated by Canvas. After authentication, all interactions are through web pages served-up by the WebTA server.

10.1.2 Configuration

Students using WebTA are engaged in communication-by-proxy with the instructor. The instructor configures WebTA with common critiques that are triggered by errors, warnings, or textual analysis of the student's code. These critiques are issued to the students immediately; as needed by the student to support concept formation. This communication is not meant to replace instructor feedback; rather, it codifies common feedback scenarios to assist the instructor in reaching students in tight feedback loops just when the student is engaged in problem solving and learning. A particularly effective configuration is one in which students work in pairs — with appropriate mentoring in pair programming skills [38, 40] — and utilize both live feedback from the instructor and automated critique from WebTA. In this way, students get exposure to consulting and processing feedback from a range of sources.

10.1.2.1 Antipatterns

Many problems can be caught by compiling, testing, and analyzing the code for known antipatterns. But sometimes WebTA must rely on configuration setup by an instructor knowledgeable in the problem domain.

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 JUnit 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, The instructor formulates the following assignment:

Develop a class named Fibonacci that contains the methods:
`fibRecursion` which given an integer n returns the long nth element of the Fibonacci Sequence. if n is out of range, return -1.

For this problem, the instructor sets up two tests, one of them being secret. The tests, using JUnit test conventions, are shown in Listing 10.1.

Listing 10.1: Instructor Tests

```
@Test
@CanvasTaTest( points = 5, name = "fibRecursive Test.", description = "Checks for all values of n from 0 to 9.", hint = "Remember to test for the base case." )
public void fibRecursiveTest( ) {
Fibonacci prog = new Fibonacci( );
long[ ] solutions = {
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
};
for( int n = 0; n < solutions.length; n++ ) {
long result = prog.fibRecursive( n ); if ( solutions[ n ] != result) {
      fail( "fibRecursive( " + n + " ) = " + solutions[ n ]
        + ". Your method returned " + result );
} }
}
// Secret Test
@Test
@CanvasTaTest( points = 5, name = "fibRecursive Edge Case", description = "Checks for n = -1" )
public void fibRecursiveEdgeCaseTest( ) {
Fibonacci prog = new Fibonacci( ); long result = prog.fibRecursive( -1 );
if ( result != -1 ) {
  fail( "fibRecursive( " + -1 + " ) = " + -1
      + ". Your method returned " + result );
}
}
```

The instructor also creates (or reuses existing) rules for the static analyzer. Rules consist of a regular expression, that matches with a problematic code snippet, coupled with advice the instructor would give a student who wrote such code. For example, anticipating a Knee-Jerk pattern, the instructor could setup the rule shown in Listing 10.2.

Listing 10.2: Instructor rule capturing an empty base case.

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

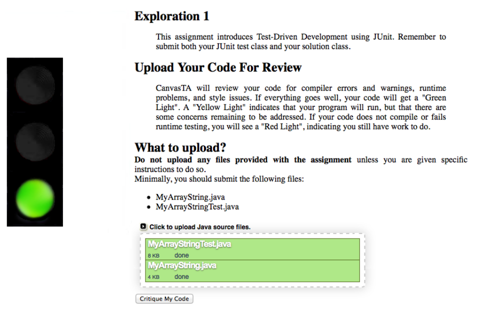
All of the built-in tests and feedback can be toggled by the instructor on a per assignment basis.

10.1.3 Operation

When students connect with WebTA, a startup screen that explains the current problem and tells them which files they should upload to receive a code critique (Fig. 10.3). After clicking on the “Critique My Code” button, students receive an online report which includes a Critique Summary (Fig. 10.4). A stoplight metaphor, commonly used within the agile development community, is used to indicate student progress through the assignment. The stoplight indicators and code critiques prompt students to reflect and refactor. The critique summary includes a stoplight that tells the student at a glance if they succeeded in their programming task. A green light indicates a satisfactory state and a red light indicates serious errors.

In addition to the “pass-fail” criteria of the green-red stoplight metaphor (which are useful, especially if done in a scaffolded way like Test My Code), WebTA allows instructors to include more heuristic conditions that can be triggered when students may be diverging from “good practice”, e.g. style or design issues that may not cause tests to fail. An amber light indicates the presence this type of problem. WebTA also allows for automated positive feedback. The assessment summary section also lists the parts of the critique and how the student performed in them.

Under the hood, the system has compiled their code and run it through a series of rigorous 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. The instructor can configure the system to run both public and secret tests, run the student’s own test code against their program, or assess the student’s JUnit test cases to determine their ability to generate



Exploration 1

This assignment introduces Test-Driven Development using JUnit. Remember to submit both your JUnit test class and your solution class.

Upload Your Code For Review

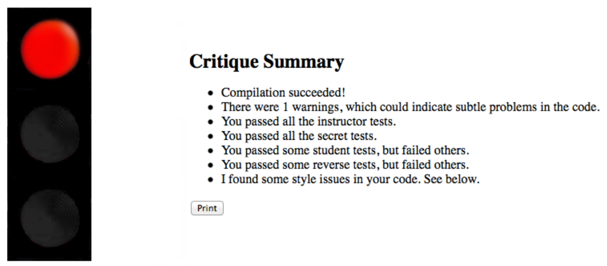
CanvasTA will review your code for compiler errors and warnings, runtime problems, and style issues. If everything goes well, your code will get a "Green Light". A "Yellow Light" indicates that your program will run, but that there are some concerns remaining to be addressed. If your code does not compile or fails runtime testing, you will see a "Red Light", indicating you still have work to do.

What to upload?

Do not upload any files provided with the assignment unless you are given specific instructions to do so.
Minimally, you should submit the following files:

- MyArrayString.java
- MyArrayStringTest.java

Figure 10.3: WebTA startup screen.

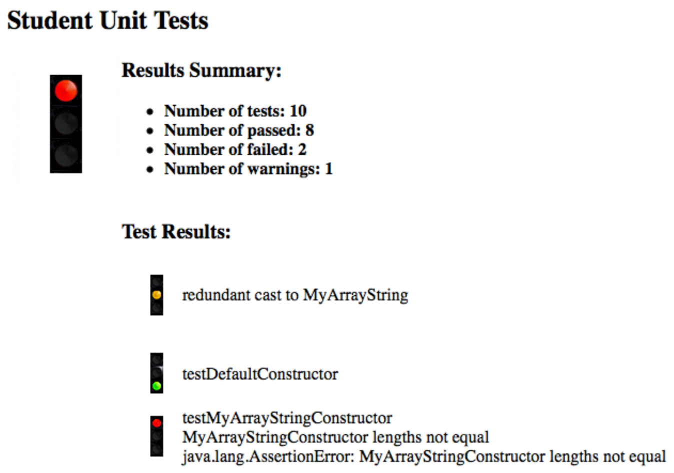


Critique Summary

- Compilation succeeded!
- There were 1 warnings, which could indicate subtle problems in the code.
- You passed all the instructor tests.
- You passed all the secret tests.
- You passed some student tests, but failed others.
- You passed some reverse tests, but failed others.
- I found some style issues in your code. See below.

Figure 10.4: WebTA critique summary.

edge cases (Fig. 10.5).



Student Unit Tests

Results Summary:

- **Number of tests: 10**
- **Number of passed: 8**
- **Number of failed: 2**
- **Number of warnings: 1**

Test Results:

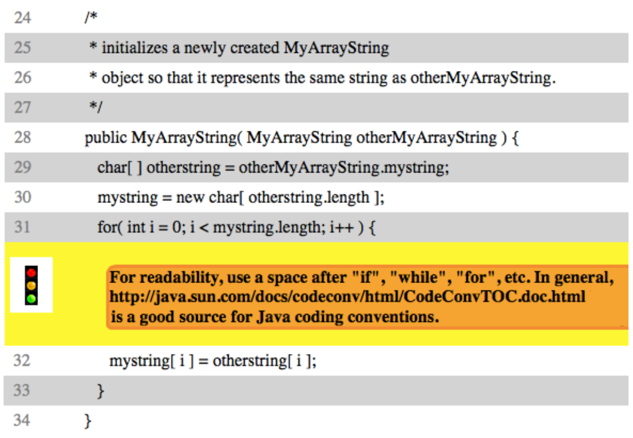
redundant cast to MyArrayString

testDefaultConstructor

testMyArrayStringConstructor
MyArrayStringConstructor lengths not equal
java.lang.AssertionError: MyArrayStringConstructor lengths not equal

Figure 10.5: WebTA student tests.

Scrolling further down the code critique, students find a listing of each code file submitted that includes style advice generated via textual analysis of the code (Fig. 10.6).



```
/*
 * initializes a newly created MyArrayString
 * object so that it represents the same string as otherMyArrayString.
 */
public MyArrayString( MyArrayString otherMyArrayString ) {
  char[ ] otherstring = otherMyArrayString.mystring;
  mystring = new char[ otherstring.length ];
  for( int i = 0; i < mystring.length; i++ ) {
```

For readability, use a space after "if", "while", "for", etc. In general, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html is a good source for Java coding conventions.

```
    mystring[ i ] = otherstring[ i ];
  }
}
```

Figure 10.6: WebTA style critique.

Students using WebTA are engaged in Learning by Doing [52]. Instructors provide students with authentic problems. While developing solutions to problems, students engage in an iterative conversation: developing code, receiving critiques, reflecting on feedback, and revising their solutions. WebTA applies Cognitive Apprenticeship practices that role-model authentic skills for students. Students are repeatedly exposed to patterns of coding and critiques from which they learn how to identify and communicate about issues that crop up during software development.

Features of WebTA include

- code compilation with student-friendly explanations of errors and warnings;
- rigorous, assignment-based unit test shakedown of student code, featuring both student-visible test to guide their code development and hidden tests to exercise their inquiry skills;
- evaluation of student test code, to support them as creative testers;
- textual analysis of source code, fully customizable by the instructor, to provide feedback on coding style;
- built-in plagiarism detection;

- preliminary grade assessment, for use by instructors or teaching assistants as a basis for final scores.

10.1.4 A WebTA walkthrough

We provide a brief walkthrough of the WebTA critique process. For space reasons, we focus on the student perspective and provide a synopsis of the instructor actions.

10.1.4.1 Instructor setup

The instructor formulates the following assignment:

- Develop a class named *Fibonacci* that contains two methods: *fibIteration* and *fibRecursion* that given an integer n returns the long n^{th} element of the Fibonacci Sequence using iterative and recursive methods, respectively. if n is out of range, return -1.

For this problem, the instructor sets up three tests, one of them being secret. The tests, using JUnit test conventions, are as follows:

```
@Test
@CanvasTaTest( points = 5, name = "fibIterative Test.",
description = "Checks for all values of n from 0 to 9.",
hint = "Remember to track the previous two values." )
public void fibIterativeTest( ) {
  Fibonacci prog = new Fibonacci( );
  long[ ] solutions = {
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
  };
  for( int n = 0; n < solutions.length; n++ ) {
    long result = prog.fibIterative( n );
    if (solutions[ n ] != result) {
      fail( "fibIterative( " + n + " ) = " + solutions[ n ]
        + ".  Your method returned " + result );
    }
  }
}
@Test
@CanvasTaTest( points = 5, name = "fibRecursive Test.",
description = "Checks for all values of n from 0 to 9.",
hint = "Remember to test for the base case." )
public void fibRecursiveTest( ) {
  Fibonacci prog = new Fibonacci( );
  long[ ] solutions = {
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377
  };
  for( int n = 0; n < solutions.length; n++ ) {
    long result = prog.fibRecursive( n );
    if (solutions[ n ] != result) {
      fail( "fibRecursive( " + n + " ) = " + solutions[ n ]
        + ".  Your method returned " + result );
    }
  }
}
// Secret Test
@Test
@CanvasTaTest( points = 5, name = "fibRecursive Edge Case",
description = "Checks for n = -1" )
public void fibRecursiveEdgeCaseTest( ) {
  Fibonacci prog = new Fibonacci( );
  long result = prog.fibRecursive( -1 );
  if ( result != -1 ) {
    fail( "fibRecursive( " + -1 + " ) = " + -1
      + ".  Your method returned " + result );
  }
}
```

The instructor also creates (or reuses existing) rules for the style critic. Rules consist of a regular expression, that matches with a problematic code snippet, coupled with advice the instructor would give a student who wrote such code. For example, anticipating that the student might not space code within parentheses, such as:

```
for(int i = 0;i < n; i++){
  x = y;
  y = x + y;
}
```

The instructor might develop a rule whose trigger is matches a form “(EXPR)” and whose advice is “For readability, use a space after (and before)”.

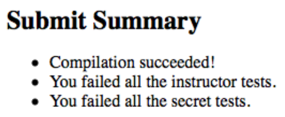
10.1.4.2 Student development

When students submit the assignment, it is uploaded to WebTA for testing and analysis. Depending on the instructor configurations, student code is compiled, tested against instructor tests, secret instructor tests, the students own tests, and the students test can even be run against an instructor solution to help students develop better coverage in testing. Furthermore, source files uploaded by the student are subjected to textual analysis to provide feedback on their programming style.

In this case, the student submits the following code skeleton, which should fail all tests.

```
public class Fibonacci {
  //Iterative method
  public long fibIterative(int n) {
    return 0;
  }
  //Recursive method
  public long fibRecursive(int n) {
    return 0;
  }
}
```

WebTA provides immediate feedback in the form of a critique report that the student can print. The critique report contains an executive summary of results, as well as a detailed listing of all errors and warnings encountered during the various kinds of analysis performed (as configured by the instructor.)



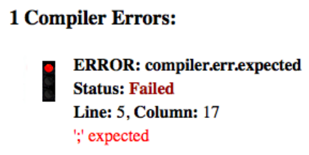
Submit Summary

- Compilation succeeded!
- You failed all the instructor tests.
- You failed all the secret tests.

Figure 10.7: First Pass Executive Summary.

Fig. 10.7 shows that the students skeleton code compiled, but all tests failed as expected. So now the student tackles the *fibIterative* method, which seems like it should be easiest to implement.

```
public class Fibonacci {
  //Iterative method
  public long fibIterative( int n ) {
    long x = 0;
    long y = 1
    for( int i = 0; i < n; i++ ) {
      x = y;
      y = x + y;
    }
    return x;
  }
  //Recursive method
  public long fibRecursive(int n) {
    return 0;
  }
}
```

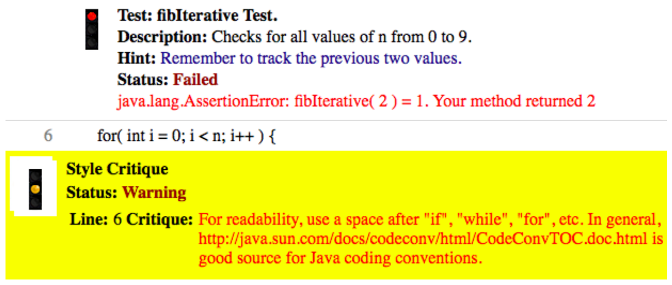


1 Compiler Errors:

ERROR: compiler.err.expected
Status: Failed
Line: 5, Column: 17
';' expected

Figure 10.8: Second Pass Compile Time Error.

Oops — the student missed a semicolon on line 17 (Fig. 10.8). The student makes a quick fix and resubmits the code. This time, WebTA indicates that the test for *fibIterative* failed and a style issue was found (Fig. 10.9).



Test: fibIterative Test.
Description: Checks for all values of n from 0 to 9.
Hint: Remember to track the previous two values.
Status: Failed
java.lang.AssertionError: fibIterative(2) = 1. Your method returned 2

```
6       for( int i = 0; i < n; i++ ) {
```

Style Critique
Status: Warning
Line: 6 Critique: For readability, use a space after "if", "while", "for", etc. In general, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html is good source for Java coding conventions.

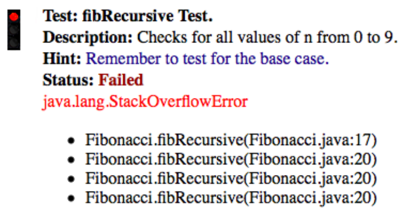
Figure 10.9: Third Pass: Test failure and style critique.

The student fixes the style issue by adding a space between the **for** keyword and the opening parenthesis. The test failure is a more difficult matter. The student adds a main method and prints the first ten values. It is evident that the results are not the Fibonacci Sequence. But what is wrong? Fortunately, WebTA has provided a hint, “Remember to track the previous two values”. Taking this into consideration, the student realizes that she did not implement the formula correctly and makes some changes.

```
public class Fibonacci {
  // Iterative method
  public long fibIterative( int n ) {
    long x = 0;
    long y = 1;
    long z = 1;
    for ( int i = 0; i < n; i++ ) {
      x = y; // fib(n)
      y = z; // next fib(n-2)
      z = x + y; // next fib(n-1)
    }
    return x;
  }
  // Recursive method
  public long fibRecursive( int n ) {
    return 0;
  }
  // TEST CODE
  public static void main( String[ ] args ) {
    Fibonacci self = new Fibonacci( );
    System.out.println( "n ITERATIVE RECURSIVE" );
    for( int n = 0; n < 10; n++ ) {
      long fibI = self.fibIterative( n );
      long fibR = self.fibRecursive( n );
      System.out.printf( "
    }
  }
}
```

This time the student passes the *fibIterative* test. Moving on, the student implements and submits the *fibRecursive* method.

```
//Recursive method
public int fibRecursive(int n) {
  if (n > 0 ) {
    return n;
  }
  return fibRecursive(n - 1) + fibRecursive(n - 2);
}
```



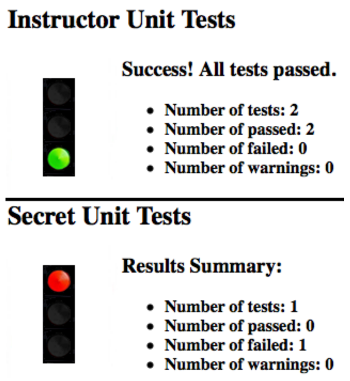
Test: fibRecursive Test.
Description: Checks for all values of n from 0 to 9.
Hint: Remember to test for the base case.
Status: Failed
java.lang.StackOverflowError

- Fibonacci.fibRecursive(Fibonacci.java:17)
- Fibonacci.fibRecursive(Fibonacci.java:20)
- Fibonacci.fibRecursive(Fibonacci.java:20)
- Fibonacci.fibRecursive(Fibonacci.java:20)

Figure 10.10: Student encounters infinite recursion.

The *fibRecursive* test fails due to infinite recursion (Fig. 10.10). The student, new to recursive algorithms, spends some time on this before connecting the hint about testing the base case to the fact that the base case includes 0! That's a quick fix (post epiphany).

```
//Recursive method
public int fibRecursive(int n) {
  if (n = 0 || n = 1 ) {
    return n;
  }
  return fibRecursive(n - 1) + fibRecursive(n - 2);
}
```



Instructor Unit Tests

Success! All tests passed.

- **Number of tests: 2**
- **Number of passed: 2**
- **Number of failed: 0**
- **Number of warnings: 0**

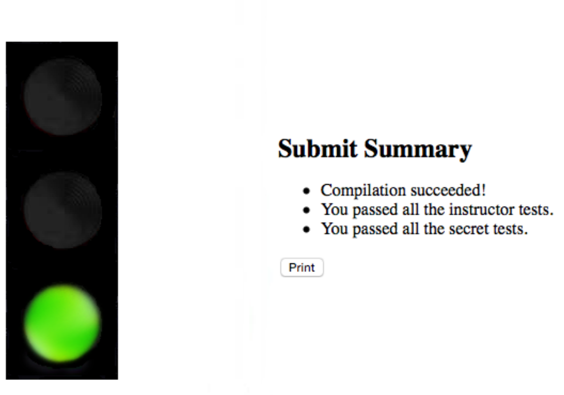
Secret Unit Tests

Results Summary:

- **Number of tests: 1**
- **Number of passed: 0**
- **Number of failed: 1**
- **Number of warnings: 0**

Figure 10.11: Student Passes Instructor Tests, But Fails Secret Test.

The student is almost there. All Instructor Tests are passed, but the Secret Test is still failing (Fig. 10.11). No information is given to guide the student on a secret test so the student has to resort to manual debugging and rereading the specification. In this case, the assignment says that values of n that are out of range should cause the methods to return -1. With a final modification to the code, all tests pass!



Submit Summary

- Compilation succeeded!
- You passed all the instructor tests.
- You passed all the secret tests.

Figure 10.12: Student Passes All Tests

Assignment submission. When it comes time to submit an assignment, students can submit to WebTA and get immediate feedback about how well their code compiled and tested via the Assignment Submit tool. When creating an assignment, instructors set the Submission Type to External Tool and select WebTA Assignment Submit. Instructor configuration is similar to the Code Critique tool. Students see the WebTA submit page at the bottom of their assignment.

Upon submitting, students receive an executive summary of how their code performed. This does not indicate a grade. However, a preliminary grade report is saved for a human TA to review before assigning a grade.

10.1.4.3 Instructor evaluates submissions

Through WebTA, the instructor may view student submissions, and select a student to see the students' code and a preliminary score assigned by WebTA.

The instructor then examines the results and the student's code, providing additional comments and feedback or grade modifications. Once a grade has been assigned, WebTA sends a grade report to the student.

10.2 The Future of WebTA

WebTA will continue to be used as a research platform. After some Instructor-side UX work, I plan to release WebTA to the open-source community at the end of 2020. I am developing a graduate-level course in which WebTA will be a teaching platform. With MATLAB-TA, Marissa Walther and I began exploring adapting WebTA to other languages and other disciplines. I plan to continue this work.

Chapter 11

Corpus of Novice Code Submissions

11.1 Corpus of Code

In this chapter we discuss data collected by WebTA, some interesting results, and the future work these results suggest. We started collecting data in 2014. Since that time, the system has been used by 1,421 students in 27 courses. These students made 64,964 submissions to 119 assignments. The system generated 14,650,677 individual critiques indicating issues detected in submissions, including compile-time errors and warnings, run-time errors and warnings, and style issues.

11.2 Results from initial beta testing of WebTA

2014-2015 was our first year of deployment. We beta tested WebTa in two courses (Intro to Programming 1, Data Structures), each with approximately 100 students enrolled. There have been growing pains: some technological, some perceptual.

Technologically, we experienced problems with server load and browser incompatibility. There was a period of time in the fall when several major browsers were pushed security updates and WebTa stopped working with all but the Chrome browser. During crunch periods around midterms and finals week, we experienced severe server lag, making it difficult for students to submit code or for WebTA to execute it within the specified thread time-out parameters. Working with our IT department, we have resolved most of these issues. Server load during crunch times is still an issue when over one hundred panicked students make last minute submissions in the hours and minutes before the due date. My understanding is that peak server load is an issue for most institutions developing autograders and code critiquers.

Unfortunately, these issues create perception problems with the students:

- "[WebTA] is also a bit difficult because we can't access it from our own computers."

Yet many more students have expressed an appreciation for WebTA:

- "I like [WebTA] because it shows be where my error is or which test is wrong so I can spend more time on fixing it rather than taking forever to search for the error."

- "[WebTA] gives good input on style and how to fix my errors."
- "I really enjoyed CanvasTA. Mostly every aspect of it was very helpful. I really loved how quick it was to simply drag and drop my .java file in and simply click to have it run its checks and turn it in. The checks and some of the style tips it made were also very helpful. There were some bad programming practices I've done that I never realized before until I read through its style suggestions."

WebTA was beta tested in the Fall 2014 Data Structures course and in the Spring 2015 Introduction to Programming course. Programming project scores compared to the previous year were higher (Fig. 11.1), but more study is required to determine if the difference in scores is solely or in part attributable to WebTA and to identify other influencing factors.

Course	Semester	Mean	Median	Mode	Standard Deviation
CS1121	Spring 2015	80.4	95.0	99.8	28.0
CS1121	Spring 2014	72.4	77.7	93.2	22.7
CS2321	Fall 2014	82.9	86.6	95.4	17.6
CS2321	Fall 2013	75.0	80.8	59.4	20.4

Figure 11.1: Programming project scores.

Qualitatively, we have the sense that more effort needs to focus on fading scaffolds and teaching students how to test their code.

Data Structures students were required to submit JUnit test cases with their code during both Fall 2013 and 2014 semesters. WebTA tested their JUnit tests against the assignment API. Over the course of Fall 2014 we saw marked improvement in student conformance to the specified API. However, we also noticed students who, upon failing an attempt to test an edge case, would remove their test method to eliminate the problem, instead of trying to understand the edge case and fixing their test.

Some instructors have voiced concern that students might be relying on WebTA to test their code. When asked how he used WebTA, one student responded:

- "I mainly used it for testing purposes. It was great that it gave me the results and failures so I could go back and try to figure out what went wrong. On the downside I think it made me put a little less effort in actual testing myself though I ended up having to anyway to fix some of the errors it showed."

Another student said:

- "Whenever I felt that I had working code (i.e. I fixed any bugs I could think or the bugs pointed out by [CanvasTA]) I would submit my program file to [CanvasTA] to see if it passed or not."

Based on an informal in-class survey, second-year Data Structures students were more accepting and less critical of WebTA while first-year Intro students, who had never used a different system for assignment submission, provided more critical feedback.

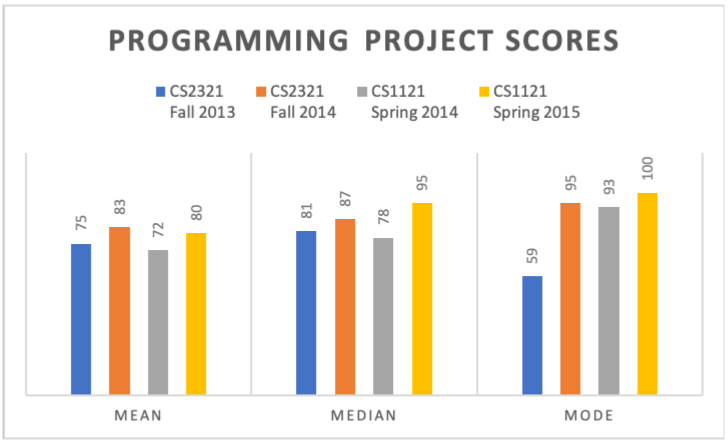


Figure 11.2: Programming project scores.

11.3 Future Work: Analysis of Corpus Data

Critiquer systems provide students with prompt feedback. However, when the system provides inappropriate feedback, such as false identification of a problem, or cryptic error messages that are difficult to understand, novice students may become confused and discouraged whereas experts are able to make more appropriate use of feedback messages.

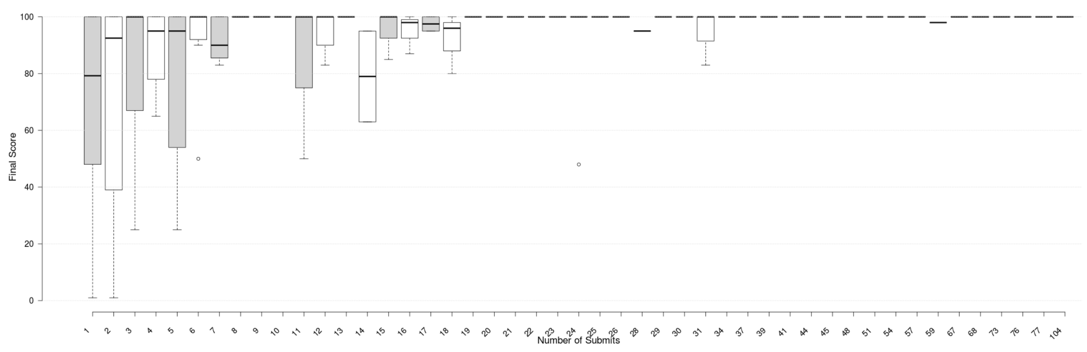
A study by Munson [39] suggests that, when given a list of critiques, students will address the first feedback message about 52% of the time. Higher assignment scores are positively associated with addressing the first error. Novice programmers reported problems understanding compiler generated error messages.

We need to analyze the WebTA data to form an understanding of student utilization of error messages between submissions. Can we use our data to determine the order in which students solve errors in the code? Do different kinds of critiques receive a higher priority? Can we determine if our critiques are more useful to the students than raw error messages? How does feedback generated by a false positive affect their path to a solution?

I plan to compare the feedback given between consecutive submissions to determine which critiques the student focused on between submissions.

11.3.1 Why do some students submit more?

The average number of submits by students per assignment is 8.85 with a standard deviation of 12.63 with the maximum being 192 submits by a student.



course	assignment	user #	# submits	submission scores
CS1121	5159768	306	7	65, 73, 86, 81, 95, 100, 100
CS1121	5159768	334	7	50, 50, 50, 50, 50, 85, 85
CS1121	5159768	346	24	50, 50, 50, 50, 50, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, . . .
CS1121	5159768	370	14	48, 63, 63, 56, 56, 56, 56, 63, 63, 56, 56, 63, 65, 63

Figure 11.3: Submission Scores.

11.3.2 Efficacy & Impact of Stoplight.

The course-grained stoplight metaphor indicator:

- GREEN: No antipatterns detected.
- YELLOW: Only non-show-stopping antipatterns detected.
- RED: Critical failures detected.

8% of student final submission scores on assignments were less than the student's max submission score.

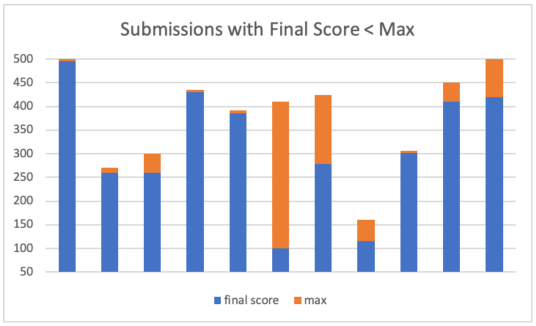


Figure 11.4: Final Score < Max.

11.3.3 Which issues take the longest to address?

- What are the most frequent critiques encountered by students?
- What are the hardest critiques for students to resolve?
- How do student issues change across the semester?

course	assignment	user #	# submits	submission timestamps
CS 1121	5159768	273941	1	1572797632
CS 1121	5159768	274354	5	1572797639, 1572797640, 1572797640, 1572797641, ...
CS 1121	5159768	280458	3	1572797796, 1572797798, 1572797799
CS 1121	5159768	552974	1	1572798708
CS 1121	5159768	571280	1	1572798144
CS 1121	5159768	581689	2	1572798678, 1572798681
CS 1121	5159768	583753	28	1572802356, 1572802362, 1572802368, 1572802375, ...
CS 1121	5159768	955186	24	1572797579, 1572797579, 1572797579, 1572797580, ...
CS 1121	5159768	956615	1	1572798693

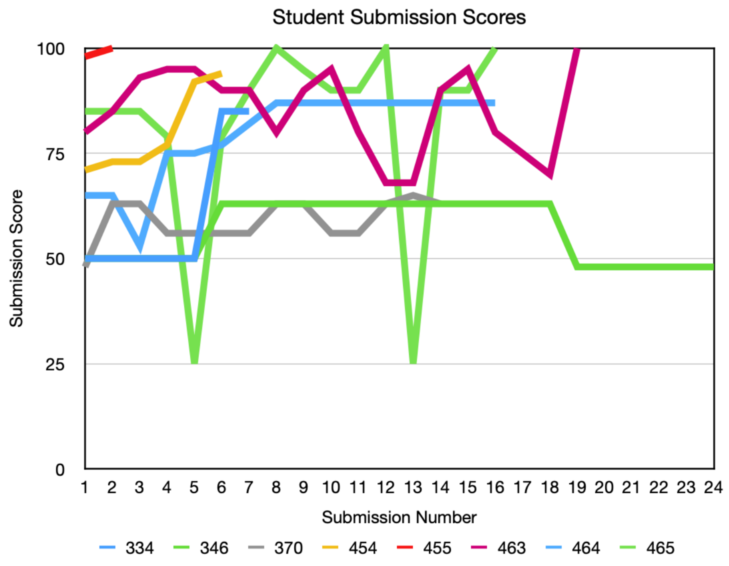
Figure 11.5: Submission Times.

11.3.4 Can we identify struggling students?

As many as one-third of incoming students fail their first CS course. Estey and Coady [23] examined interaction patterns could identify struggling students. Their analysis of 652 students over three semesters highlighted a number of predictors for success. Their work suggests that struggling students can be identified early in the semester.

Estey and Coady found a correlation between the number of hints received and the frequency of compilation that can be used as an indicator of struggling students. Our data contains the frequency of submission and the number of critiques provided, can we find a similar correlation in our data?

When we look at submission scores as students resubmit code for an assignment, we see some patterns that may indicate struggling students. What is happening when we see students making several submissions with no change in score? What is happening when we see students repeatedly improving their score only to experiencing a large drop in score for just one submission? These variations in submission score need to be investigated to determine if they can help identify struggling students.



11.3.5 Analyzing Use of Critiques

- ▶ How are student using critiques to improve their code?
- ▶ In which order do students address critiques?
- ▶ Do better students address one or more critiques?
- ▶ Does the order and number in which students address critiques change as the course progresses?

11.3.6 Can we utilize machine learning?

Can we utilize machine learning techniques to

- ▶ improve or enhance the detection of antipatterns in student submissions?
- ▶ identify new antipatterns in our corpus of student code?

11.3.7 Analysis of Student Errors

Altadmri and Brown [6] analysed a compilation data from over 250,000 students in their large Blackbox data set. They examined the frequency, time-to-fix, and spread of errors among users. These factors can be used to identify the most frequent (or hardest to fix) errors.

Their work utilized compile-time errors and lexical analysis to identify errors in student code. Our dataset contains this information plus the results of shake-down testing. Using this information can we discover new antipatterns made by introductory cs students?

Additionally, Altadmri and Brown analyzed time-to-fix information. Syntax errors were the most popular category of errors among novice programmers, but were also the quickest to fix. Semantic and Types errors exhibited longer time-to-fix values. Could we use a similar analysis to identify struggling students?

The WebTA data contains submission time data, error messages and stacktraces, and all feedback provided for each submission. Using this data, I should be able to the same factors as Altadmri and Brown.

11.3.8 Detection of Code Smells

The presence of code smells can negatively impact the quality of a program. Khomh, et al. developed a Bayesian approach to detecting code smells [34]. Their paper presents a systematic process for converting existing detection rules to a probabilistic model. They illustrate the process by generating a model to detect the Blob Antipattern, validating the model, and measuring its accuracy.

Code smells are usually indicative of design antipatterns. Can we adapt their technique to grovel over our corpus of student submission to automatically detect new antipatterns that can be added to our critique library?

Part IV

Catalog of Patterns & Antipatterns

Chapter 12

Antipattern Library

12.1 Identifying New Antipatterns

Through the course of this research, I've developed a catalog of over 200 antipatterns that instructors can draw on. My ad hoc process of identifying antipatterns to add to the library involves combing the literature and community standards to identify bugs, traps, and coding patterns that novices might encounter, utilize, or design. Beyond the literature and community standards, the most valuable resource available are instructors who bring both coding experience and pedagogical knowledge to the table. Lastly spend lots of time scrutinizing student code.

To recap: I identify new patterns for the library by

1. combing the literature,

2. exploring community standards,

3. drawing on instructor experience,

4. and scrutinizing student code.

Once I've identified several antipatterns, I reflect on them and ask myself:

1. How can the antipattern be detected in student code?

2. Can the issues, solutions, debugging be explained to novices?

3. Why do novices exhibit the antipattern?

For example, a common issue is students declaring a class-level variable, but only using it in a single, non-trivial method. Students may be trying to develop flexible code, but they are misunderstanding the principles of modularity and encapsulation. (See Listing 12.1) This antipattern is only possible in simple scenarios. Novice programmers get away with it because assignments are limited in scope and very few instances are created.

12.1.1 Antipattern: Localized Instance Variable

Listing 12.1: Localized Instance-Variable

```java
public class Reverse {
  String result = "";
  public String reverseString( String s ) {
    for(int i = 0; i < s.length(); i++) {
      result = s.charAt( i ) + result;
    }
    return result;
  }
}
```

Another commonly seen antipattern is the Magic Incantation, where students insert code just because it might be necessary. For example, adding an empty loop (Listing 12.2), which has no effect, to the code. This sometimes happens when a students has just learned a topic and think it should be used everywhere. It indicates an incomplete understanding of the effects of using the code.

12.1.2 Antipattern: Magic Incantation

Listing 12.2: Invoking code as incantation

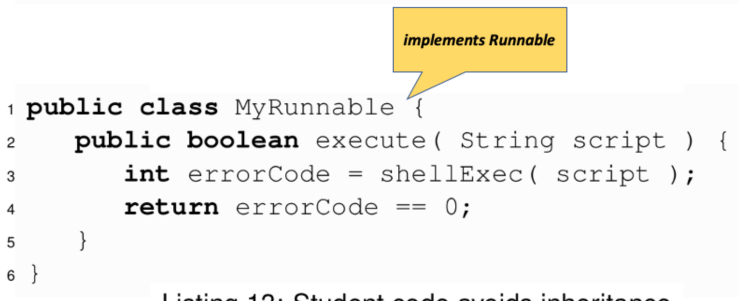
```
public double abs( double  d ) {
  double result = d;
  if ( d < 0 ) {
      for( int i = 0; i < 10; i++ ) {
      }
      result = -d;
  }
  return result;
}
```

Finally, the Inheritance Pseudo-Implementation antipatter occurs when students have a buggy understanding of inheritance. When provided with an interface or abstract class, instead of inheriting, they re-implement the code. Thus they are not inheriting anything and they are violating polymorphism. (Listing 12 & 13)

12.1.3 Antipattern: Inheritance Pseudo-Implementation

Listing 12: Interface provided to student.

```
public interface Runnable {
   public boolean execute( String script );
}
```



```
public class MyRunnable {
   public boolean execute( String script ) {
      int errorCode = shellExec( script );
      return errorCode == 0;
   }
}
```

Listing 13: Student code avoids inheritance.

12.2 Structural Antipatterns

These patterns deal with the structure or syntax of the code. Novices are in the process of learning the syntax of the language. During this process, students can form misconceptions concerning the rules of the language. It is critical that these misconceptions be identified early.

12.2.1 BAD-ARGUMENT-TYPES

Type: Structure

Source: Diagnostic

Description Triggered when a method is called with incompatible argument types.

Example:

Listing 12.3: Code exemplar for BAD-ARGUMENT-TYPES

```java
public class MethodCannotBeAppliedToGivenTypes {
  public static int absoluteValue( int number ) {
    return number >= 0 ? number : -number;
  }

  public static void main( String [ ] args ) {
    System.out.println( "The absolute value of -4 ↩
       ↪ is " + (absoluteValue( -4L )) );
  }
}
```

Description Triggered when a method is called with incompatible argument types.

12.2.2 BAD-TYPES-FOR-BINARY-OPERATOR

Type: Structure

Source: Diagnostic

Description Triggered when the types of the values on either side of a binary operator do not match.

Description Triggered when the types of the values on either side of a binary operator do not match.

12.2.3 BAD-TYPES-FOR-COMPARISON-OPERATOR

Type: Structure

Source: Diagnostic

Description Triggered when the types of the values on either side of a binary operator do not match.

Example:

Listing 12.4: Code exemplar for BAD-TYPES-FOR-COMPARISON-OPERATOR

```java
import java.util.List;
public class BadOperandTypesForComparisonOperator ↩
    ↪ {
  public static void bubbleSort( List<String> list↩
      ↪  ) {
    for ( int i = 0; i < list.size()-1; i++ ) {
      for ( int j = i+1; j < list.size(); j++ ) {
        if ( list.get(j-1) > list.get(j) ) {
          String temp = list.get(j-1);
          list.set(j-1, list.get(j));
          list.set(j, temp);
        }
      }
    }
  }
}
```

Description Triggered when the types of the values on either side of a binary operator do not match.

12.2.4 BAD-TYPES-FOR-UNARY-OPERATOR

Type: Structure

Source: Diagnostic

Description Triggered when the type of the value applied to a unary operator is invalid for that operation.

Example:

Listing 12.5: Code exemplar for BAD-TYPES-FOR-UNARY-OPERATOR

```
public class BadOperandTypesForUnaryOperator {
  public String incrementStringCounter( String ↩
      ↪ counter ) {
    return ++counter;
  }
}
```

Description Triggered when the type of the value applied to a unary operator is invalid for that operation.

12.2.5 BAD-TYPE-IN-CONDITIONAL-EXPRESSION

Type: Structure

Source: Diagnostic

Description Triggered when the ternary operator evaluates to an unexpected type.

Example:

Listing 12.6: Code exemplar for BAD-TYPE-IN-CONDITIONAL-EXPRESSION

```
public class BadOperandTypesForBinaryOperator {
  public static String stringSubtraction( String ↩
     ↪ minuend, String subtrahend ) {
    return minuend - subtrahend;
  }
}
```

Description Triggered when the ternary operator evaluates to an unexpected type.

12.2.6 CANNOT-FIND-SYMBOL

Type: Structure

Source: Diagnostic

Description Triggered when a symbol (variable/method/class name) is evaluated before it is declared.

Example:

Listing 12.7: Code exemplar for CANNOT-FIND-SYMBOL

```
public class CannotFindSymbol {
  public static int absoluteValue( int number ) {
    result = number;
    if ( result < 0 ) {
      result = -result;
    }
    return result;
  }
}
```

Description Triggered when a symbol (variable/method/class name) is evaluated before it is declared.

12.2.7 CANNOT-INFER-TYPE-ARGUMENTS

Type: Structure

Source: Diagnostic

Description Triggered when there is either incomplete or inconsistent information concerning the type arguments for a generic structure.

Example:

Listing 12.8: Code exemplar for CANNOT-INFER-TYPE-ARGUMENTS

```
import java.util.ArrayList;
import java.util.Arrays;

public class CannotInferTypeArguments<E> {
  public static void main ( String [ ]  args ) {
    ArrayList<String> list = new ArrayList<>( ↩
       ↪ Arrays.asList( "1", 2, "3", 4 ));
  }
}
```

Description Triggered when there is either incomplete or inconsistent information concerning the type arguments for a generic structure.

12.2.8 CLASS-CAST-EXCEPTION

Type: Structure

Source: Exception

Description Trigger at when casting between incompatible class types.

Example:

Listing 12.9: Code exemplar for CLASS-CAST-EXCEPTION

```
class A {
  int i = 10;
}
class B extends A {
  int j = 20;
}
public void classCastException() {
  A a = new A( );
  B b = ( B ) a;
}
```

Description Trigger at when casting between incompatible class types.

12.2.9 CLASS-INTERFACE-ENUM-EXPECTED

Type: Structure

Source: Diagnostic

Description Triggered when the compiler encounters unexpected code outside the context of a class block.

Example:

Listing 12.10: Code exemplar for CLASS-INTERFACE-ENUM-EXPECTED

```java
// Generates class, interface, or enum expected error
// Code is outside of class structure
for( int i = 0; i < 10; i++ ) {
  System.out.println( i );
}

public class ClassInterfaceEnumExpected {

}
```

Description Triggered when the compiler encounters unexpected code outside the context of a class block.

12.2.10 CLASS-NOT-SAME-NAME-AS-FILE

Type: Structure

Source: Diagnostic

Description Triggered when a public class is declared in a file with a different filename.

Example:

Listing 12.11: Code exemplar for CLASS-NOT-SAME-NAME-AS-FILE

```java
public class FilenameAndClassNameAreDifferent {

}
```

Description Triggered when a public class is declared in a file with a different filename.

12.2.11 CONSTRUCTOR-RETURN-TYPE

Type: Structure

Source: Code

Description Triggered with a constructor has a return type.

Example:

Listing 12.12: Code exemplar for CONSTRUCTOR-RETURN-TYPE

```java
public class ConstructorWithReturnType<E> {
  public int ConstructorWithReturnType ( int ↩
     ↪ number ) {
    this.number = number;
    return number;
  }
}
```

Description Triggered with a constructor has a return type.

12.2.12 DIVISION-BY-ZERO

Type: Structure

Source: Diagnostic

Description Triggered when the compiler detects integer division by zero.

Example:

Listing 12.13: Code exemplar for DIVISION-BY-ZERO

```java
public class DivisionByZero {
  public static final int DIVISOR = 0;
  private Integer divideByConstant( int dividend )↩
     ↪  {
    return dividend / DIVISOR;
  }
}
```

Description Triggered when the compiler detects integer division by zero.

Repair: As this is detectable by the compiler, look for an expression with an obvious division by zero. Either by a numeric literal, a variable initialized to 0 with no changes before the division, or a constants set to 0 and used as a denominator.

Best Practices

- Be mindful of the type preservation rule, which states that arithmetic omputations between two integers must produce an integer result.

- When using integers in an expression that contains a division, be verify that the denominator will not equal 0.

12.2.13 EXTENDS-OBJECT

Type: Structure

Source: Code

Description Triggered when a class extends Object.

Example:

Listing 12.14: Code exemplar for EXTENDS-OBJECT

```java
public class ExtendsObject extends Object {


}
```

Description Triggered when a class extends Object.

12.2.14 IDENTIFIER-EXPECTED

Type: Structure

Source: Diagnostic

Description Triggered when the compiler encounters an operator when expecting an identifier.

Example:

Listing 12.15: Code exemplar for IDENTIFIER-EXPECTED

```
public class IdentifierExpected {
  public static int absoluteValue( int num ) {
    int result = num;
  }
    return num >= 0 ? num : - ;
  }
}
```

Description Triggered when the compiler encounters an operator when expecting an identifier.

12.2.15 ILLEGAL-START-OF-EXPRESSION

Type: Structure

Source: Diagnostic

Description Triggered when the compiler encounters something unexpected.

Example:

Listing 12.16: Code exemplar for ILLEGAL-START-OF-EXPRESSION

```
public class IllegalStartOfExpression {
  public static int absoluteValue( int num ) {
    public int result = num;
```

```
        return num >= 0 ? num : -num;
    }
}
```

Description Triggered when the compiler encounters something unexpected.

Repair: Walk through the code talking aloud about the syntax. Pay attention to possible missing parentheses, curly brackets, or semicolons. Check for methods declared inside methods - not allowed in Java. Finally, check for cases where public, private, or protected modifiers are used within a method - also not allowed.

Best Practices

- Be careful to place a semicolon at the end of every statement.
- Methodically balance parenthesis and curly brackets.
- Do not declare methods inside methods.
- Do not use access modifiers inside methods.
- Indent your code to help balance curly braces.

12.2.16 ILLEGAL-START-OF-TYPE

Type: Structure

Source: Diagnostic

Description Triggered when the compiler encounters something unexpected.

Example:

Listing 12.17: Code exemplar for ILLEGAL-START-OF-TYPE

```
public class IllegalStartOfType {
  public static int absoluteValue( int num ) {
    int result = num;
  }
    return num >= 0 ? num : -num;
  }
}
```

Description Triggered when the compiler encounters something unexpected.

Repair: Walk through the code talking aloud about the syntax. Pay attention to possible missing parentheses, curly brackets, or semicolons. Check for methods declared inside methods - not allowed in Java. Finally, check for cases where public, private, or protected modifiers are used within a method - also not allowed.

Best Practices

- Be careful to place a semicolon at the end of every statement.
- Methodically balance parenthesis and curly brackets.
- Do not declare methods inside methods.
- Do not use access modifiers inside methods.
- Indent your code to help balance curly braces.

12.2.17 IMPORTS-JAVA.LANG

Type: Structure

Source: Code

Description Triggered when the code imports java.lang, which is automatically imported.

Example:

Listing 12.18: Code exemplar for IMPORTS-JAVA.LANG

```java
import java.lang.Math;
```

Description Triggered when the code imports java.lang, which is automatically imported.

12.2.18 IMPORT-OWN-PACKAGE

Type: Structure

Source: Code

Description Triggered when code imports the package it resides within.

Example:

Listing 12.19: Code exemplar for IMPORT-OWN-PACKAGE

```java
package ed.mtu.cs;

import edu.mtu.cs.*;

public class ImportsOwnPackage {

}
```

Description Triggered when code imports the package it resides within.

12.2.19 INCOMPATIBLE-TYPES

Type: Structure

Source: Diagnostic

Description Triggered when a value of one type is used where a different type is expected.

Example:

Listing 12.20: Code exemplar for INCOMPATIBLE-TYPES

```
public class IncompatibleTypes {
  public static int absoluteValue( String number )↩
     ↪  {
    return number >= 0 ? number : -number;
  }
}
```

Description Triggered when a value of one type is used where a different type is expected.

12.2.20 MISSING-COLON-OR-ARROW

Type: Structure

Source: Diagnostic

Description Triggered when code is missing a colon or arrow operator, such as after a case or within a lambda expression.

Example:

Listing 12.21: Code exemplar for MISSING-COLON-OR-ARROW

```java
public class MissingColonOrArrow {
  private Double volume(String nameOfCurvedSolid, double height, double base, double radius) {
    Double volume = null;
    switch (nameOfCurvedSolid.toUpperCase()) {
      case "SPHERE"
        volume = 4 * Math.PI * Math.pow( radius, 2 );
        break;
      case "CYLINDER":
        volume = 2 * Math.PI * Math.pow( radius, 2 ) + 2 * Math.PI * radius * height;
        break;
      case "CONE":
        volume = Math.PI * radius * ( radius + Math.sqrt(
          Math.pow( height, 2 ) + Math.pow( radius, 2 ) ) );
        break;
    }
    return volume;
  }
}
```

Description Triggered when code is missing a colon or arrow operator, such as after a case or within a lambda expression.

12.2.21 MISSING-RETURN-STATEMENT

Type: Structure

Source: Diagnostic

Description Triggered when the method does not terminate with a return statement and the method type is not void.

Example:

Listing 12.22: Code exemplar for MISSING-RETURN-STATEMENT

```java
public class MissingReturnStatement {
  public static int absoluteValue( int number ) {
    int result = number >= 0 ? number : -number;
  }
}
```

Description Triggered when the method does not terminate with a return statement and the method type is not void.

12.2.22 MISSING-RETURN-VALUE

Type: Structure

Source: Diagnostic

Description Triggered when the a method, declared with a non-void return type, contains a standalone return statement.

Example:

Listing 12.23: Code exemplar for MISSING-RETURN-VALUE

```
public class MissingReturnValue {
  public String[ ] inplaceSort( String[] array1 ) {
    String[] array2 = new String[array1.length];
    System.arraycopy(array1, 0, array2, 0, array1.length);
    return;
  }
}
```

Description Triggered when the a method, declared with a non-void return type, contains a standalone return statement.

12.2.23 MISSING-SEPARATOR

Type: Structure

Source: Diagnostic

Description Triggered when code is missing a something, such as a semicolon or a parenthesis.

Example:

Listing 12.24: Code exemplar for MISSING-SEPARATOR

```java
public class MissingSomething {
  private Double volume(String nameOfCurvedSolid, double height, double base, double radius) {
    Double volume = null;
    int [] foo = new int[10
    switch (nameOfCurvedSolid.toUpperCase() {
      case "SPHERE":
        volume = 4 * Math.PI * Math.pow( radius, 2 );
        break;
      case "CYLINDER":
        volume = 2 * Math.PI * Math.pow( radius, 2 ) + 2 * Math.PI * radius * height
        break;
      case "CONE":
        volume = Math.PI * radius * ( radius + Math.sqrt(
         Math.pow( height, 2 ) + Math.pow( radius, 2 ) ) );
        break;
    }
    return volume;
  }
}
```

Description Triggered when code is missing a something, such as a semicolon or a parenthesis.

Repair: Debugging missing separators can be a challenge. The compiler doesn't always indicate where the separator goes. Students should start where the compiler indicates and carefully read back through their code looking for a place where the missing separator should be inserted. Alternatively, one of the separators may exist in the code by mistake, such as a remnant opening bracket after code refactoring, causing the compiler to look for an unintended and therefore nonexistent closing bracket.

Best Practices

- Deliberately balance all parentheses, curly-brackets, and square brackets.
- Place a semicolon at the end of every statement.
- Separate all elements in static array initialization with commas.

12.2.24 NONSTATIC-IN-STATIC-CONTEXT

Type: Structure

Source: Diagnostic

Description Most often triggered when calling an instance method from the main method.

Example:

Listing 12.25: Code exemplar for NONSTATIC-IN-STATIC-CONTEXT

```
public class NonStaticMethodFromStaticContext<E> {
  public String stringifyArray( int[ ] array ) {
    String arrayString = "[";
    if ( array.length > 0 ) {
      arrayString += array[ 0 ];
      for( int i = 1; i < array.length; i++ ) {
        arrayString += ", " + array[ i ];
      }
    }
    arrayString += "]";
    return arrayString;
  }

  public static void main ( String [ ]  args ) {
    int[ ] array = { 1, 2, 3 };
    System.out.println( stringifyArray( array ) );
  }
}
```

Description Most often triggered when calling an instance method from the main method.

12.2.25 NONSTATIC-VAR-STATIC-CONTEXT

Type: Structure

Source: Diagnostic

Description Most often triggered when accessing an instance variable from the main method.

Example:

Listing 12.26: Code exemplar for NONSTATIC-VAR-STATIC-CONTEXT

```
public class NonStaticVariableFromStaticContext<E>↩
   ↪ {
  int count = 0;

  public static void main ( String [ ]  args ) {
    count++;
  }
}
```

Description Most often triggered when accessing an instance variable from the main method.

12.2.26 NOT-A-STATEMENT

Type: Structure

Source: Diagnostic

Description Triggered when an expression stands alone on a line.

Example:

Listing 12.27: Code exemplar for NOT-A-STATEMENT

```java
public class NotAStatment {
  public static int absoluteValue( int number ) {
    number >= 0 ? number : -number;
  }
}
```

Description Triggered when an expression stands alone on a line.

12.2.27 POSSIBLE-LOSS-OF-PRECISION

Type: Structure

Source: Diagnostic

Description Triggered when more bits are assigned to a value than it can hold based on data type.

Example:

Listing 12.28: Code exemplar for POSSIBLE-LOSS-OF-PRECISION

```java
public class PossibleLossOfPrecision {
  public static int absoluteValue( long number ) {
    return number >= 0 ? number : -number;
  }
}
```

Description Triggered when more bits are assigned to a value than it can hold based on data type.

12.2.28 RAWTYPE-FOUND

Type: Structure

Source: Diagnostic

Description Triggered when generic class is used without specifying a parameterized type.

Example:

Listing 12.29: Code exemplar for RAWTYPE-FOUND

```java
import java.util.ArrayList;

public class RawTypeFound {
  ArrayList list = new ArrayList();
}
```

Description Triggered when generic class is used without specifying a parameterized type.

12.2.29 REACHED-END-OF-FILE-WHILE-PARSING

Type: Structure

Source: Diagnostic

Description Triggered when the parser unexpected reaches the end of the file; more code is expected.

Example:

Listing 12.30: Code exemplar for REACHED-END-OF-FILE-WHILE-PARSING

```
public class ReachedEndOfFileWhileParsing {
  public static int absoluteValue( int number ) {
    return number >= 0 ? number : - number;
  }
```

Description Triggered when the parser unexpected reaches the end of the file; more code is expected.

12.2.30 RETURN-TYPE-REQUIRED

Type: Structure

Source: Diagnostic

Description Triggered when the method signature is missing a return type.

Example:

Listing 12.31: Code exemplar for RETURN-TYPE-REQUIRED

```java
public class ReturnTypeRequired {
  public static absoluteValue( int number ) {
    return number >= 0 ? number : -number;
  }
}
```

Description Triggered when the method signature is missing a return type.

12.2.31 SUPER-DEFAULT

Type: Structure

Source: Code

Description Triggered by a call to the default super constructor.

Example:

Listing 12.32: Code exemplar for SUPER-DEFAULT

```java
public class SourceCodeExample {
  public SourceCodeExample( ) {
    super( );
  }
}
```

Description Triggered by a call to the default super constructor.

12.2.32 SUPPRESS-WARNINGS-RAWTYPES

Type: Structure

Source: Code

Description Triggered when students use @SuppressWarnings to hide rawtypes cast warnings.

Example:

Listing 12.33: Code exemplar for SUPPRESS-WARNINGS-RAWTYPES

```
@SuppressWarnings( "rawtypes" )
public void suppressRawTypes( ) {
  ArrayList list = new ArrayList( );
}
```

Description Triggered when students use @SuppressWarnings to hide rawtypes cast warnings.

12.2.33 SUPPRESS-WARNINGS-UNCHECKED

Type: Structure

Source: Code

Description Triggered when students use @SuppressWarnings to hide unchecked cast warnings.

Example:

Listing 12.34: Code exemplar for SUPPRESS-WARNINGS-UNCHECKED

```
@SuppressWarnings( "unchecked" )
public <E> void suppressUncheckedCast( ) {
  E[ ] array = (E[]) new Object[ 10 ];
}
```

Description Triggered when students use @SuppressWarnings to hide unchecked cast warnings.

12.2.34 UNCHECKED-ARRAY-CAST

Type: Structure

Source: Diagnostic

Description Triggered when there is an implicit or explicit cast from a specific array type to a generic array type.

Example:

Listing 12.35: Code exemplar for UNCHECKED-ARRAY-CAST

```
public class UncheckedArrayCast {
  public <E> E[] cloneArray( E[] array1 ) {
    E[] array2 = (E[]) new Object[array1.length];
    System.arraycopy(array1, 0, array2, 0, array1.length);
    return array2;
  }
}
```

Description Triggered when there is an implicit or explicit cast from a specific array type to a generic array type.

12.2.35 UNCHECKED-CALL

Type: Structure

Source: Diagnostic

Description Triggered when there is a call to a generic method whose parameterized type was not specified.

Example:

Listing 12.36: Code exemplar for UNCHECKED-CALL

```
public class UncheckedCall {
  public <E extends Comparable> void inplaceSort( E[] array ) {
    for( int i = 0; i < array.length - 1; i++ ) {
      for ( int j = i+1; j < array.length - i; j++ ) {
        if ( array[ j - 1 ].compareTo( array[ j ] ) > 0 ) {
          E temp = array[ j - 1 ];
          array[ j - 1 ] = array[ j ];
          array[ j ] = temp;
        }
      }
    }
    return;
  }
}
```

Description Triggered when there is a call to a generic method whose parameterized type was not specified.

12.2.36 UNCHECKED-CAST

Type: Structure

Source: Diagnostic

Description Triggered when there is an implicit or explicit cast from a specific type to a generic type. This often occurs in conjunction with the use of a rawtype.

Example:

Listing 12.37: Code exemplar for UNCHECKED-CAST

```
import java.util.ArrayList;
import java.util.Arrays;

public class UncheckedCast<E> {
  public ArrayList<E> duplicateList( ArrayList<↩
    ↪ Object> list1 ) {
    ArrayList<E> list2 = new ArrayList<>(  );
    for( int i = 0; i < list1.size(); i++ ) {
      list2.add( (E) list1.get( i ) );
    }
    return list2;
  }

  public static void main ( String [ ]  args ) {
    UncheckedCast<Integer> thing = new ↩
       ↪ UncheckedCast<>();
    ArrayList<Object> list = new ArrayList<>( ↩
       ↪ Arrays.asList( "1", 2, "3", 4 ));
    for( Integer i : thing.duplicateList( list ) )↩
       ↪  {
      System.out.println( i.getClass().getName() )↩
         ↪ ;
    }
  }
}
```

Description Triggered when there is an implicit or explicit cast from a specific type to a generic type. This often occurs in conjunction with the use of a rawtype.

12.2.37 UNEXPECTED-RETURN-VALUE

Type: Structure

Source: Diagnostic

Description Triggered when the a void method attempts to return a value.

Example:

Listing 12.38: Code exemplar for UNEXPECTED-RETURN-VALUE

```
public class UnexpectedReturnValue {
  public <E extends Comparable<E>> void ↩
     ↪ inplaceSort( E[] array ) {
```

```
    for( int i = 0; i < array.length - 1; i++ ) {
      for ( int j = i+1; j < array.length - i; j++↩
        ↪ ) {
        if ( array[ j - 1 ].compareTo( array[ j ] ↩
          ↪ ) > 0 ) {
          E temp = array[ j - 1 ];
          array[ j - 1 ] = array[ j ];
          array[ j ] = temp;
        }
      }
    }
    return array;
  }
}
```

Description Triggered when the a void method attempts to return a value.

12.2.38 UNMATCHED-DOUBLE-QUOTE

Type: Structure

Source: Diagnostic

Description Triggered when a string literal is missing either an opening or closing double-quote.

Example:

Listing 12.39: Code exemplar for UNMATCHED-DOUBLE-QUOTE

```
public class MissingSomething {
  private Double volume(String nameOfCurvedSolid, ↩
    ↪ double height, double base, double radius)↩
    ↪ {
    Double volume = null;
    switch (nameOfCurvedSolid.toUpperCase()) {
      case SPHERE":
        volume = 4 * Math.PI * Math.pow( radius, 2↩
           ↪ );
        break;
      case "CYLINDER":
        volume = 2 * Math.PI * Math.pow( radius, 2↩
           ↪ ) + 2 * Math.PI * radius * height;
        break;
      case "CONE":
        volume = Math.PI * radius * ( radius + ↩
           ↪ Math.sqrt(
             Math.pow( height, 2 ) + Math.pow( ↩
               ↪ radius, 2 ) ) );
        break;
    }
    return volume;
  }
}
```

Description Triggered when a string literal is missing either an opening or closing double-quote.

Repair: Look back from the indicated position for a string that does not begin or end with double quotes. Be aware of embedded double quotes and verify they are properly escaped.

Best Practices

- Break long string literals into multiple lines concatenated with a plus sign.
- Escape all double quotes that are embedded in the middle of a string by prepending the double quotes with a backslash. ()̈

12.2.39 UNREACHABLE-STATEMENT

Type: Structure

Source: Diagnostic

Description Triggered when the a statement can never be executed; usually because it is after a return, break, or in an never executed branch of an if-statement.

Example:

Listing 12.40: Code exemplar for UNREACHABLE-STATEMENT

```java
public class UnreachableStatement {
  public static int sum( int[ ] items ) {
    int sum = 0;
    for( int i = 0; i < items.length; i++ ) {
      if ( items[ i ] < 0 ) {
        break;
        System.out.println( "EXITING LOOP");
      }
      sum += items[ i ];
    }
    return sum;
    System.out.println("END OF METHOD: sum");
  }
}
```

Description Triggered when the a statement can never be executed; usually because it is after a return, break, or in an never executed branch of an if-statement.

12.2.40 VARIABLE-MAY-NOT-BE-INITIALIZED

Type: Structure

Source: Diagnostic

Description Most often triggered when a variable is declared without assigning a value.

Example:

Listing 12.41: Code exemplar for VARIABLE-MAY-NOT-BE-INITIALIZED

```java
public class VariableMayNotBeInitialized {
  public static void printVal( int num ) {
    int val;
    if ( num == 0 ) {
      val = 0;
    }
    System.out.println( val );
  }
}
```

Description Most often triggered when a variable is declared without assigning a value.

12.3 Behavioral Antipatterns

These patterns deal with the behavior or semantics of the code. Here, antipatterns can interfere with student solutions producing negative or unexpected results.

12.3.1 ARITHMETIC-EXCEPTION-DIV-BY-ZERO

Type: Behavior

Source: Exception

Description Trigger at runtime by integer division by zero.

Example:

Listing 12.42: Code exemplar for ARITHMETIC-EXCEPTION-DIV-BY-ZERO

```
public void divisionByZero( ) {
  divisionByZero( 5 );
}
private void divisionByZero( int divisor ) {
  int dividend = 5;
  double quotient = dividend / divisor;
  divisionByZero( divisor - 1 );
}
```

Description Trigger at runtime by integer division by zero.

12.3.2 ARRAY-INDEX-OUT-OF-BOUNDS-LOWER

Type: Behavior

Source: Exception

Description Trigger at runtime by referencing an element in an array using an index that is less than zero.

Example:

Listing 12.43: Code exemplar for ARRAY-INDEX-OUT-OF-BOUNDS-LOWER

```java
public void lowerArrayIndexOutOfBounds( ) {
  String [] stooges = {"Larry", "Curly", "Moe"};
  for( int index = stooges.length-1; ; index-- )↩
    ↪ {
    System.out.println( stooges[index] );
  }
}
```

Description Trigger at runtime by referencing an element in an array using an index that is less than zero.

12.3.3 ARRAY-INDEX-OUT-OF-BOUNDS-UPPER

Type: Behavior

Source: Exception

Description Trigger at runtime by referencing an element in an array using an index that is greater than or equal to the length of the array.

Example:

Listing 12.44: Code exemplar for ARRAY-INDEX-OUT-OF-BOUNDS-UPPER

```
public void loopArrayIndexOutOfBounds( ) {
  String [] stooges = {"Larry", "Curly", "Moe"};
  for( int index = 0; index <= stooges.length; ↩
     ↪ index++ ) {
    System.out.println( stooges[index] );
  }
}
```

Description Trigger at runtime by referencing an element in an array using an index that is greater than or equal to the length of the array.

12.3.4 ARRAY-STORE-EXCEPTION

Type: Behavior

Source: Exception

Description Trigger at when the rules for coercing elements of an array are violated.

Example:

Listing 12.45: Code exemplar for ARRAY-STORE-EXCEPTION

```java
public void arrayStoreException( ) {
  Object[] val = new Integer[ 4 ];
  val[ 0 ] = 5.8;
}
```

Description Trigger at when the rules for coercing elements of an array are violated.

12.3.5 CONSOLE-SCANNER-IN-LOOP

Type: Behavior

Source: Code

Description Triggered when a console Scanner is created within a loop.

Example:

Listing 12.46: Code exemplar for CONSOLE-SCANNER-IN-LOOP

```java
public String[ ] scannerInLoop( int numItems ) {
  String[ ] array = new String[ numItems ];
  System.out.printf( "Enter %d items:", numItems↩
    ↪ );
  for( int i = 0; i < numItems; i++ ) {
    Scanner scanner = new Scanner( System.in );
    array[i] = scanner.next( );
  }
  return array;
}
```

Description Triggered when a console Scanner is created within a loop.

12.3.6 EMPTY-LOOP

Type: Behavior

Source: Code

Description Sometimes students insert an empty loop in code for no apparent reason.

Example:

Listing 12.47: Code exemplar for EMPTY-LOOP

```
public int emptyLoop( int num ){
  for ( int i = 0; i < num; i++ ) {
    // Empty Loop
  }
  if( num >= 0 ){
    return num;
  }else{
    return -num;
  }
}
```

Description Sometimes students insert an empty loop in code for no apparent reason.

12.3.7 FILE-NOT-FOUND-EXCEPTION

Type: Behavior

Source: Exception

Description Trigger at when a file is not found. Most often when the student has hard-coded the filename.

Example:

Listing 12.48: Code exemplar for FILE-NOT-FOUND-EXCEPTION

```java
public void fileNotFoundException( String ↩
    ↪ filename ) throws FileNotFoundException {
  File file = new File( "myfile.data" );
  Scanner scanner = new Scanner( file );
}
```

Description Trigger at when a file is not found. Most often when the student has hard-coded the filename.

12.3.8 INPUT-MISMATCH-EXCEPTION-SCANNER

Type: Behavior

Source: Exception

Description Trigger at when a Scanner tries to access the next input as the wrong type.

Example:

Listing 12.49: Code exemplar for INPUT-MISMATCH-EXCEPTION-SCANNER

```java
public void inputMismatchException( ) {
  Scanner scanner = new Scanner( "ABC" );
  int i = scanner.nextInt( );
}
```

Description Trigger at when a Scanner tries to access the next input as the wrong type.

12.3.9 NO-SUCH-ELEMENT-ITERATOR

Type: Behavior

Source: Exception

Description Trigger at when an Iterator tries to access the next element and there is none.

Example:

Listing 12.50: Code exemplar for NO-SUCH-ELEMENT-ITERATOR

```java
public void listNoSuchElementException( ) {
  ArrayList<String> dwarves = new ArrayList<>();
  dwarves.add( "Sleepy" );
  Iterator<String> iterator = dwarves.iterator()↩
     ↪ ;
  while( true ) {
    System.out.println( iterator.next() );
  }
}
public void arraysNoSuchElementException( ) {
  List<String> dwarves = Arrays.asList( "Thorin", "Fili", "Kili", "Dwalin", "Balin", "Oin", "Gloin", "Dori", "Nori", "Ori", "Bifur", "Bofur", "Bombur");
  Iterator<String> iterator = dwarves.iterator();
  while( true ) {
    System.out.println( iterator.next() );
  }
}
public void spliteratorNoSuchElementException( ) {
  String[] dwarves = { "Thorin", "Fili", "Kili", "Dwalin", "Balin", "Oin", "Gloin", "Dori", "Nori", "Ori", "Bifur", "Bofur", "Bombur" };
  Iterator<String> iterator = Arrays.stream(dwarves).iterator();
  while( true ) {
    System.out.println( iterator.next() );
  }
}
public void primitiveIteratorNoSuchElementException( ) {
  int[] numbers = { 1, 2, 3 };
  PrimitiveIterator.OfInt iterator = Arrays.stream(numbers).iterator();
  while( true ) {
    System.out.println( iterator.next() );
  }
}
```

Description Trigger at when an Iterator tries to access the next element and there is none.

12.3.10 NO-SUCH-ELEMENT-SCANNER

Type: Behavior

Source: Exception

Description Trigger at when a a Scanner tries to read beyond the end of its input stream.

Example:

Listing 12.51: Code exemplar for NO-SUCH-ELEMENT-SCANNER

```
public void scannerNoSuchElementException( ) {
  Scanner scanner = new Scanner( "ABC" );
  String s1 = scanner.next( );
  String s2 = scanner.next( );
}
```

Description Trigger at when a a Scanner tries to read beyond the end of its input stream.

12.3.11 NULL-POINTER-EXCEPTION

Type: Behavior

Source: Exception

Description Trigger at runtime referencing a object with a null value.

Example:

Listing 12.52: Code exemplar for NULL-POINTER-EXCEPTION

```
private class Node<E> {
  public E value = null;
  public Node<E> next = null;
  public Node( E value ) {
    this.value = value;
  }
}
public <E> void add( Node<E> head, E value ) {
  if (value == null) {
    throw new IllegalArgumentException( "Can't ↩
       ↪ add null values to list." );
  }
  head.next = new Node<E>( value);
}
public void nullPointerException( ) {
  Node<Integer> head = null;
  this.<Integer>add( head, 42 );
}
```

Description Trigger at runtime referencing a object with a null value.

12.3.12 SCAN-STRING-FILENAME

Type: Behavior

Source: AST

Description Pattern occurs when student creates a new Scanner with the string filename instead of a File object.

Example:

Listing 12.53: Code exemplar for SCAN-STRING-FILENAME

```java
public void scanStringFilename( String filename ↩
    ↪ ) throws FileNotFoundException {
  Scanner scanner = new Scanner( filename );
  while( scanner.hasNext( ) ) {
    System.out.println( scanner.next( ) );
  }
}
```

Description Pattern occurs when student creates a new Scanner with the string filename instead of a File object.

12.3.13 STRING-INDEX-OUT-OF-BOUNDS

Type: Behavior

Source: Exception

Description Trigger at runtime by calling a String method with an index that is out of the range [0, length).

Example:

Listing 12.54: Code exemplar for STRING-INDEX-OUT-OF-BOUNDS

```java
public void upperStringIndexOutOfBounds( ) {
  String str = "It's not black magic; it's just ↩
      ↪ Java code!";
  char ch = str.charAt(50);
}
```

Description Trigger at runtime by calling a String method with an index that is out of the range [0, length).

12.3.14 SUBSTRING-INDEX-OUT-OF-BOUNDS

Type: Behavior

Source: Exception

Description Trigger at runtime by calling substring with an index that is outside the accepted ranges.

Example:

Listing 12.55: Code exemplar for SUBSTRING-INDEX-OUT-OF-BOUNDS

```
public void substringStringIndexOutOfBounds( ) {
  String stooge = "Shemp";
  for( int index = stooge.length(); ; index-- ) ↩
      ↪ {
    System.out.println( stooge.substring(0, ↩
        ↪ index) );
  }
}
```

Description Trigger at runtime by calling substring with an index that is outside the accepted ranges.

12.3.15 SUPPRESS-WARNINGS

Type: Behavior

Source: Code

Description Triggered when students use @SuppressWarnings to hide warnings.

Example:

Listing 12.56: Code exemplar for SUPPRESS-WARNINGS

```java
@SuppressWarnings( {"deprecation", "divzero", "empty", "rawtypes", "unchecked", "unused"} )
public <E> void suppressWarnings( ) {
  // unused
  int neverUsed = 0;
  // unchecked
  E[ ] array = (E[]) new Object[ 10 ];
  // rawtypes
  ArrayList list = new ArrayList( );
  // divzero
  int num = 5/0;
  // empty
  if ( true );
}
```

Description Triggered when students use @SuppressWarnings to hide warnings.

12.4 Style Antipatterns

These patterns deal with coding style. Style is an mastery concept. Students need to learn to code according to community standards as they journey from novice to expert. This ensures that they and others will be able to read, understand, and maintain their code.

12.4.1 ARITHMETIC-ASSIGNMENT

Type: Style

Source: Code

Description Triggered when failing to use concise += syntax

Example:

Listing 12.57: Code exemplar for ARITHMETIC-ASSIGNMENT

```
public double addingToVar( double total, double ↩
    ↪ amount ) {
  return total = total + amount;
}
```

Description Triggered when failing to use concise += syntax

12.4.2 BRACES-MISSING

Type: Style

Source: Code

Description Triggered when an if statement is followed by a statement instead of a code block;

Example:

Listing 12.58: Code exemplar for BRACES-MISSING

```java
public void missingBrackets( ) {
  if ( true )
    return;
  else return;
  for( int i = 0; i < 10; i++ )
    System.out.println( i );
}
```

Description Triggered when an if statement is followed by a statement instead of a code block;

12.4.3 CAPITALIZED-VARIABLE

Type: Style

Source: Code

Description Triggered when a variable begins with a capital letter.

Example:

Listing 12.59: Code exemplar for CAPITALIZED-VARIABLE

```
public void varStartsWithUpperCase( ) {
  int X = 5;
  int a = 2;
  System.out.println( X + a );
}
```

Description Triggered when a variable begins with a capital letter.

12.4.4 COMMA-WITHOUT-SPACE

Type: Style

Source: Code

Description Tiggered when there is not a space after a comma.

Example:

Listing 12.60: Code exemplar for COMMA-WITHOUT-SPACE

```
public void commadWithoutSpace( ) {
  List<String> list = java.util.Arrays.asList("↩
     ↪ Person","Woman","Man","Camera","TV");
}
```

Description Tiggered when there is not a space after a comma.

12.4.5 COMPARING-BOOLEANS

Type: Style

Source: Code

Description Triggered when a value is compared to a boolean.

Example:

Listing 12.61: Code exemplar for COMPARING-BOOLEANS

```java
public String compareBoolean( boolean flag ) {
  if ( flag != true ) {
    return "Flag is false";
  } else if ( flag == true ) {
    return "Flag is true";
  }
  // Code never reached but required by compiler
  return "Flag is uncertain!?!";
}
```

Description Triggered when a value is compared to a boolean.

12.4.6 COPYINTO

Type: Style

Source: Code

Description Triggered when a student uses the Vector method copyInto( ).

Example:

Listing 12.62: Code exemplar for COPYINTO

```java
public void copyIntoVsToArray( String [ ] array ↩
   ↪ ) {
  Vector<String> vector = new Vector<>( );
  vector.copyInto( array );
}
```

Description Triggered when a student uses the Vector method copyInto( ).

12.4.7 CRAMMED-OPERATORS

Type: Style

Source: Code

Description Sometimes operators not surrounded by spaces are okay in very short expressions or to indicate precedence. Don't over-use this.

Example:

Listing 12.63: Code exemplar for CRAMMED-OPERATORS

```java
public double crammedOperators( double celsius )↩
   ↪ {
  return (celsius-32.0)*5.0/9.0;
}
```

Description Sometimes operators not surrounded by spaces are okay in very short expressions or to indicate precedence. Don't over-use this.

12.4.8 CRAMMED-PARENS

Type: Style

Source: Code

Description Triggered when there isn't a space separating parens from their contents.

Example:

Listing 12.64: Code exemplar for CRAMMED-PARENS

```java
public void crammedParens ( ) {
  for(int i = 0; i < 10; i++){
    System.out.println(i);
  }
}
```

Description Triggered when there isn't a space separating parens from their contents.

12.4.9 FLOAT-USED

Type: Style

Source: Code

Description Triggered when the primitive data type float is used.

Example:

Listing 12.65: Code exemplar for FLOAT-USED

```
  public void shortOrFloatUsed( ) {
    short num = 0;
    float f = 0.0;
  }
```

Description Triggered when the primitive data type float is used.

12.4.10 IMPORTS-EVERYTHING-IN-PACKAGE

Type: Style

Source: Code

Description

Example:

Listing 12.66: Code exemplar for IMPORTS-EVERYTHING-IN-PACKAGE

```
import java.net.*;
```

Description

12.4.11 LOCAL-PATHNAME

Type: Style

Source: Code

Description Triggered when a DOS Path is used.

Example:

Listing 12.67: Code exemplar for LOCAL-PATHNAME

```
public void dosFilename( ) {
  File file = new File( "C:\\users\\home" );
}
```

Description Triggered when a DOS Path is used.

12.4.12 LOOP-FOR-VAR-NOT-LOCAL

Type: Style

Source: Code

Description Triggered when a for-loop variable is declared outside the scope of the loop.

Example:

Listing 12.68: Code exemplar for LOOP-FOR-VAR-NOT-LOCAL

```
    int i;
    for( i = 0; i < 10; i++ ) {

    }
```

Description Triggered when a for-loop variable is declared outside the scope of the loop.

12.4.13 LOWERCASE-CLASS-NAME

Type: Style

Source: Code

Description Triggered when a class name begins with a lowercase letter.

Example:

Listing 12.69: Code exemplar for LOWERCASE-CLASS-NAME

```
public class lowercaseClassname {

}
```

Description Triggered when a class name begins with a lowercase letter.

12.4.14 METHOD-STARTS-WITH-UPPERCASE

Type: Style

Source: Code

Description Triggered when a method name begins with an uppercase letter.

Example:

Listing 12.70: Code exemplar for METHOD-STARTS-WITH-UPPERCASE

```
public int UppercaseMethodName ( ) {

}
```

Description Triggered when a method name begins with an uppercase letter.

12.4.15 MISSING-COMMENT

Type: Style

Source: Diagnostic

Description Triggered when the compiler is expecting a JavaDoc comment before the indicated code structure.

Example:

Listing 12.71: Code exemplar for MISSING-COMMENT

```java
public class MissingComment {
  public static void main( String [ ] args ) {
    System.out.println( "Hello World" );
  }
}
```

Description Triggered when the compiler is expecting a JavaDoc comment before the indicated code structure.

12.4.16 MULTIPLE-VAR-ON-LINE

Type: Style

Source: Code

Description Triggered when multiple variables are declared in a single statement.

Example:

Listing 12.72: Code exemplar for MULTIPLE-VAR-ON-LINE

```java
  public void multipleVarDeclaredSingleLine ( ) {
    int x = 5, y = 6, z = 50;
    System.out.println( x + y + z );
  }
```

Description Triggered when multiple variables are declared in a single statement.

12.4.17 NAMING-UNDERBARS

Type: Style

Source: Code

Description Triggered when a variable name contains underbars.

Example:

Listing 12.73: Code exemplar for NAMING-UNDERBARS

```java
public void variableNameWithUnderbars( ){
  int howl_jenkins_pendragon = 0;
}
```

Description Triggered when a variable name contains underbars.

12.4.18 NO-SPACE-IN-FOR-LOOP

Type: Style

Source: Code

Description

Example:

Listing 12.74: Code exemplar for NO-SPACE-IN-FOR-LOOP

```
public int noSpaceInForLoop( int num ) {
  int sum = 0
  for(int i=1;i<num;i++){
    sum+=i;
  }
  return sum;
}
```

Description

12.4.19 OPERATORS-++

Type: Style

Source: Code

Description Triggered when pattern x = x + 1.

Example:

Listing 12.75: Code exemplar for OPERATORS-++

```
public void addingOneToVar( ) {
  int count = 0;
  count = count + 1;
}
```

Description Triggered when pattern x = x + 1.

12.4.20 PACKAGE-NAME-TOO-GENERAL

Type: Style

Source: Code

Description E.g., com as a package name

Example:

Listing 12.76: Code exemplar for PACKAGE-NAME-TOO-GENERAL

```java
package mathlib;

public class PackageTooGeneral {
  public static int absoluteValue( long number ) {
    return number >= 0 ? number : -number;
  }
}
```

Description E.g., com as a package name

12.4.21 PACKAGE-NEEDLESS-QUALIFIED-NAME

Type: Style

Source: Code

Description Triggered when a package is imported and specified in a method call.

Example:

Listing 12.77: Code exemplar for PACKAGE-NEEDLESS-QUALIFIED-NAME

```
import java.util.Arrays;

public class PackageQualifiedName {
  public void mumble( ) {
    List<String> list = java.util.Arrays.asList( "↩
       ↪ Person", "Woman", "Man", "Camera", "TV" ↩
       ↪ );
    System.out.println( list );
  }
}
```

Description Triggered when a package is imported and specified in a method call.

12.4.22 PACKAGE-QUALIFIED-NAMES

Type: Style

Source: Code

Description Triggered when a fully qualified method name is used in a call.

Example:

Listing 12.78: Code exemplar for PACKAGE-QUALIFIED-NAMES

```
  public void packageQualifiedNames( ) {
    List<String> list = java.util.Arrays.asList( "↩
      ↪ Person", "Woman", "Man", "Camera", "TV" ↩
      ↪ );
    System.out.println( list );
  }
```

Description Triggered when a fully qualified method name is used in a call.

12.4.23 PUBLIC-INSTANCE-VARIABLES

Type: Style

Source: Code

Description Triggered when instance variables are public.

Example:

Listing 12.79: Code exemplar for PUBLIC-INSTANCE-VARIABLES

```
public class SourceCodeExample {
  public String name = "Georges";
  public int age = 27;
  public int grade = 12;

  // call to default super constructor
  public SourceCodeExample( ) {
    super( );
  }
}
```

Description Triggered when instance variables are public.

12.4.24 RETURN-WITH-PARENS

Type: Style

Source: Code

Description Triggered when return value is bracketed within parens

Example:

Listing 12.80: Code exemplar for RETURN-WITH-PARENS

```
public int returnWithParens( int a, int b ) {
  return ( a + b );
}
```

Description Triggered when return value is bracketed within parens

12.4.25 SHORT-USED

Type: Style

Source: Code

Description Triggered when the the primitive data type short is used.

Example:

Listing 12.81: Code exemplar for SHORT-USED

```
public void shortOrFloatUsed( ) {
  short num = 0;
  float f = 0.0;
}
```

Description Triggered when the the primitive data type short is used.

12.4.26 SPACE-BEFORE-COMMA

Type: Style

Source: Code

Description

Example:

Listing 12.82: Code exemplar for SPACE-BEFORE-COMMA

```
public void spaceBeforeComma( ) {
  List<String> list = java.util.Arrays.asList("↩
     ↪ Person" ,"Woman" ,"Man" ,"Camera" ,"TV")↩
     ↪ ;
}
```

Description

12.4.27 SPACE-BEFORE-CURLY-BRACE

Type: Style

Source: Code

Description Triggered when there is no space before an opening curly brace.

Example:

Listing 12.83: Code exemplar for SPACE-BEFORE-CURLY-BRACE

```
public void missingSpaceBeforeCurlyBrace( ){
  for( int i = 1; i < 12; i++ ){
    System.out.println( "Hello" );
  }
}
```

Description Triggered when there is no space before an opening curly brace.

12.4.28 STRING-EQUALS-HINT

Type: Style

Source: AST

Description Triggered when comparing two strings using using ==.

Example:

Listing 12.84: Code exemplar for STRING-EQUALS-HINT

```
public boolean stringEqualsHint( String s1, ↩
    ↪ String s2 ) {
  return s1 == s2;
}
```

Description Triggered when comparing two strings using using ==.

12.4.29 STRING-EQUALS-LITERAL

Type: Style

Source: Code

Description Triggered when comparing a string literal using ==.

Example:

Listing 12.85: Code exemplar for STRING-EQUALS-LITERAL

```
public void stringEquals( String name ) {
  if ( name == "Scooby Doo" ) {
    System.out.println( "Time for a Scooby Snack↩
       ↪ !" );
  }
}
```

Description Triggered when comparing a string literal using ==.

12.4.30 THIS-METHOD

Type: Style

Source: Code

Description Triggered when the keyword "this" is used to call a method.

Example:

Listing 12.86: Code exemplar for THIS-METHOD

```
public class ThisMethodCall {
  public int a ( ) {
    this.b( );
  }
  public int b ( ) {

  }
}
```

Description Triggered when the keyword "this" is used to call a method.

12.4.31 VAR-STARTS-WITH-UPPERCASE

Type: Style

Source: Code

Description Triggered when a variable starts with an uppercase letter.

Example:

Listing 12.87: Code exemplar for VAR-STARTS-WITH-UPPERCASE

```java
public void varStartsWithUpperCase( ) {
  int X = 5;
  int a = 2;
  System.out.println( X + a );
}
```

Description Triggered when a variable starts with an uppercase letter.

12.4.32 VECTOR-FOR-FIXED-ARRAY

Type: Style

Source: Code

Description Triggered when a Vector is instantiated at a specific size.

Example:

Listing 12.88: Code exemplar for VECTOR-FOR-FIXED-ARRAY

```java
public static Vector<String> vectorForFixedArray↩
  ↪ ( ) {
  Vector<String> planets = new Vector( 8 );
  int index = 0;
  planets.add( index++, "Mercury" );
  planets.add( index++, "Venus" );
  planets.add( index++, "Earth" );
  planets.add( index++, "Mars" );
  planets.add( index++, "Jupiter" );
  planets.add( index++, "Saturn" );
  planets.add( index++, "Uranus" );
```

```
        planets.add( index++, "Neptune" );
        return planets;
    }
```

Description Triggered when a Vector is instantiated at a specific size.

12.5 Test-Driven Development Antipatterns

These patterns deal with testing code. Learning how and what to test is among the most challenging tasks for novice programmers. Often the solutions and algorithms seems so obvious to them that any problems will be attributed to the instructor's inability to properly execute the code!

12.6 Design Development Antipatterns

Ultimately, we want to teach students good design pactices. This can seem like a herculean task in the introductory computer science sequence where students are still learning the syntax of a language. However, there are many opportunities to identify design antipatterns and remediate them early on.

Part V

Final Thoughts

Chapter 13

Conclusion

13.1 Conclusion

There are two mirror research paths that started in the 1950s when we started experimenting with using computers in the classroom as part of teaching programming. In 1960 instructors were using computers to support teaching and Hollingsworth published the first paper describing an Autograder [29]. The other side of this research is supporting students and by the early 1980s researchers had started to investigate how the computer could be used to provide critical feedback to students that would help them learn to program [31].

WebTA is a code critiquer developed to identify novice antipatterns in student code. WebTA functions both as an Autograder and a Code Critiquer presenting opportunities for research in both areas. As a research platform, WebTA has proven fruitful; enabling the compilation of a library of novice antipatterns and the aggregation of a large corpus of student submission data, both of which are also major contributions of this work.

Novice antipatterns typically do not appear in professional or academic software engineering texts because they do not represent the kinds of misunderstandings and mistakes that seasoned developers would exhibit. Using the antipattern library, 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 expect the large corpus of student submission data to be a rich source of information as we continue research in this area. We plan to use it to determine the frequency of antipatterns and determine if student antipattern production declines with use of WebTA. Continued and broader use of these resources will provide us with the data needed to confirm the effectiveness of our automated critique method.

I feel like I have more questions now than when I set out on this journey. I am looking forward to exploring the future research laid out in Chapter 11.

References

[1] Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102.

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

[3] Ali, N. M., Hosking, J., and Grundy, J. (2010). A taxonomy of computer-supported critics. In *2010 International Symposium on Information Technology*, volume 3, pages 1152–1157. IEEE.

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

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

[6] Altadmri, A. and Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA. ACM.

[7] AntiPattern Catalog (2012).

[8] Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.-M., Pearce, J. L., and Prather, J. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR 19, page 177210, New York, NY, USA. Association for Computing Machinery.

[9] Brown, C., Pastel, R., Siever, B., and Earnest, J. (2012a). Jug: A junit generation, time complexity analysis and reporting tool to streamline grading. *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education - ITiCSE '12*.

[10] Brown, C., Pastel, R., Siever, B., and Earnest, J. (2012b). 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, New York, NY, USA.

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

[12] Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998a). *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc.

[13] Brown, W. J., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. (1998b). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons.

[14] Collins, A. (1991). Cognitive apprenticeship and instructional technology. *Educational values and cognitive instruction: Implications for reform*, 1991:121–138.

[15] Collins, A. (2006). *Cognitive apprenticeship*. na.

[16] Collins, A., Brown, J. S., and Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. *American educator*, 15(3):6–11.

[17] Collins, A., Brown, J. S., and Newman, S. E. (1988). Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. *Thinking: The Journal of Philosophy for Children*, 8(1):2–10.

[18] DeNero, J., Sridhara, S., Pérez-Quiñones, M., Nayak, A., and Leong, B. (2017). Beyond autograding: Advances in student feedback platforms. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 651–652, NY, NY. ACM, ACM.

[19] Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 208–212.

[20] Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4.

[21] Edwards, S. H. and Perez-Quinones, M. A. (2008a). Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 328–328.

[22] Edwards, S. H. and Perez-Quinones, M. A. (2008b). Web-cat: Automatically grading programming assignments. *Proceedings of the 13th annual conference on Innovation and technology in computer science education - ITiCSE '08*.

[23] Estey, A. and Coady, Y. (2016). Can interaction patterns with supplemental study tools predict outcomes in cs1? In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 236–241, New York, NY, USA. ACM.

[24] Ettles, A., Luxton-Reilly, A., and Denny, P. (2018). Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 83–89.

[25] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2018). *How to design programs: an introduction to programming and computing.* MIT Press.

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

[27] Fowler, M., Highsmith, J., et al. (2001). The agile manifesto. *Software Development*, 9(8):28–35.

[28] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley.* Addison-Wesley.

[29] Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529.

[30] Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10.*

[31] Joni, S.-N. A. and Soloway, E. (1986). But my program runs! discourse rules for novice programmers. *Journal of Educational Computing Research*, 2(1):95–125.

[32] Joy, M. and Griffiths, N. (2004). Online submission of coursework—A technological perspective. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, pages 430–434.

[33] Joy, M., Griffiths, N., and Boyatt, R. (2005). The boss online submission and assessment system. *Journal on Educational Resources in Computing*, 5(3):2–es.

[34] Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. (2009). A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE.

[35] Koenig, A. (1995). Patterns and antipatterns. *Journal of Object-oriented Programming*, 8:46–48.

[36] Kolb, A. Y. (2005). The kolb learning style inventory-version 3.1 2005 technical specifications. *Boston, MA: Hay Resource Direct*, 200(72).

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

[38] McDowell, C., Werner, L., Bullock, H. E., and Fernald, J. (2003). The impact of pair programming on student performance, perception and persistence. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 602–607. IEEE.

[39] Munson, J. P. and Schilling, E. A. (2016). Analyzing novice programmers' response to compiler error messages. *J. Comput. Sci. Coll.*, 31(3):53–61.

[40] Nagappan, N., Williams, L., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., and Balik, S. (2003). Improving the cs1 experience with pair programming. *ACM SIGCSE Bulletin*, 35(1):359–362.

[41] Nienaltowski, M.-H., Pedroni, M., and Meyer, B. (2008). Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 168–172.

[42] Oman, P. W. and Cook, C. R. (1990). A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 244–250.

[43] Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA. ACM.

[44] Prather, J., Pettit, R., McMurry, K. H., Peters, A., Homer, J., Simone, N., and Cohen, M. (2017). On novices' interaction with compiler error messages: A human factors approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 74–82.

[45] Qiu, L. and Riesbeck, C. (2004a). Making critiquing practical: incremental development of educational critiquing systems. In Vanderdonckt, J., Nunes, N. J., and Rich, C., editors, *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI 2004, pages 304–306. ACM.

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

[47] Qiu, L. and Riesbeck, C. K. (2003). Facilitating critiquing in education: The design and implementation of the java critiquer. In *Proceedings of the International Conference on Computers in Education*.

[48] Qiu, L. and Riesbeck, C. K. (2004b). Making critiquing practical: Incremental development of educational critiquing systems. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 304–306. ACM.

[49] Qiu, L. and Riesbeck, C. K. (2008b). Human-in-the-loop: a feedback-driven model for authoring knowledge-based interactive learning environments. *Journal of Educational Computing Research*, 38(4):469–509.

[50] Rodrigo, M. M. T., Andallaza, T. C. S., Castro, F. E. V. G., Armenta, M. L. V., Dy, T. T., and Jadud, M. C. (2013). An analysis of java programming behaviors, affect, perceptions, and syntax errors among low-achieving, average, and high-achieving novice programmers. *Journal of Educational Computing Research*, 49(3):293–325.

[51] Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON, 1970*, pages 1–9.

[52] Schank, R. C., Berman, T. R., and Macpherson, K. A. (1999). Learning by doing. *Instructional-design theories and models: A new paradigm of instructional theory*, 2(2):161–181.

[53] Sousa, B. L., Souza, P. P., Fernandes, E. M., Ferreira, K. A., and Bigonha, M. A. (2017). Findsmells: Flexible composition of bad smell detection strategies. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC).*

[54] Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., and Padua-Perez, N. (2006). Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education - ITICSE '06.*

[55] Spacco, J. W. (2006). *Marmoset: A Programming Project Assignment Framework to Improve the Feedback Cycle for Students, Faculty and Researchers*. PhD thesis, University of Maryland at College Park, College Park, MD, USA. AAI3241457.

[56] Trætteberg, H. and Aalberg, T. (2006). Jexercise. *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange - eclipse '06.*

[57] Traver, V. J. (2010). On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010.

[58] Vihavainen, A. and Luukkainen, M. (2013). Results from a three-year transition to the extreme apprenticeship method. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 336–340. IEEE.

[59] Vihavainen, A., Paksula, M., and Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 93–98. ACM.

[60] Vihavainen, A., Vikberg, T., Luukkainen, M., and Pärtel, M. (2013). Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, New York, NY, USA. ACM.

[61] Wiese, E. S., Rafferty, A. N., Kopta, D. M., and Anderson, J. M. (2019). Replicating novices' struggles with coding style. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 13–18. IEEE.