University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

2015

Automated Style Feedback for Advanced Beginner Java Programmers

Hannah Blau
University of Massachusetts - Amherst

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Software Engineering Commons

Recommended Citation

Blau, Hannah, "Automated Style Feedback for Advanced Beginner Java Programmers" (2015). *Doctoral Dissertations*. 543.
https://scholarworks.umass.edu/dissertations_2/543

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

AUTOMATED STYLE FEEDBACK FOR ADVANCED BEGINNER JAVA PROGRAMMERS

A Dissertation Presented

by

HANNAH BLAU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2015

College of Information & Computer Sciences

© Copyright by Hannah Blau 2015
All Rights Reserved

AUTOMATED STYLE FEEDBACK FOR ADVANCED BEGINNER JAVA PROGRAMMERS

A Dissertation Presented

by

HANNAH BLAU

Approved as to style and content by:

W. Richards Adrion, Co-chair

Robert Moll, Co-chair

Barbara Lerner, Member

J. Eliot B. Moss, Member

James Allan, Chair
College of Information & Computer Sciences

ACKNOWLEDGMENTS

This work would not have been possible without the support and guidance of my Ph.D. advisors W. R. Adrion and R. Moll and my committee members B. Lerner and J. E. B. Moss. Y. Smaragdakis also offered valuable insights at an early stage of the project. Prof. Moss gave me indispensable technical guidance and made substantial contributions to the plug-in implementation. I thank the professors who graciously allowed me to collect data in their classrooms: J. Allan, D. A. M. Barrington, M. Corner, W. Lehnert, G. Miklau, and T. Richards. Many fruitful discussions with R. Fall helped me clarify my thoughts. I am grateful to S. Kolovson for her unflagging effort on data analysis.

ABSTRACT

AUTOMATED STYLE FEEDBACK FOR ADVANCED BEGINNER JAVA PROGRAMMERS

SEPTEMBER 2015

HANNAH BLAU

B.A., YALE UNIVERSITY

M.S.E., UNIVERSITY OF PENNSYLVANIA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor W. Richards Adrion and Professor Robert Moll

FrenchPress is an Eclipse plug-in that partially automates the task of giving students feedback on their Java programs. It is designed not for novices but for students taking their second or third Java course: students who know enough Java to write a working program but lack the judgment to recognize bad code when they see it. FrenchPress does not diagnose compile-time or run-time errors, or logical errors that produce incorrect output. It targets silent flaws, flaws the student is unable to identify for himself because nothing in the programming environment alerts him.

FrenchPress diagnoses flaws characteristic of programmers who have not yet assimilated the object-oriented idiom. Such shortcomings include misuse of the `public` modifier, fields that should have been local variables, and instance variables that should have been class constants. Other rules address the all too common misunderstanding of the boolean data type. FrenchPress delivers explanatory messages

in a vocabulary appropriate for advanced beginners. FrenchPress does not fix the problems it detects; the student must decide whether to change the program.

The plug-in has been tested by undergraduates in the UMass data structures and algorithms course, the target audience for FrenchPress diagnostics. A pilot study took place during winter break 2013–2014 and a preliminary classroom trial in Spring 2014. This dissertation reports results from the final classroom trial covering four programming assignments in Fall 2014. Among students whose code triggered one or more of the diagnostic rules, the percentage who modified their program in response to FrenchPress feedback varied from a high of 59% on the first project to a low of 23% on the second and fourth projects. User satisfaction surveys indicate that among students who said FrenchPress gave them suggestions for improvement, the percentage who found the feedback helpful bounced from around 55% on the first and third assignments to 32–40% on the second and fourth assignments. The lower acceptance on the second and fourth projects corresponds to a higher incidence of false positives and other confusing feedback messages. Nevertheless, the percentage of survey respondents who said they were satisfied with FrenchPress performance ranged from 56% to 66% on all four assignments.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS .... **iv**

ABSTRACT .... **v**

LIST OF TABLES .... **x**

LIST OF FIGURES .... **xi**

CHAPTER

1. MOTIVATION AND OBJECTIVES **1**

1.1 Need for automated feedback 1
1.2 FrenchPress in action 3
1.3 Research hypotheses 6
1.4 Objectives 6

1.4.1 Effective feedback 7
1.4.2 Easy for the student 8
1.4.3 Easy for the professor 9

1.5 Design tradeoffs in a diagnostic tool 9
1.6 FrenchPress is not a grading instrument 11
1.7 Evolution of CMPSCI 187 projects 12

2. RELATED WORK **16**

2.1 Automated assessment systems in academia 17
2.2 Static analysis tools for professionals 22
2.3 Code smells 27

3. DIAGNOSTICS **30**

3.1 Four categories of flaws 31

3.1.1 Misuse of fields . . . 31
3.1.2 Misuse of the public modifier . . . 32
3.1.3 Misunderstanding booleans . . . 36
3.1.4 For loops . . . 39

3.2 Minimize duplication with prior work . . . 40
3.3 Implementation . . . 41

4. FRENCHPRESS PLUG-IN . . . 46

4.1 Implementing an Eclipse plug-in . . . 46
4.2 Running the FrenchPress plug-in . . . 46
4.3 Running FrenchPress on the entire `src` folder . . . 48

5. CLASSROOM TRIAL OF FRENCHPRESS PLUG-IN . . . 51

5.1 Plan for classroom trial . . . 52
5.2 User satisfaction survey . . . 53

5.2.1 Keep it short . . . 54
5.2.2 Survey questions . . . 55
5.2.3 Limitations of survey data . . . 56

5.3 Culling data from the FrenchPress archive folder . . . 58
5.4 Off-target feedback . . . 59

5.4.1 Implementation errors . . . 59
5.4.2 Software out of date . . . 60
5.4.3 Student ran plug-in too early . . . 61
5.4.4 Student ran plug-in on a JUnit test class . . . 61
5.4.5 Unused field or method . . . 61
5.4.6 Getter and setter methods . . . 62

5.5 Lessons learned . . . 62

6. RESULTS OF CLASSROOM TRIAL . . . 64

6.1 Data collection . . . 65
6.2 FrenchPress program archive . . . 66

6.2.1 Recap of diagnostic rules . . . 69
6.2.2 Short-term indicator of student learning . . . 69
6.2.3 Longer-term indicator of student learning . . . 70

6.3 User satisfaction survey responses . . . 76
6.4 Selected student comments . . . 80

7. FUTURE WORK **83**

7.1 Student experience of FrenchPress 83
7.2 New features for professors 84
7.3 Improved diagnostics 85
7.4 Extended classroom trial 86
7.5 Conclusion 87

APPENDICES

A. CMPSCI 187 ASSIGNMENTS **89**
B. STUDENT INTERACTION WITH FRENCHPRESS **98**
C. MOTIVATION FOR UNIMPLEMENTED RULES **119**
D. CLASSROOM TRIAL INFORMED CONSENT FORM **122**
E. FRENCHPRESS USER SATISFACTION SURVEY **128**

BIBLIOGRAPHY **131**

LIST OF TABLES

Table **Page**

2.1 Automated assessment systems for introductory programming courses . . . 19

2.2 JDeodorant code smells and the corresponding refactorings . . . 28

5.1 FrenchPress usage by version . . . 60

6.1 CMPSCI 187 assignment topics . . . 66

6.2 Fluctuation in classroom trial participation . . . 66

6.3 Substantive feedback varies by project . . . 69

6.4 Recap of FrenchPress diagnostics . . . 69

6.5 Off-target feedback varies by project . . . 70

LIST OF FIGURES

Figure **Page**

1.1 FrenchPress feedback for a single `.java` file 3

1.2 Student's BirthdayDriver.java 4

1.3 FrenchPress feedback for BirthdayDriver.java 5

2.1 Student's HanoiSolverImp.java, part 1 23

2.2 Student's HanoiSolverImp.java, part 2 24

2.3 PMD feedback for HanoiSolverImp.java 24

2.4 FrenchPress feedback for HanoiSolverImp.java 25

2.5 Revised HanoiSolverImp.java 26

2.6 PMD feedback for revised HanoiSolverImp.java 27

2.7 FrenchPress feedback for revised HanoiSolverImp.java 27

3.1 Student's ListImp.java 33

3.2 FrenchPress feedback for ListImp.java 34

3.3 Revised ListImp.java 35

3.4 Student's PostOrderIterator.java, part 1 37

3.5 Student's PostOrderIterator.java, part 2 38

3.6 FrenchPress feedback for PostOrderIterator.java 39

3.7 Student's BirthdayParadox.java, part 1 41

3.8 Student's BirthdayParadox.java, part 2 42

3.9 FrenchPress feedback for BirthdayParadox.java 43

4.1 FrenchPress popup dialog for entire `src` folder 49

6.1 Rates of substantive feedback and program modification 67

6.2 Rules 1 and 2 feedback trends 73

6.3 Rules 3 and 4 feedback trends 74

6.4 Rules 5 and 6 feedback trends 75

6.5 Is feedback confusing or easy to understand? 77

6.6 Is feedback helpful or unhelpful? 78

6.7 Overall satisfaction with FrenchPress 79

CHAPTER 1
MOTIVATION AND OBJECTIVES

1.1 Need for automated feedback

Anyone who has taught a college programming course at the freshman or sophomore level knows how difficult and time-consuming it can be to address issues of program style in student submissions. Even if the professor discusses numerous examples of well-written code in class, many students ignore or mangle these templates when they write their own programs. Often the class is so large that the teaching assistant/grader can spend only a few minutes on each program, checking to see whether it produces the expected output for selected test inputs. Or the course staff adopt an automated test harness to grade student submissions. This naturally leads students to conclude that a good program is one that has the desired input/output behavior, and it matters not how they achieve that behavior. A student could get a perfect grade on all his assignments and still be writing poor code.

When the teaching staff does take the time to inspect student programs, issues of subjectivity arise. Judgments of code quality are hard to pin down, all the more so if multiple instructors are employed for different sections of the same course or over time. The potential for inconsistency makes the instructor reluctant to assign much weight to program style in calculating grades. If the student loses only a few points for a poorly written program, he will have little motivation to follow the corrections he receives. He might even dismiss these comments as a reflection of the instructor's idiosyncrasies. The student could reach upper level courses before he gets a professor whose grading policy enforces good programming practices. By this point the student

might already have developed bad programming habits. These bad habits carry over into advanced classes where they are a hindrance for the student and a headache for the professor.

To address this problem I developed FrenchPress, an Eclipse plug-in that partially automates the task of giving students feedback concerning their Java programs. I chose Java because it has been widely adopted (including at UMass Amherst) for undergraduate curricula in object-oriented programming. Students get guidance to make improvements without depending on the instructor or the teaching assistant to review their code. My target population is not raw beginners but students taking their second or third undergraduate Java course: students at the level of an introduction to data structures and algorithms, who know enough Java to write a working program but lack the judgment to recognize bad code when they see it. While it is never too early for a professor to explain in class the elements of good programming style, I do not think that students in their first semester of Java would be receptive to automated stylistic feedback. Getting a program to compile and run to completion is already a high hurdle for beginning students to surmount. Novice learners struggle mightily to identify and correct logical errors. Showering them with feedback on stylistic points they do not yet have the experience to understand is more likely to create information overload than to inspire better programming.

The goal of FrenchPress is not to diagnose compile-time or run-time errors, or logical flaws that produce incorrect output. A student who is paying attention can recognize these issues without a special plug-in. I am after the silent flaws—flaws the student is unable to identify for himself because he gets no feedback from the programming environment to alert him that a problem exists. Such shortcomings include misuse of the `public` modifier, fields that should have been local variables, and instance variables that should have been class constants. These flaws are characteristic of programmers who have not yet assimilated the object-oriented idiom.

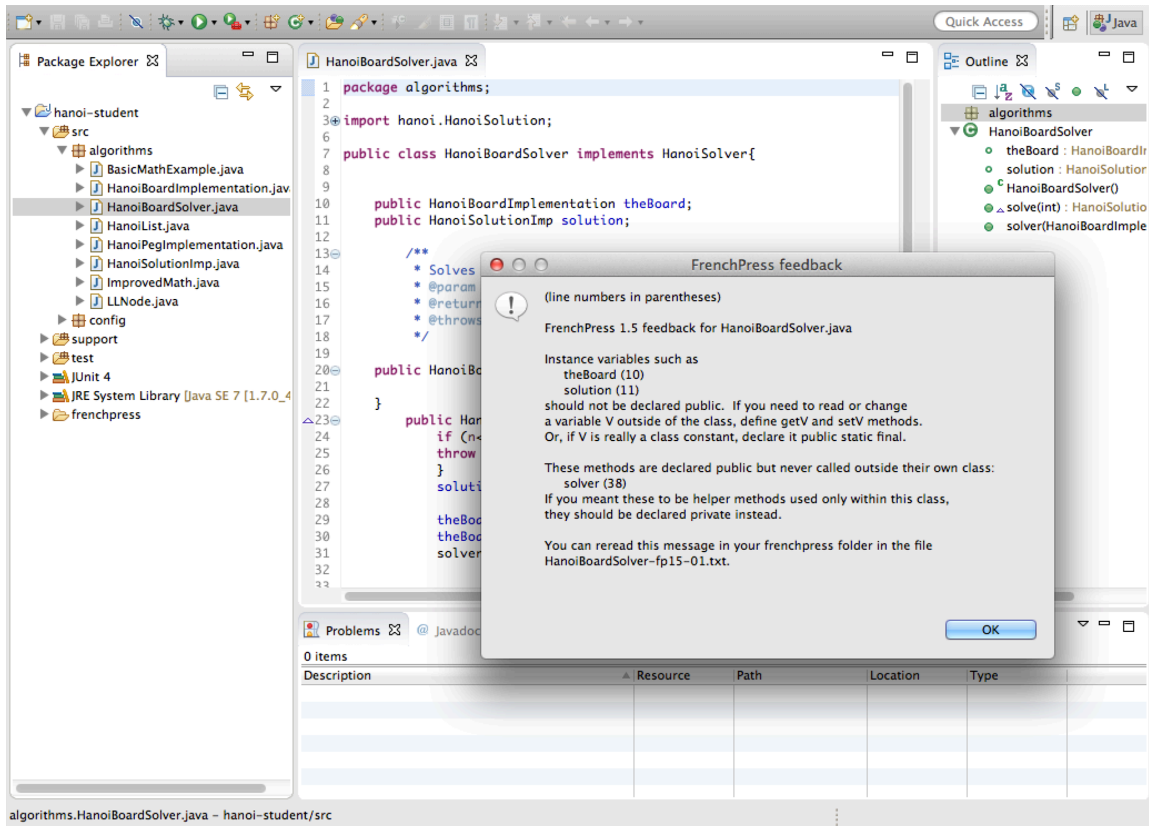

Figure 1.1. FrenchPress feedback for a single `.java` file

Other diagnostic rules target the all too common misunderstanding of the boolean data type.

1.2 FrenchPress in action

Figure 1.1 shows the FrenchPress plug-in in action within the Eclipse IDE. The editor window displays one of the project's class definitions. FrenchPress presents feedback for that file in a dialog box. Figure 1.2 shows a class definition written by a student in the data structures and algorithms course in Spring 2008. This `.java` file triggers feedback from four of FrenchPress's seven diagnostic rules (Figure 1.3). Chapter 3 discusses FrenchPress's diagnostics in more detail.

```
1    public class BirthdayDriver{
2      private final int trialRuns = 10000;
3      int counter = 0;
4
5      public static void main(String[] args){
6         BirthdayDriver driver = new BirthdayDriver(10, 30);
7       }
8       public BirthdayDriver(int min, int max){
9         for(;min<max; min++){
10         printResult(runTrials(min), min);
11        }
12      }
13
14      public double runTrials(int roomSize){
15        counter = 0;
16        for(int a = 0; a < trialRuns; a++){
17          BDayTest bDay = new BDayTest(roomSize);
18          if(bDay.match){
19            counter++;
20          }
21          }
22        return (double)counter/trialRuns;
23      }
24
25      public void printResult(double per, int size){
26         System.out.println("people: " + size + " -> " + per);
27      }
28
29      }
```

Figure 1.2. Student's BirthdayDriver.java

```
FrenchPress 1.5 feedback for BirthdayDriver.java

Variables such as
     counter (3)
are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.

Instance variables such as
     trialRuns (2)
could be declared static final (class constants) because they are
initialized to a constant value and never changed later.

These methods are declared public but never called outside their
own class:
     runTrials (14)
     printResult (25)
If you meant these to be helper methods used only within this class,
they should be declared private instead.

Method parameters such as
     min (9)
should not be used as for loop control variables.
It is preferable to use a separate variable as in, for (int i = ...).
```

Figure 1.3. FrenchPress feedback for BirthdayDriver.java

1.3 Research hypotheses

The research hypotheses underlying this work are:

H1 FrenchPress can reliably identify stylistic issues in Java code and communicate those problems to students.

H2 Students who use FrenchPress consistently for all the programming assignments in the course will by the end of the semester be making fewer errors of the types diagnosed by the tool than they did at the beginning. They will learn to avoid the flaws about which the tool has warned them.

It is tempting to hypothesize that students who use the diagnostic tool regularly will achieve better grades on their homework. However, as noted in Chapter 1.1, grading criteria in large undergraduate courses do not usually reflect the qualities of program style FrenchPress is designed to promote. FrenchPress nudges students toward better programming practices, but the nudge is gentle and might not pay off until a semester or more after the end of the student's exposure to the automated feedback. Students who chose to participate in the classroom trial of FrenchPress (Chapter 5) were rewarded with a small amount of extra credit toward their final grade. Those who opted out of the trial could earn the same extra credit points by completing a small research assignment about Java programming style recommendations. Introducing the plug-in into the data structures and algorithms course did not create a grade differential between FrenchPress users and non-users.

1.4 Objectives

The main objectives that guided the design of the diagnostic tool are:

1. Give the student effective and useful feedback without increasing TA or professor workload.

2. Make it easy for the student to run the diagnostic tool, so he can iteratively improve his program before the final submission.

3. Liberate the course author from the burden of writing customized design checks for each programming exercise.

1.4.1 Effective feedback

Many of the programming best practices articulated in books such as *Effective Java* [6] evolved in the context of projects with multiple team members producing software destined to be maintained and enhanced for several years. This perspective is essentially meaningless to a student who works no longer than two weeks on any assignment and who knows neither he nor anyone else will ever look at his code again after he gets his grade. The feedback FrenchPress gives this student must be relevant for his situation, not some hypothetical situation of large-scale software development. My goal is to encourage coding discipline that will serve students well throughout their career, while offering feedback they find meaningful at their current level of knowledge.

Giving effective feedback means formulating messages in language the student can understand, even if that entails glossing over the subtleties. Giving effective feedback also means that the incidence of false positives must be kept to a minimum. A false positive occurs when FrenchPress delivers feedback that is inappropriate for the student's program: a diagnostic rule triggers when it should not have. Students at the advanced beginner level in Java would not be able to distinguish between a true flaw and a false report. They might respond to a spurious feedback message by modifying their program in a way that makes it worse, not better. This would undermine their trust in the tool's suggestions and they would stop using it. A false negative occurs when FrenchPress skips over a student mistake that could have been corrected: a diagnostic rule fails to trigger when it should have. False negatives are less of a

problem than false positives for a user population of inexperienced programmers. As these students were not receiving much stylistic feedback from their instructors, if FrenchPress misses an opportunity to be helpful the student is no worse off than he was before he installed the software.

1.4.2 Easy for the student

Even when the professor or TA takes the time to write comments on submitted assignments, that feedback might come too late to be of interest to the student. In most cases, the student does not look at his program after it is graded, because he is focussed on upcoming deadlines rather than prior submissions. Only in a course where subsequent assignments build upon earlier ones would a student be motivated to review his old projects. It is essential to integrate the diagnostic tool into the student's development environment so he can get feedback while still working on his program. As Eclipse is now required for UMass Amherst's CMPSCI 187, Programming with Data Structures, I decided to implement FrenchPress as a plug-in. The student can easily install the software and run it repeatedly as he changes and improves his code. Eclipse is a widely used development environment and has a well-established and free mechanism for software distribution. I created an update site for FrenchPress to take advantage of the *Install New Software* and *Check for Updates* features in Eclipse. The student runs FrenchPress by selecting a menu item in the Package Explorer view. He can choose to analyze a single `.java` file, or all the `.java` files in the `src` folder of his Java project. The single file mode of operation is illustrated in Figure 1.1. FrenchPress writes a feedback file for each `.java` file it analyzes. The plug-in creates a `frenchpress` folder in the student's Java project and stores all its feedback files there. The student can review the feedback whenever he wishes.

1.4.3 Easy for the professor

Some systems that give automated feedback to students (for example, the Environment for Learning to Program [36], the Java Critiquer [28, 29], and the MIT system for novice Python programmers [33]) require a substantial effort from the instructor in the form of customized diagnostic checks or a model solution for each programming assignment. Conscious of the many claims on the time of both professors and teaching assistants, I sought an approach that would be less labor-intensive for them while still offering a benefit to their students. I created a generic tool that demands nothing from the instructor, and has no knowledge of what problem the student is attempting to solve.

There is a wide range of emphasis in data structures and algorithms courses, as evidenced by the multiplicity of textbooks on offer to teach this material in Java. FrenchPress diagnostics are not limited to a particular course syllabus. The rules described in Chapter 3 embody principles of good programming that are appropriate for assignments from any of these textbooks. Any professor who uses Eclipse in her course can take advantage of FrenchPress. The FrenchPress update site will enable her students to install the plug-in without additional effort on the part of course staff.

1.5 Design tradeoffs in a diagnostic tool

In designing a pedagogical program analysis system, one faces a tradeoff between its range of applicability and the quality of its feedback. At one end of this spectrum, I could create a tool to diagnose only a fixed set of programming assignments. The system could give detailed feedback because I would have a good idea of what I expected to see in a student solution. However, a small repertoire of assignments would be useful for only one course and one instructor. Repeating the same set of programming projects semester after semester could also increase the risk that students would forgo the learning experience of writing their own code, opting instead

to “inherit” solutions from friends who have already taken the course. Extending the diagnostic system to accommodate new programming assignments would be very labor-intensive.

At the other end of the spectrum, I could produce a generic tool that has no knowledge of the specific programming assignment the student is attempting to solve. A generic tool could be used in many different courses, but would not be able to provide the rich feedback one could achieve with a set of instrumented assignments.

Between these two extremes lies an intermediate approach: a system that places no a priori restrictions on the programs it can analyze, but requires the instructor to provide a model solution for the assignment before any student submissions are processed. The student’s program could be compared to the model solution with respect to features such as the number of classes, choice of instance variables, number of methods, and depth of the call graph. The instructor would specify via a checkbox interface which of these aspects she wants the diagnostic tool to examine. This approach is feasible only if the assignment description gives clear guidance about program structure. The professor would have to tell her students how many class definitions she expects, and what are the important instance variables and methods in each class. She might provide the `main` method for the application so her students can see how it instantiates classes and calls methods they have to write.

I decided against the intermediate approach for two reasons. It goes against my objective (Section 1.4) to avoid increasing the workload of the teaching staff. It is certainly desirable to write a model solution before releasing a programming assignment to students, but not every professor adheres to this practice and FrenchPress should not impose additional obligations. I also think the inherent creativity of programming (in Java or any other language) makes it difficult to give useful stylistic advice by comparing the student’s program to a reference solution. Such a comparison is feasible for small-scale “fill in the gap” exercises; indeed, the Environment for Learn-

ing to Program described on page 18 operates on this principle. But assignments of the scale that would be typical in a data structures and algorithms course involve many implementation decisions for which more than one option is acceptable. Even when the assignment description is carefully written, the student's implementation decisions may deviate from what the instructor anticipated. Telling the student how far his approach is from the recommended path might not help him recognize the stylistic weaknesses of his own code.

Another FrenchPress design choice is dynamic versus static analysis of student programs. Most of the assessment systems cited in Section 2.1 execute the student submission to check its output against the output of the model solution for the programming assignment. Evaluating the input/output behavior of the student's code is beyond the scope of FrenchPress. I chose static analysis with the understanding that the student has other ways to check his program for correctness, including of course JUnit tests written by the teaching staff or by the student himself. Static analysis is appropriate for all of the diagnostics described in Chapter 3. The one rule that would be easier to implement with dynamic analysis is the over-ambitious constructor rule described in Section 1.7. This rule is not currently part of the FrenchPress prototype.

1.6 FrenchPress is not a grading instrument

If the diagnostics reported by the plug-in were integrated into the grading rubric for the programming assignments, students would have a strong motivation to read the feedback messages. Conversely, if the grading criteria rely solely on the program's input/output behavior, the students might conclude that checking their work for hidden flaws is a waste of time. Nevertheless, I would not recommend that the professor consider FrenchPress as a grading instrument. My primary goal is to provide instructional feedback that helps the student improve, not a numeric score to be averaged in with other measures of homework performance. In the first few semesters

of use I would expect to encounter many problems with diagnostic rules that I did not discover during my testing. If students feel that their grades are in part based on inaccurate diagnostics from my tool, they will become justifiably indignant. The teaching assistant would be obliged to double-check the output of the automated analysis to determine which warnings are false positives that should be ignored in determining the grade. This would place an unreasonable burden on the TA, just the opposite of what I set out to do.

In a future version of the plug-in where the false positive rate has been driven near zero, it would be feasible to factor FrenchPress usage into the grading scheme of a course such as CMPSCI 187. I would advocate for positive rather than negative reinforcement: give students some extra points for using FrenchPress, instead of penalizing them for ignoring the automated feedback. The plug-in's diagnostic messages are intended as suggestions for improvement, not imperatives. The student should retain the autonomy to decide whether FrenchPress's guidance is appropriate in the context of his program.

1.7 Evolution of CMPSCI 187 projects

The idea of an Eclipse plug-in that could automate stylistic feedback grew out of my experience as a TA for CMPSCI 187, the data structures and algorithms course. Instead of hiring an instructor to teach this course every semester, UMass Computer Science rotates responsibility for CMPSCI 187 among the professors of the college. This introduces variability in the teaching style and the type of assignments given in the course. Steadily increasing class sizes have also changed the character of the programming projects. Higher enrollments have forced a move to automated grading of submissions based solely on input/output behavior. Automated grading requires greater uniformity in the submitted programs than might have been the case five or ten years ago when the submissions were graded by TAs.

I devised my diagnostic rules by examining student submissions from the Spring 2008 offering of CMPSCI 187 taught by Prof. Robert Moll. The class size was about 45. Section A.1 reproduces the description for the first programming assignment Prof. Moll gave in this course. He explains the problem to be solved and gives an example of correct output. The only directive related to the implementation is "two class application".

As the class size and the grading burden grew, CMPSCI 187 assignments evolved in the direction of giving students more guidance. By the fall of 2011, Prof. David Mix Barrington had approximately 105 students. His first programming assignment can be found in Section A.2. Prof. Barrington's project description includes details of the classes, fields, constructors, methods, and exceptions the students must incorporate in their solution. He also gives an example of the data structure as well as values returned by the `toString` and other methods. No code is provided as part of the assignment.

In the fall of 2013, CMPSCI 187 had grown to an initial enrollment of 151. Prof. James Allan's assignments included a description of the classes and methods to be implemented, and an example of correct output. For some assignments he provided Java class definitions, or interface definitions the students had to implement. The assignment specification included input files and a description of the expected output. For some projects the course staff gave students a test harness (not JUnit). Grading was automated; students could submit their program and get a grade immediately. If they were disappointed by their grade, they could improve the program and resubmit repeatedly up to the deadline for the assignment. As the semester progressed, Prof. Allan placed a limit on the number of resubmissions, to encourage the students to test their code in other ways. Section A.3 shows the first programming assignment for this course.

One semester later in Spring 2014, initial enrollment in CMPSCI 187 had reached 191. Prof. Gerome Miklau and Lecturer Tim Richards gave their students not only a written description for each assignment but also a Java project to be imported into Eclipse. The Java project contained any starter code and interfaces the instructors chose to provide, as well as JUnit test classes. The student was allowed to add classes or methods to his `src` folder but could not deviate from the program structure expected by the JUnit tests. Grading was fully automated and included more rigorous JUnit tests that were not released to the students. Section A.4 gives the first programming assignment for the Spring 2014 edition of Programming with Data Structures.

The changes in CMPSCI 187 assignments over the years affected the development of FrenchPress. The original plan for the plug-in included two rules that fall into the category of misconceptions about object-oriented programming. Many inexperienced Java programmers have only a shaky grasp of the concept of inheritance. Some students appear to confuse *Is-a* with *Has-a*: they use inheritance when composition would be suitable. For example, they create a class that extends `ArrayList` when they really should have given their class an `ArrayList` instance variable. I wrote a rule that would signal an inappropriate inheritance relationship when a subclass does not override any method of its superclass. An exception to this rule would be a subclass that implements an interface the superclass does not implement. This rule would catch cases in which the student declares his class to extend a Java library class, or makes spurious inheritance relationships between two classes of his own devising.

Another error in some student programs is a constructor that does not stop at constructing an object but tries to run the entire program by calling instance methods. This anomaly is difficult to detect with static analysis, but FrenchPress could at least warn the student that a class constructor should only call `final` or `private` methods (see [39, pp. 70–71]). The class definition in Chapter 3, Figures 3.7 and 3.8 exhibits

this flaw. Appendix C shows a student program from 2008 that would have triggered both the inappropriate inheritance and over-ambitious constructor rules.

After consulting my committee, I eliminated these two rules from the FrenchPress prototype because they seemed less applicable to CMPSCI 187 as it is now taught at our institution. CMPSCI 187 assignments have of necessity become more and more constrained, whereas these two rules are geared toward projects in which the student develops his program “from scratch” without much design guidance from the instructor. Obviously one is more likely to see design flaws when the student is given the freedom and the responsibility to organize his program as he sees fit. To become a competent programmer, the student must progress from filling in the missing pieces of a pre-existing structure to creating the structure for himself. New inheritance and constructor rules could be helpful as the student is struggling to make that transition, whether it occurs in his second programming course or not until the third.

CHAPTER 2

RELATED WORK

FrenchPress's target population of advanced beginners is not well served by the program analysis tools that are currently available. Existing style checkers and automatic assessment systems developed in academic environments are aimed at students who are just learning how to program. They flag mistakes that will cause compile-time or run-time errors, and common confusions that can lead to incorrect program behavior (for example, `=` in place of `==`). My research aims for the silent flaws that do *not* cause compile-time or run-time errors, or produce incorrect output: flaws that reveal the student's misunderstanding of the object-oriented programming paradigm. These flaws are described in Chapter 3.

On the other end of the spectrum, a professional program analysis tool such as FindBugs [12, 18] is too complex for students in a course such as CMPSCI 187. FindBugs does not report many of the stylistic flaws one sees in student submissions, because it is looking for more subtle errors an experienced programmer might make (synchronization of threads, vulnerability to malicious code). Bug reports written for professional programmers can be intimidating to learners who have completed only a few semesters of Java. FrenchPress gives students explanatory messages in a simple vocabulary they can read without frustration. FrenchPress bridges the gap between pedagogic support for novices that CMPSCI 187 students have outgrown and industrial strength diagnostics they are not yet ready to tackle.

2.1 Automated assessment systems in academia

Many course management systems designed for Computer Science provide a function to evaluate student submissions and assign grades. The most common form of assessment is to run the student program on a test suite created by the author of the programming exercise. The author specifies correct input/output pairs in a configuration file, or writes a model solution that will generate such pairs. ASSYST [20] and Web-CAT [11] require the student to submit a test suite, which is itself evaluated for completeness.

At UMass, students in the introductory course of the computer science major (CMPSCI 121) rely on *Interactive Java: An Online Approach to Java Learning*, by Prof. Moll [24]. This online textbook reinforces programming language concepts with embedded exercises that prompt the reader to write short segments of code. Like FrenchPress, Interactive Java gives its user immediate feedback: the student's solution is evaluated on a remote server the moment he submits it. Interactive Java tests the code for correctness and, if needed, offers help specifically written for each programming exercise.

Testing input/output behavior does not give the student any feedback on the style or design of his program. Many systems compensate for this drawback by incorporating a software quality metric into their scoring. These metrics are a combination of quantitative measures of the program such as number of comments, length of identifiers, and length of methods. Some automated assessment systems offer static analysis of qualitative aspects of the student's program. ELP [36], Expresso [19], and Web-CAT aim to discourage bad programming practices such as unused variables or risky side effects. Web-CAT merges diagnostics from both Checkstyle [7] and PMD [27] into a unified report, so the student can inspect his code with all the warning messages displayed line by line. Cedilh [5] and CourseMarker [15, 16, 17] enable the instructor to define exercise-specific "features" in the form of regular expressions that

are matched against the student submission. In Scheme-Robo [32], the assignment author can write a customized structural analysis to enforce requirements that the student include (or conversely, avoid) particular language constructs in his solution.

Table 2.1 summarizes the evaluation mechanisms available in (a subset of) the many systems that have been developed for the automated assessment of programming exercises. Most of the systems listed are geared toward students taking their first programming course, and they diagnose errors beginners are more likely to make. As an example, Expresso is a pre-compiler for Java programs that helps beginners avoid common mistakes that can lead to incomprehensible compiler messages or unexpected run-time behavior. Expresso will flag errors such as misuse or omission of parentheses, braces, and brackets; confusion of `=` and `==`; string comparison with `==` instead of `equals`; insertion of a semicolon where it does not belong; and invocation of a method with arguments of the wrong types. The target audience for my diagnostic tool is not absolute novices but students who already have some programming experience, albeit limited. I expect that these students have already learned to interpret compiler error messages. FrenchPress requires that its input program compile correctly.

Two research efforts that come closer to what I have in mind are the Environment for Learning to Program (ELP) from the Queensland University of Technology [36] and the Java Critiquer from Northwestern University [28, 29]. ELP does both dynamic and static analysis of program snippets submitted by students to complete "fill in the gap" programming exercises in the introductory Java course. The static analysis module operates on an XML representation of the abstract syntax tree. It computes statistics for each gap (total number of variables, statements, expressions) as well as the cyclomatic complexity [23]. It checks for unused variables, unused parameters, redundant logical expressions, numeric literals that should be named constants, and other stylistic blunders. ELP also performs a structural similarity analysis between

Table 2.1. Automated assessment systems for introductory programming courses

System	Origin	Refs	Languages	T	M	S
ASSYST	U Liverpool	[20]	Ada, C	×	×	
Automatic Marker	U Cape Town	[34]	Java	×		
BOSS	U Warwick	[22]	Java, Perl	×	×	
Cedilh	U Nottingham	[5]	C, C++, Prolog, Z	×	×	×
CourseMarker	U Nottingham	[15, 16, 17]	C++, Java	×	×	×
EduComponents	U Magdeburg	[1]	Haskell, Lisp, Prolog, Python, Scheme	×		
ELP	Queensland U Tech	[35, 36, 37]	C#, Java	×	×	×
Expresso	Bryn Mawr Coll	[19]	Java			×
Java Critiquer	Northwestern	[28, 29]	HTML, Lisp, Java			×
HoGG	Rutgers	[25]	Java	×		
Scheme-Robo	Helsinki U Tech	[32]	Scheme	×		×
TRY	RIT	[30, 31]	Any	×		
Web-CAT	VA Tech	[11]	Java	×		×
WebToTeach	Brooklyn Coll	[2, 9]	Ada, C, C++, Fortran, Java, Pascal	×		

T stands for dynamic testing; M for software metric; S for static analysis.

the student's submission and the instructor's model solution(s) for the exercise. The structural similarity analysis compares simplified versions of the abstract syntax trees and reports to the student any discrepancies between his code and the instructor's. For example, if the instructor's solution has two loops but the student's has only one, this might alert the student that he has missed part of the exercise. Truong *et al.* acknowledge that this approach works only with the "fill in the gap" type of exercise because these are so short there is relatively little room for structural variation. The structural similarity analysis could not be extended to more substantial assignments, involving multiple class definitions, that students would encounter in a data structures and algorithms course.

Qiu and Riesbeck, creators of the Java Critiquer, advocate an approach of incremental development for educational critiquing systems. The Java Critiquer provides an authoring interface that allows professors and teaching assistants to write new critiques as the need arises. The software was integrated into the grading process for some introductory programming courses at Northwestern University. The human grader first writes a critique in response to a student's blunder, then saves the text for future use. If the problem shows up frequently, the grader can convert the hand-written critique into a static analysis rule so that the system can recognize the error and generate appropriate feedback automatically. The left-hand side of each rule is a pattern that matches the problematic code, and the right-hand side of the rule is an appropriate diagnostic message for the student. The pattern can be written either as a regular expression or as a structure in Java Markup Language (JavaML), an XML representation of the Java source code [4]. JavaML has the expressive power of a context-free language. Variables in the JavaML pattern are bound to matching parts of the student program. The teacher can reference these variables in the right-hand side of the rule to quote the student's code in his critique. As the teacher authors new critiques, he monitors the system's application of those rules and refines the left-hand

side of any rule that results in too many false positives or false negatives. When the rules reach an acceptable level of accuracy, the professor releases them to the students so they can run the automated critiques for themselves and get feedback before the final submission of their assignment. The critiques are divided into a default rule set that applies to any Java code, and task-specific rule sets that each apply to a specific programming assignment. One of the task-specific rules warns the student that defining a `Circle` to be a subclass of `Point` is incorrect. The default rule set includes critiques related to boolean expressions, increment operators, unnecessary parentheses, and floating point data types (use `double` not `float`). It also includes a prohibition of `public` instance variables.

If the examples discussed in Qiu and Riesbeck's papers are representative, many of the Java Critiquer's rules address localized stylistic issues. One rule instructs the student to use `++var;` instead of `var = var + 1;` (not clear why the authors think this rule is worth keeping). The rule against `public` instance variables is the one area of overlap between the Java Critiquer's default rule set and the program design flaws that are my focus. The Java Critiquer can analyze only one class file at a time. Diagnosing an inappropriate inheritance relationship between the `Circle` and `Point` classes is possible only because the critique pertains to a particular programming assignment, so the author knows which class names are likely to appear.

The MIT system described in [33] tackles a class of problem FrenchPress does not cover: logic errors that cause incorrect output. MIT's system gives automated feedback to novice Python programmers. To formulate feedback for a programming exercise, it requires a reference solution for the exercise and a set of corrections for mistakes the instructor anticipates students will make. The system gives hints to help the student transform his program into one that matches the expected behavior. The instructor controls how much of a hint she wants to give the student, ranging from the line number of an error to a suggestion of exactly what transformation to make on

the original code. The suggested transformations may improve program correctness but do not generally improve program style.

2.2 Static analysis tools for professionals

Static analysis tools such as FindBugs and PMD are geared for large-scale professional projects. Their bug reports presuppose a sophisticated understanding of Java that college students are unlikely to attain in their first year of exposure to the language. Professional tools do not look for, and consequently do not find, the program flaws FrenchPress catches, precisely because the errors of an experienced software engineer are not those of a second-semester student. My diagnostic tool might flag a programming practice that looks dodgy in student code, while the same practice in the hands of a Java professional would not arouse suspicion.

Figures 2.1 through 2.7 compare FrenchPress and these two industrial strength static analysis tools applied to one student's `.java` file from the Fall 2014 CMPSCI 187 course. Figures 2.1 and 2.2 show the student's code for a class called `HanoiSolverImp`. Findbugs has no diagnostic results for this file. PMD feedback for this file is shown in Figure 2.3, and FrenchPress feedback in Figure 2.4. FrenchPress comments on features of the code that FindBugs and PMD pass over in silence, because FrenchPress searches for flaws that are characteristic of an advanced beginner Java programmer. PMD trusts the programmer to choose the right fields for the `HanoiSolverImp` class. FrenchPress is on the lookout for local variables masquerading as instance variables, because some students in their second Java course still do not fully understand the distinction. PMD does not question the decision to make instance variables `public`, perhaps assuming those fields must serve some function in another class of the application. FrenchPress assumes that `public` instance variables as well as `public` methods not called outside of the defining class are both manifestations of the inexperienced programmer's disregard for access control.

```
1    package hanoi;
2
3    import structures.ListImp;
4    import hanoi.HanoiBoardImp;
5
6    public class HanoiSolverImp implements HanoiSolver {
7
8       public int ringNum;
9       public HanoiBoardImp gameBoard = new HanoiBoardImp();
10      public HanoiSolution solved;
11      public ListImp<HanoiMove> storage =
     new ListImp<HanoiMove>();
12
13      @Override
14      public HanoiSolution solve(int n) {
15         System.out.println("Solving for " + n + "\n");
16         gameBoard.setup(n);
17         ringNum= n;
18         if (n != 0)
19            moveRec(0, 2, n);
20         solved = new HanoiSolutionImp(ringNum, storage);
21         return solved;
22      }
23      public void moveRec(int a, int b, int k){
24         int c = 2;
25         if (a != 0 && b != 0)
26            c = 0;
27         else if (a != 1 && b != 1)
28               c = 1;
29
```

Figure 2.1. Student's HanoiSolverImp.java, part 1

```
30          if (k == 1){
31             gameBoard.doMove(new HanoiMove(a, b));
32             System.out.println("Made move from " + (a+1) +
      " to " + (b+1) + "\n");
33             storage.append(new HanoiMove(a, b));
34          }
35          else {
36             moveRec(a, c, k-1);
37             gameBoard.doMove(new HanoiMove(a, b));
38             System.out.println("Made move from " + (a+1) +
      " to " + (b+1) +"\n");
39             storage.append(new HanoiMove(a, b));
40             moveRec(c, b, k-1);
41          }
42       }
43
44    }
```

Figure 2.2. Student's HanoiSolverImp.java, part 2

```
src/hanoi/HanoiSolverImp.java:25: Avoid if (x != y) ..; else ..;
```

Figure 2.3. PMD feedback for HanoiSolverImp.java

```
FrenchPress 1.3 feedback for HanoiSolverImp.java

Variables such as
     ringNum (8)
     solved (10)
are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.

Instance variables such as
     ringNum (8)
     gameBoard (9)
     solved (10)
     storage (11)
should not be declared public.  If you need to read or change
a variable V outside of the class, define getV and setV methods.
Or, if V is really a class constant, declare it public static final.

These methods are declared public but never called outside their
own class:
     moveRec (23)
If you meant these to be helper methods used only within this class,
they should be declared private instead.
```

Figure 2.4. FrenchPress feedback for HanoiSolverImp.java

```
1     package hanoi;
2
3     import structures.ListImp;
4     import hanoi.HanoiBoardImp;
5
6     public class HanoiSolverImp implements HanoiSolver {
7
8        private HanoiBoardImp gameBoard = new HanoiBoardImp();
9        private ListImp<HanoiMove> storage =
       new ListImp<HanoiMove>();
10
11       @Override
12       public HanoiSolution solve(int n) {
13          gameBoard.setup(n);
14          int ringNum= n;
15          if (n != 0)
16             moveRec(0, 2, n);
17          HanoiSolution solved = new HanoiSolutionImp(ringNum,
      storage);
18          return solved;
19       }
20       private void moveRec(int a, int b, int k){
21          int c = 2;
22          if (a != 0 && b != 0)
23             c = 0;
24          else if (a != 1 && b != 1)
25                c = 1;
26
27          if (k == 1){
28             gameBoard.doMove(new HanoiMove(a, b));
29             storage.append(new HanoiMove(a, b));
30          }
31          else {
32             moveRec(a, c, k-1);
33             gameBoard.doMove(new HanoiMove(a, b));
34             storage.append(new HanoiMove(a, b));
35             moveRec(c, b, k-1);
36          }
37       }
38
39    }
```

Figure 2.5. Revised HanoiSolverImp.java

```
src/hanoi/HanoiSolverImp.java:8: Private field 'gameBoard' could
be made final; it is only initialized in the declaration or constructor.
src/hanoi/HanoiSolverImp.java:9: Private field 'storage' could
be made final; it is only initialized in the declaration or constructor.
src/hanoi/HanoiSolverImp.java:18: Consider simply returning
the value vs storing it in local variable 'solved'
src/hanoi/HanoiSolverImp.java:22: Avoid if (x != y) ..; else ..;
```

Figure 2.6. PMD feedback for revised HanoiSolverImp.java

```
FrenchPress 1.3 feedback for HanoiSolverImp.java

Good work!  FrenchPress found no flaws in your code.
```

Figure 2.7. FrenchPress feedback for revised HanoiSolverImp.java

The student revised her code to follow FrenchPress's suggestions. Note the changes on lines 8, 9, 14, 17, and 20 of Figure 2.5. Running FindBugs on the revised `HanoiSolverImp.java` again produces no feedback. PMD has more to say about the revised version than it did about the original (Figure 2.6), as two `public` instance variables have become `private`, and two instance variables have become local variables of the `solve` method. PMD correctly points out that the local variable `solved` could easily be eliminated. The same is true of local variable `ringNum`, although none of the three static analysis tools can detect this. The revised class definition triggers none of FrenchPress's diagnostic rules, as the student has fixed the three flaws identified in the initial version (Figure 2.7).

2.3 Code smells

FrenchPress is similar in spirit to systems such as Stench Blossom [26] and JDeodorant [13, 38, 14, 21] that alert programmers to *code smells*, questionable program features that indicate the code should be refactored or redesigned. Stench Blossom offers programmers an interactive visualization that warns them of code smells as they are

Table 2.2. JDeodorant code smells and the corresponding refactorings

Code smell	Refactoring
FEATURE ENVY	MOVE METHOD
STATE CHECKING	REPLACE CONDITIONAL WITH POLYMORPHISM
	REPLACE TYPE CODE WITH STATE/STRATEGY
LONG METHOD	EXTRACT METHOD
GOD CLASS	EXTRACT CLASS

writing Java in Eclipse. The tool has three levels of visualization. *Ambient View* shows the relative strength of smells in the method the programmer is currently editing. *Active View* tells the user the name of each smell identified. *Explanation View* displays a summary from the smell analyzer and points to the code causing the smell. Stench Blossom can recognize eight code smells: DATA CLUMPS, FEATURE ENVY, MESSAGE CHAIN, SWITCH STATEMENT, TYPECAST, INSTANCEOF, LONG METHOD, and LARGE CLASS.

JDeodorant detects four categories of code smell and can automatically refactor the code to eliminate them. The code smells and their remedies are shown in Table 2.2. JDeodorant calculates and ranks multiple candidate refactorings that would remedy each code smell it has identified. The user can select a refactoring, preview its effects, and have JDeodorant apply it automatically to his program.

FrenchPress flaws can be considered code smells specific to advanced beginner Java programmers. The poor programming practices FrenchPress highlights for students are more localized than the smells identified by Stench Blossom and JDeodorant. The repairs FrenchPress suggests in feedback messages are much smaller in scope than the refactorings JDeodorant proposes. I chose not to automate any code modification because I think the student learns more by thinking through the feedback he gets from FrenchPress, then deciding for himself whether and how he will change his program. To test this hypothesis, I would have to implement a new FrenchPress plug-in that would not only diagnose program flaws but also repair them using functions

provided in Eclipse to manipulate the abstract syntax tree. I could then conduct a controlled classroom trial to compare students using the original version of the plug-in to students using the new version of the plug-in. In the classroom trial reported in Chapters 5 and 6, I had difficulty estimating how well students had learned from FrenchPress feedback messages. I would have to overcome this difficulty before I could evaluate whether it is more effective to let the student make his own code modifications or have FrenchPress correct the flaws it finds.

CHAPTER 3
DIAGNOSTICS

The current version of the FrenchPress prototype comprises seven diagnostic rules. I developed these rules after examining a set of student programs submitted for the UMass data structures and algorithms course in 2008. I focussed on programming practices that

- do not affect the output of the program, but violate professional Java coding standards;
- are amenable to automated analysis;
- occur frequently enough among the students to be worth investing effort to correct.

I chose stylistic flaws I thought would not cause much debate among experienced programmers. By "amenable to automated analysis" I mean I could come up with a succinct list of criteria to determine whether or not a particular program exhibited the flaw. There were, of course, other questionable features of these programs that I could not characterize with sufficient precision to diagnose reliably. These include:

- The student relies on instance variables as global variables to avoid more complex mechanisms for sharing information among methods. The symptom is a class whose methods are all parameterless with a return type of `void`.
- The student calls a method repeatedly with incrementing or decrementing parameters. The repetitious code should be rewritten as a `for` loop.

- The student uses a set of `String`s to represent the values of what should be an enumerated type.

These ill-defined rules are not amenable to automated analysis as they stand now, although it might be possible to refine them to the point where they could be implemented.

3.1 Four categories of flaws

I grouped FrenchPress's rules into four broad categories so that explanatory messages about related concepts would be displayed together. Some of these dodgy programming practices reveal the student's poor grasp of the object-oriented programming paradigm. Others are stylistic blunders one might see in any programming language. If the student's `.java` file contains flaws of multiple types, FrenchPress presents feedback in the order of the rules listed below. I ordered the categories according to my judgment of their severity. Misconceptions about the use of variables and access modifiers seem to me more serious in their potential consequences than redundant boolean expressions or unexpected `for` loop control variables. I want to first draw the student's attention to the issues I consider to be more significant. For each diagnostic I include below an example of the feedback the plug-in gives the student. If no diagnostic rules are triggered, the student gets the message

```
Good work!  FrenchPress found no flaws in your code.
```

3.1.1 Misuse of fields

Advanced beginners in Java do not always understand the significance of fields in a class definition. They might declare something as an instance variable but then use it as a class constant. Or they declare fields that are unrelated to the class's data representation; these are really local variables that have been inappropriately promoted to the status of instance or class variables.

Rule 1. Field could have been a local variable

A field could be made local if, in every method that uses the variable, it is always written before it is read, and it is read at least once. The same variable name might appear in several different methods but it is used as a local variable in each of them.

```
Variables such as
     game (8)
     m (9)
are declared at the class level but appear to function as local
variables.  Each of these variables could be declared locally
in each method where it is used.  To find all the places a
variable is used, select the variable name and Eclipse will
highlight every occurrence of that variable.
```

Student submissions that exhibit this flaw may be found in Figures 3.7 and 3.8 on pages 41–42, in Chapter 1, Figure 1.2, in Chapter 2, Figures 2.1 and 2.2, and in Appendix B.

Rule 2. Instance variable could have been a static final constant

The instance variable is initialized to a constant expression and never modified thereafter.

```
Instance variables such as
     numTrials (4)
could be declared static final (class constants) because they
are initialized to a constant value and never changed later.
```

Figure 3.1 shows a program that triggers this rule, leading to the message in Figure 3.2. The student revised his program in light of the feedback as shown in Figure 3.3 (note change on line 6). Other student submissions that exhibit this flaw may be found in Figures 3.7 and 3.8 and in Chapter 1, Figure 1.2.

3.1.2 Misuse of the public modifier

Inexperienced programmers often do not attach much importance to the principle of hiding the details of a class's data representation and internal methods. They

```
1    package structures;
2
3    public class ListImp<T> implements ListInterface<T> {
4
5       private int size = 0;
6       private int defSize = 10;
7       private Object[] contents = new Object[defSize];
8
9       @Override
10      public int size() {
11         return size;
12      }
13
14      @Override
15      public ListInterface<T> append(T elem) {
16         if (elem == null)
17            throw new NullPointerException("Null pointer.");
18         contents[size]= elem;
19         size++;
20         return this;
21      }
22
23      @Override
24      public T remove(int n) {
25         if(n > size)
26            throw new IndexOutOfBoundsException("No such location
    in the list; Index out of bounds");
27         T temp = (T)contents[n];
28         contents[n]= null;
29         for(int i = n; i < size; i++)
30            contents[i] = contents[i+1];
31         contents[size]= null;
32         size--;
33         return temp;
34      }
35
36   }
```

Figure 3.1. Student's ListImp.java

```
FrenchPress 1.3 feedback for ListImp.java

Instance variables such as
     defSize (6)
could be declared static final (class constants) because they are
initialized to a constant value and never changed later.
```

Figure 3.2. FrenchPress feedback for ListImp.java

routinely declare instance variables `public` or make a method `public` even though it is not part of the API for the class. Many times a method ends up `public` simply because the student copied and adapted a method definition from his lecture notes without thinking whether the access modifier was appropriate for his own program. This carelessness is understandable in the case where the student is working on his program alone (true for most courses at this level) and has no intention of re-using his code after the due date of the assignment. FrenchPress pushes back against these bad habits by reminding the student that access control is an important concept.

Rule 3. Instance variable declared public

The student should define getter and setter methods instead of exposing the class's instance variables.

```
Instance variables such as
     count (8)
should not be declared public.  If you need to read or change
a variable V outside of the class, define getV and setV
methods.  Or, if V is really a class constant, declare it
public static final.
```

A student submission that exhibits this flaw may be found in Chapter 2, Figures 2.1 and 2.2.

```
1    package structures;
2
3    public class ListImp<T> implements ListInterface<T> {
4
5       private int size = 0;
6       static final int defSize = 10;
7       private Object[] contents = new Object[defSize];
8
9       @Override
10      public int size() {
11         return size;
12      }
13
14      @Override
15      public ListInterface<T> append(T elem) {
16         if(size == contents.length)
17            contents = new Object[2*contents.length];
18         if (elem == null)
19            throw new NullPointerException("Null pointer.");
20         contents[size]= elem;
21         size++;
22         return this;
23      }
24
25      @Override
26      public T remove(int n) {
27         if(n > size)
28            throw new IndexOutOfBoundsException("No such location
     in the list; Index out of bounds");
29         T temp = (T)contents[n];
30         contents[n]= null;
31         for(int i = n; i < size; i++)
32            contents[i] = contents[i+1];
33         contents[size]= null;
34         size--;
35         return temp;
36      }
37
38   }
```

Figure 3.3. Revised ListImp.java

Rule 4. Non-static method declared public

If a `public` method is not called outside of its class, it does not need to be `public`. (This rule does not trigger if the method is inherited from a superclass or required by an interface the class implements.)

```
These methods are declared public but never called outside
their own class:
     moveRec (23)
If you meant these to be helper methods used only within
this class, they should be declared private instead.
```

Student submissions that exhibit this flaw may be found in Figures 3.7 and 3.8 (pages 41–42), in Chapter 1, Figure 1.2, and in Chapter 2, Figures 2.1 and 2.2.

3.1.3 Misunderstanding booleans

The boolean data type seems to baffle some students more than other primitive data types. FrenchPress recognizes two forms this misunderstanding can take in student code.

Rule 5. Integer variable used as a boolean flag

In some cases the student declares an integer variable but uses it as a boolean flag. A integer variable is suspect if it never gets any value other than 0 or 1, is compared to 0 or 1 in at least one expression, and never compared to any other values.

```
Variables such as
     Check (27)
are declared int but appear to function as boolean flags.
Instead of giving them the values 1 and 0, declare them as
boolean and give them the values true and false.
```

Figures 3.4 and 3.5 show a program that triggers this rule, leading to the message in Figure 3.6. This student ignored the feedback. Appendix B contains the program of a student who did not ignore the feedback from Rule 5. He changed his code and in the process created a case of Rule 6 (below), which he fixed in the next iteration of

```
1    package structures;
2
3    import java.util.Deque;
4    import java.util.Iterator;
5    import java.util.LinkedList;
6
7    public class PostOrderIterator<T> implements Iterator<T> {
8       private final Deque<BinaryTreeNode<T>> stack;
9       private int count=0;
10      public PostOrderIterator(BinaryTreeNode<T> root){
11         if (root==null)
12            throw new NullPointerException("");
13
14         stack = new LinkedList<BinaryTreeNode<T>>();
15         stack.push(root);
16      }
17
18      @Override
19      public boolean hasNext() {
20         if(count!=0)
21         {
22            BinaryTreeNode<T> toVisit=stack.pop();
23
24            if (stack.isEmpty())
25               return false;
26            stack.push(toVisit);
27            return true;
28         }
29         return !stack.isEmpty();
30      }
31
32      @Override
33      public T next() {
34         BinaryTreeNode<T> toVisit=stack.peek();
35         if (count!=0)
36         {
37            toVisit=stack.pop();
38            if (stack.peek()!=null&&stack.peek().hasLeftChild()&&
     toVisit==stack.peek().getLeftChild())
39            {
```

Figure 3.4. Student's PostOrderIterator.java, part 1

```
40             if (stack.peek().hasRightChild())
41             {
42                toVisit=stack.peek().getRightChild();
43                stack.push(toVisit);
44             }
45             else
46                return stack.peek().getData();
47          }
48          else if (stack.peek()!=null&&stack.peek().
      hasRightChild()&&toVisit==stack.peek().getRightChild())
49             return stack.peek().getData();
50       }
51       count++;
52       while (!toVisit.hasLeftChild()&&
      toVisit.hasRightChild())
53       {
54          toVisit=toVisit.getRightChild();
55          stack.push(toVisit);
56       }
57       while (toVisit.hasLeftChild())
58       {
59           toVisit=toVisit.getLeftChild();
60           stack.push(toVisit);
61           while (!toVisit.hasLeftChild()&&
      toVisit.hasRightChild())
62           {
63             toVisit=toVisit.getRightChild();
64             stack.push(toVisit);
65           }
66       }
67       return toVisit.getData();
68    }
69
70    @Override
71    public void remove() {
72       throw new UnsupportedOperationException();
73    }
74
75 }
```

Figure 3.5. Student's PostOrderIterator.java, part 2

```
FrenchPress 1.3 feedback for PostOrderIterator.java

Variables such as
     count (9)
are declared int but appear to function as boolean flags.
Instead of giving them the values 1 and 0, declare them as boolean
and give them the values true and false.
```

Figure 3.6. FrenchPress feedback for PostOrderIterator.java

interaction with the plug-in. The program is too long to include here; please refer to Appendix B. Figures 3.7 and 3.8 below show another example of Rule 5.

Rule 6. Redundant boolean expressions

Boolean expressions that compare a boolean variable to the constants `true` or `false` will be familiar to anyone who has read student code.

```
Boolean expressions such as
     isLegalMove(move) == false (11)
     temp.hasRings() != true (63)
are redundant and can be shortened.  If B is a boolean
expression,
B == true or B != false means the same thing as B
B != true or B == false means the same thing as !B.
```

The student program in Appendix B exhibits this flaw.

3.1.4 For loops

Rule 7. Inappropriate for loop control

Students occasionally use an instance variable or a constructor/method parameter as a loop control variable. Perhaps this reflects a misconception that their code is more economical or efficient if they re-use variables instead of declaring a new `int i` in their `for` loop.

```
Instance variables such as
     numPeople (39)
should not be used as for loop control variables.  It is
preferable to use a separate variable as in, for (int i = ...).
```

Rule 7 never triggered for any program in the classroom trial. The example in Figures 3.7 and 3.8 comes from the Spring 2008 edition of CMPSCI 187. It also triggered Rules 1, 2, 4, and 5 (Figure 3.9). Another student submission from Spring 2008 that exhibits this flaw may be found in Chapter 1, Figure 1.2.

3.2 Minimize duplication with prior work

In choosing which rules to implement in the plug-in, I tried to avoid duplicating work that had been done elsewhere. Eclipse itself gives many helpful warnings to the programmer, including `private` fields and local variables whose value is never used, `private` methods that are never called. Checkstyle enforces (among many others) naming and indentation conventions, and length limits on lines, methods, and files. PMD identifies (among many others) unused formal parameters as well as the same unused fields, local variables, and methods that Eclipse would mark with a warning. While advanced beginner Java learners could benefit from similar warnings, I wanted to focus my energy on feedback that was not available in other tools. There is nevertheless some overlap between FrenchPress and other diagnostic systems. PMD's *SingularField* is a limited version of Rule 1. Flaws caught by Rule 3 *Instance variable declared public* would also be flagged by Checkstyle's *VisibilityModifier* and by the Java Critiquer discussed on page 20. Rule 6 *Redundant boolean expressions* matches Checkstyle's *SimplifyBooleanExpression* and PMD's *SimplifyBooleanExpressions*. PMD's *AvoidReassigningParameters* overlaps with Rule 7 when method parameters are used as `for` loop control variables.

```
 1    public class BirthdayParadox{
 2      int[] birthdays;
 3      int count = 0;
 4      double numTrials = 10000;
 5      int numPeople=10;
 6      int temp;
 7
 8      public BirthdayParadox(){
 9        this.printResults();
10      }
11
12      public void birthdayGenerator(){
13        birthdays = new int[numPeople];
14        for(int a=0; a<numPeople; a++){
15          birthdays[a] = (int)(Math.random()*365+1);
16        }
17      }
18
```

Figure 3.7. Student's BirthdayParadox.java, part 1

3.3 Implementation

The implementation of FrenchPress diagnostics relies on the abstract syntax tree (AST) maintained by Eclipse for the program under construction. Implementing these rules directly on source code would not be feasible, as the rules rely on the structure of the program as well as its type hierarchy. Prof. Moss and I considered implementing the diagnostics at the byte code level, using the ASM [3] tool. However, I was concerned about the difficulty of incorporating ASM into an Eclipse plug-in and did not want to devote a large part of my research effort to system integration. I therefore relied on the rich repertoire of data structures and built-in functions Eclipse provides for plug-in developers.

Rule 1 *Field could have been a local variable* is implemented with a custom data flow analysis. The overall idea is that a field could be (and should be) replaced by (multiple) local variables if, in every method that accesses the field, it is always

```
19      public double percentCalc(){
20        count=0;
21        for(int c=0; c<numTrials; c++){
22          this.birthdayGenerator();
23          temp = 0;
24          for(int g=0; g<numPeople; g++){
25            for(int h=g+1; h<numPeople; h++){
26              if(birthdays[g] == birthdays[h]){
27              temp++;
28              }
29            }
30          }
31          if(temp>0){
32            count++;
33          }
34        }
35        return ((double)count)/numTrials;
36      }
37
38      public void printResults(){
39        for(numPeople=10; numPeople<31; numPeople++){
40          System.out.println("People: " + numPeople + " " +
    percentCalc());
41        }
42      }
43    }
```

Figure 3.8. Student's BirthdayParadox.java, part 2

```
FrenchPress 1.5 feedback for BirthdayParadox.java

Variables such as
     count (3)
     temp (6)
are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.

Instance variables such as
     numTrials (4)
could be declared static final (class constants) because they are
initialized to a constant value and never changed later.

These methods are declared public but never called outside their
own class:
     birthdayGenerator (12)
     percentCalc (19)
     printResults (38)
If you meant these to be helper methods used only within this class,
they should be declared private instead.

Variables such as
     temp (6)
are declared int but appear to function as boolean flags.
Instead of giving them the values 1 and 0, declare them as boolean
and give them the values true and false.

Instance variables such as
     numPeople (39)
should not be used as for loop control variables.
It is preferable to use a separate variable as in, for (int i = ...).
```

Figure 3.9. FrenchPress feedback for BirthdayParadox.java

written before it is read, and it is read at least once. If the field's value is initially overwritten in every method that subsequently reads the field, then the inter-method information content of that field is never used in the class. However, if there is a method that writes the field but never reads it, that method is essentially a setter method for the field. The value written in the setter might be used in another part of the program, so FrenchPress does not recommend this field should be made local.

The data flow analysis classifies each method with respect to each field of the class into one of four groups:

- *exposed read*: on some path, the first access to the field is a read
- *write only*: the field is written but never read
- *used*: the field is written and then read
- *none*: no access to field

If the field is *used* in at least one method and there are no methods in the *exposed read* or *write only* categories, then FrenchPress suggests to the student that the field could be declared locally in every method where it appears.

Rule 2 *Instance variable could have been a static final constant* examines the AST of the class to find each field that is

- not `static final`;
- initialized to a constant value where it is declared;
- never assigned or modified (to the same value or a different one).

An instance variable that meets these criteria generates diagnostic feedback suggesting to the student that the field appears to be a class constant.

Rule 4 *Non-static method declared public* exploits the Eclipse search mechanism that allows a programmer to locate all the calls to a specified method within the

Java project. The rule considers every `public` method defined in the class, excluding abstract methods, main methods, and constructors. The plug-in takes advantage of the type hierarchy maintained in Eclipse to determine whether the method in question is inherited from a superclass or required by an interface the class implements. If the type hierarchy does not explain why this method is `public`, the plug-in searches for calls to the method from outside the class. If no such call is found, there's a good chance the method does not need to be `public` and FrenchPress's feedback urges the student to make it `private`.

Rule 5 *Integer variable used as a boolean flag* examines the AST for any field or local variable declared of type `int` that exhibits the following characteristics:

- assigned only the value 0 or 1;
- compared to 0 or to 1 somewhere in the class;
- might be incremented by 1 but never decremented;
- does not appear in a `for` loop control expression.

FrenchPress feedback points out that the integer variable functions as a boolean in the program and should be declared that way.

Rules 3, 6, and 7 are straightforward to implement because each visits only one type of AST node. Rule 3 *Instance variable declared public* examines the modifiers of each `FieldDeclaration` node in the AST. Rule 6 *Redundant boolean expressions* examines the operator and right hand side of each `InfixExpression` node looking for comparisons with `==` or `!=` to a boolean literal. Rule 7 *Inappropriate for loop control* examines the initializers and updaters of each `ForStatement` node to determine whether they change the value of a field or method parameter.

CHAPTER 4

FRENCHPRESS PLUG-IN

4.1 Implementing an Eclipse plug-in

As Section 1.4.2 explains, implementing FrenchPress as an Eclipse plug-in offers many advantages for the students who want to install and use the software with a minimum of inconvenience. This implementation choice also offers advantages for the creator of the system. Eclipse gives plug-in developers a rich API repertoire for analyzing Java code, adding items to Eclipse menus, and displaying results in the Eclipse interface. The diagnostic rules listed in Chapter 3 rely in large part on Eclipse's built-in functionality. Eclipse provides a powerful mechanism for searching the type hierarchy and call graph of the Java program under construction. However, the documentation for these functions is sometimes out of sync with the reality of how they work in Eclipse. The programmer must also gain an understanding of the software hooks that connect the code of a plug-in to the internal data structures of the Eclipse environment.

4.2 Running the FrenchPress plug-in

To run FrenchPress, the user first selects what he wants to analyze in the Package Explorer view. He can run FrenchPress on all the `.java` files in the `src` folder of a project by selecting the project name. He can run FrenchPress on a particular `.java` file in his `src` folder by selecting just that filename. Having selected his target, the user right-clicks to get a menu that includes the item "Run FrenchPress". When

FrenchPress completes its analysis, it displays a pop-up dialog box containing feedback for the student to read. If the student's code triggers no diagnostic rules, the message reads "Good work! FrenchPress found no flaws in your code." If the code triggers one or more rules, FrenchPress presents feedback in the numeric order of the rules listed in Chapter 3. In either case, the student clicks an "OK" button to dismiss the window. Figure 1.1 on page 3 shows a screenshot of FrenchPress running on a single `.java` file.

The first time the student runs the plug-in on any part of his Eclipse Java project, it creates an archive folder called `frenchpress` at the top level within the project. The archive folder contains all the feedback files FrenchPress has produced for that project, and a `.jar` file of program history. For the student, the `frenchpress` folder is a repository of all the suggestions he has received concerning his code. For the researcher, the `frenchpress` folder holds a complete record of the student's interaction with the plug-in. Every time FrenchPress runs, it records a snapshot of all the `.java` files in the `src` folder at that moment (not just the file for which feedback is produced). These snapshots are all stored in a file named `<projectname>.jar`. If the student reruns the plug-in on the same project, FrenchPress adds to the existing archive folder. The `frenchpress` folder is saved along with the rest of the student's Java project, so the feedback files and the program archive `.jar` file accumulate no matter how many times he exits and re-enters Eclipse in the course of working on his assignment.

FrenchPress feedback is initially delivered in a dialog box that the student must close in order to continue composing his program. By storing feedback files in the `frenchpress` folder, the diagnostic messages remain available over the life of the project, not the life of a dialog box. The student can look back and review the feedback he received at any time. The name of each FrenchPress feedback file contains three components: the `.java` file whose analysis produced the feedback, the version of

FrenchPress that created the feedback, and a sequence number indicating how many times the student has analyzed the same `.java` file. For example, a feedback file with the name

```
ArrayHeap-fp15-03
```

means the feedback was generated by FrenchPress version 1.5 for file `ArrayHeap.java`, and the student already has two feedback files for `ArrayHeap.java` from earlier runs of the plug-in.

Knowing which version of FrenchPress produced the feedback file is important for evaluating the results. Over the course of the classroom trial, bug fixes and feature enhancements required the release of new versions of the plug-in. Some students ignored my instructions to update and persisted in running older software. A false positive from an old version of the plug-in might linger in these students' feedback files even after a new version was released.

The final element of the feedback filename indicates how many times the student has run FrenchPress on the same source file. This number is a rough indication of how seriously he engaged with the diagnostic tool. In an ideal scenario, the student runs the plug-in and gets substantive feedback, modifies his program according to FrenchPress's suggestions, and then reruns the plug-in to see whether his edits have eliminated the issues that were flagged the first time. It might take several iterations to address all the feedback messages. The numbering of the feedback files allows both the student and the researcher to track the history of the student's interaction with FrenchPress.

4.3 Running FrenchPress on the entire `src` folder

Students wrote comments on their surveys saying that for a project with many `.java` files, it was tedious to run FrenchPress on each file individually. In response, I implemented a new menu item to analyze all the student's files at once. For

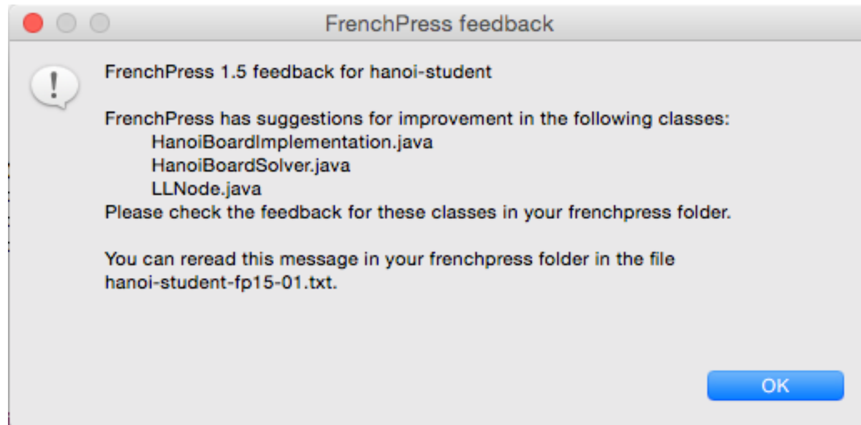

Figure 4.1. FrenchPress popup dialog for entire `src` folder

CMPSCI 187 assignments, the students were told to put all their work in the `src` folder of the project, although the professors provided starter code and JUnit tests in other folders. Therefore FrenchPress analyzes only class definitions found in the `src` folder (there are no diagnostics for interface definitions).

I had to re-think how to communicate diagnostic feedback when FrenchPress runs on the contents of the entire `src` folder. The user might feel overwhelmed if FrenchPress opened a separate feedback window for each file analyzed. FrenchPress instead displays one window that lists the `.java` files for which substantive feedback is recorded in the `frenchpress` folder (Figure 4.1). Later, the user can peruse the feedback created for each individual source file and make corrections if he wishes. If no source files generated any substantive feedback, the user gets a message of congratulations.

I also changed how the plug-in writes its archive `.jar` file when the student runs FrenchPress on the entire `src` folder. When FrenchPress runs on a single file, the plug-in records a snapshot of all the other `.java` files in the `src` folder along with the file under analysis and the feedback generated for that file. If the user analyzes two `.java` files in a row, the plug-in will record two snap-shots of the `src` folder. FrenchPress does not monitor the modification time of each file; it assumes the file might have

changed since the last time and takes a new snapshot. However, when FrenchPress runs on the entire `src` folder at once there is no need for multiple snapshots. In this case FrenchPress will archive a single snapshot of each `.java` file in `src` along with the feedback files, avoiding any repetition of source code in the `.jar`.

CHAPTER 5

CLASSROOM TRIAL OF FRENCHPRESS PLUG-IN

Under the supervision of the UMass Amherst Institutional Review Board, I have tested the FrenchPress plug-in with three groups of CMPSCI 187 students. CMPSCI 187, Programming with Data Structures, is the second Java programming class in the undergraduate computer science major sequence. These were formative evaluations, not controlled experiments. Running a classroom trial in which only half of the participants would have a chance to use a new software tool raised such difficult ethical issues that I did not attempt to create a control group.

The first time students tried FrenchPress was a pilot study over winter break starting in December 2013. I enrolled five students who had just completed CMPSCI 187 with Prof. James Allan. The pilot study allowed me to debug the mechanism for software distribution. I originally planned to have the students download the plug-in's `.jar` into the `dropins` folder of their Eclipse installation. This simple procedure described in the Eclipse documentation failed in practice for half of the students. Both Mac and Windows users had trouble with the dropin method. I undertook the more complicated procedure of building an Eclipse update site for FrenchPress, which worked for all the students.

I subsequently conducted two trials of FrenchPress in the CMPSCI 187 classroom, with the generous collaboration of the faculty teaching the course. The first occurred in the Spring 2014 semester, when Prof. Gerome Miklau and Lecturer Tim Richards shared responsibility for CMPSCI 187. This study covered three programming assignments. The implementation of FrenchPress's diagnostic rules was not yet complete,

so the prototype deployed in the spring of 2014 had limited functionality. For this reason the data collected in Spring 2014 are not included herein. The classroom trial for which results are reported in Chapter 6 occurred in the Fall 2014 semester, when Profs. David Mix Barrington and Mark Corner co-taught CMPSCI 187. This study covered four programming assignments.

5.1 Plan for classroom trial

Miklau and Richards (Spring 2014) alternated lectures for all the students together in one room, while Barrington and Corner (Fall 2014) each gave parallel lectures during the same time slot for half the class. The professors gave me 15 minutes of class time to explain the purpose of the plug-in, the potential risks and benefits of participating in the study, and the significance of the Informed Consent Form (Appendix D). Those who signed a consent form were enrolled in the study. To reward their participation in the trial, students were offered a small amount of extra credit—enough to push their final grade from (for example) a high B to a B$^+$. The Institutional Review Board was concerned that the offer of extra credit might put pressure on students to participate who would not otherwise have made that choice. To avoid that situation, I devised an alternate writing assignment that students could do to earn the same extra credit without using FrenchPress. The alternate assignment asked students to research on the internet three stylistic recommendations for Java programmers, and comment on whether they agreed or disagreed with each suggestion. In the Spring 2014 trial these alternate extra credit assignments were graded by one of the CMPSCI 187 TAs. In the Fall 2014 trial, I graded the alternate extra credit assignments.

Instructions for FrenchPress installation and usage were posted on the course website. Installation was a simple matter after I created an Eclipse update site for FrenchPress. Students were able to use the "Install new software" function of Eclipse to download the plug-in from its update site. For each programming assignment, the

student was required to run FrenchPress on one or more `.java` files from his Eclipse project and answer a short online user satisfaction survey (Appendix E) administered through SurveyMonkey. Students who completed these two tasks for each of the assignments were rewarded with full extra credit for their participation in the classroom trial. However, many students did not complete both steps for each assignment. They received partial extra credit as appropriate.

As noted above, the Spring 2014 classroom trial did not produce a complete data set because the plug-in was not fully implemented. The principal benefit to my research from that experience was learning how to manage a classroom trial. The surprises started with the informed consent process. I was encouraged that many students were willing to participate, but I did not expect that some of them would fill in their Informed Consent Form with a pencil, while others submitted a .pdf with their name typed on the keyboard instead of signed by hand. I added a paragraph to the Fall 2014 consent form explaining how to complete the form properly. I found the overhead of communication with both the professors and the study subjects took more of my time than I had anticipated. I sent repeated reminders to students about completing the online survey for each assignment, and handled emails from those who could not remember whether they had already done so.

5.2 User satisfaction survey

The user satisfaction survey for each assignment was open for one week following the due date. There was no overall survey at the end of the semester. I decided to administer multiple surveys for two reasons. First, I wanted to capture the students' impressions of the plug-in soon after they finished working with it, before their memory of the FrenchPress feedback had faded. Second, different programs written for different assignments elicit different feedback from FrenchPress. The student might find the diagnostic messages he received on one assignment were helpful, but those for

the next assignment were less helpful. Instead of asking the student for his average judgment over all assignments, I wanted to obtain a more specific evaluation of his experience on each program.

5.2.1 Keep it short

Given the small amount of extra credit offered for participating in the study, and the reality of undergraduates juggling many courses at once, my top priority in writing the survey was brevity. I was concerned that if the survey were too long, students would find it onerous and drop out of the trial because they did not want to be bothered with a tedious task after each assignment. Perhaps if I had had only one survey at the end of the semester I would have allowed myself more questions. Since the students had to repeat the survey multiple times, I was determined to keep it as short as possible.

I wrote the survey questions to include both positive and negative terms so that I would not bias the responses in one direction. For example, I included the question,

> For this assignment, was the feedback from FrenchPress confusing or easy to understand?

with possible responses

- FrenchPress found no flaws in my program
- Very confusing
- Moderately confusing
- Neither confusing nor easy to understand
- Moderately easy to understand
- Very easy to understand

The alternative would have been a statement that more closely matches the standard five-level item in a Likert-type scale [8]:

For this assignment, I found the feedback confusing.

with possible responses

- FrenchPress found no flaws in my program
- Strongly agree
- Agree
- Neither agree nor disagree
- Disagree
- Strongly disagree

But then to balance out the positive and negative vocabulary I would have needed another question on the survey of the form,

For this assignment, I found the feedback easy to understand.

with the same possible responses. This would have doubled the length of the survey and the time students would need to complete it. I was afraid including many questions to ask essentially the same thing would frustrate the students and lead them to abandon the classroom trial.

5.2.2 Survey questions

The survey after the first assignment of the classroom trial included three questions about the operating system and the installation of the FrenchPress plug-in. These questions did not appear on subsequent surveys. All the surveys included the questions numbered 1 and 5–11 below. The last question was followed by a text box

in which students could type any comments they wished to communicate to the researcher. Please refer to Appendix E for the full survey including all response choices for these questions.

1. Please enter your student ID number for this survey to count toward extra credit.
2. What operating system are you using? Please include the edition/version (e.g. Windows 7 Professional or OS X 10.9.4).
3. How easy or difficult was it to install FrenchPress?
4. How long did it take you to install FrenchPress?
5. How often did FrenchPress crash or "freeze up" on you for this assignment?
6. Did FrenchPress find any flaws in your program for this assignment?
7. For this assignment, was the feedback from FrenchPress confusing or easy to understand?
8. For this assignment, was the feedback from FrenchPress helpful or unhelpful?
9. Did the FrenchPress feedback for this assignment lead you to change your program?
10. Are you satisfied or dissatisfied with the performance of FrenchPress on this assignment?
11. How can we improve FrenchPress?

5.2.3 Limitations of survey data

The first step in interpreting the survey data was to eliminate duplicate and bogus survey responses. Sometimes a student who could not remember completing the

survey for a particular assignment would take it a second time, creating a duplicate in the data set. As the first question on the survey asked for student ID, it was possible to identify these duplicates and delete them. A few students took the survey although there was no evidence they had run the FrenchPress plug-in on the project they submitted for the corresponding assignment. It seems they were trying to game the system and get full extra credit without fulfilling the requirements of the classroom trial. I weeded out the surveys from these students.

It became obvious that some of the survey questions were poorly worded and subject to multiple interpretations. Question 6,

> Did FrenchPress find any flaws in your program for this assignment?

was very misleading. I meant to ask, did you get any substantive diagnostic feedback, something more than a "Good work" message? Some students understood the question as, did you get any feedback that you felt truly reflected a poor programming practice? If the student got feedback he judged to be a false positive or just not worth the trouble to consider, he might answer no to this question when I expected him to answer yes because he had received substantive feedback. Of course the feedback files stored in the student's `frenchpress` folder show exactly what messages FrenchPress gave for each `.java` file analyzed. So I can get the answer to the question I thought I was asking without relying on the student to tell me.

The ambiguity of question 6 might also affect the responses to questions 7 and 8 about the quality of FrenchPress feedback. The first possible answer for each of these questions was

> FrenchPress found no flaws in my program

which served the purpose of "Not applicable" for those students who could not comment on how confusing/easy to understand, how helpful/unhelpful the feedback was because they got only a "Good work" message. Students who did not understand

what I meant by question 6 might also have answered questions 7 and 8 in a way I did not anticipate.

The survey responses to question 9,

> Did the FrenchPress feedback for this assignment lead you to change your program?

did not match the FrenchPress program archive for all students. Perhaps when he completed the survey after submitting his project, the student could not remember how or why he edited his program. I relied on FrenchPress archive data only to determine what percentage of users were prompted to modify their code in light of the plug-in's diagnostic messages.

5.3 Culling data from the FrenchPress archive folder

One of the first questions one wants to ask about the classroom trial is, which diagnostic rule triggered most often? Even something that seems straightforward, counting feedback messages, requires careful attention to avoid counting duplicate messages as if they were new. Many students ran the plug-in multiple times but made no changes to their code in between. A cursory glance at the feedback files might give the impression that a rule has triggered multiple times, but more careful examination reveals the same mistake recorded again and again. A similar problem arises if the student made other changes to his code that did not affect the mistake being counted. The message generated by a different rule might disappear due to the changes, but the untouched mistake will elicit the same feedback as before, perhaps with a different line number. In these cases the feedback message counts once and the repetitions of that message do not increase the total.

It can also be difficult to determine whether the student made changes to his program following suggestions he received from FrenchPress. If the `frenchpress` folder contains multiple feedback files for the same class definition, these might indicate that

the student has modified his code to remedy a shortcoming identified by the plug-in. However, if the student runs the plug-in no more than once per `.java` file and then submits his assignment, the only way to tell if he changed anything is to compare the `.jar` in his `frenchpress` folder to the final submission of the program.

5.4 Off-target feedback

The program and feedback archives in the `frenchpress` folders submitted by students reveal instances where the suggestions offered by the plug-in were inappropriate for the program under analysis. I refer to these diagnostic messages as "off-target". Some but not all are false positives. The sources of off-target feedback include

- implementation errors in Rules 1, 2, and 4
- student neglected to update software
- student ran plug-in at an early stage of development
- student ran plug-in on a test class
- variable or method is unused
- getter/setter method

5.4.1 Implementation errors

Rule 1 *Field could have been a local variable*, Rule 2 *Instance variable could have been a static final constant*, and Rule 4 *Non-static method declared public* caused some false positives due to implementation errors. Rules 1 and 2 suffered from a problem related to the key Eclipse generates for the binding associated with a field or local variable. Rule 4 gave unnecessary warnings in cases where a public method was not called outside of the defining class in the student's code, but it was called in the JUnit test classes provided by the instructor.

5.4.2 Software out of date

Some students did not update their software when I asked them to. The first version of the plug-in that I made available to students in the Fall 2014 semester was FrenchPress 1.2. Bug fixes and feature enhancements over the course of the classroom trial led to three software releases, bringing the version to 1.5 by the last assignment in the study. Updating the plug-in demanded minimal effort of the students, they just had to select *Check for Updates* from the Eclipse *Help* menu. Nevertheless, some students were slow to update their software and three even reached the end of the classroom trial without updating at all. Figure 5.1 shows for each of the four assignments how many users were running which version of FrenchPress. Note that the totals do not necessarily count the same participants on all assignments; a few students floated in and out of the study population. I will refer to the four assignments as projects 3, 4, 5, and 6 to be consistent with the numbering they had in the course. An asterisk for a particular project and version number indicates the version was not yet released at the time of the project. Some students updated in the process of working on their assignment; their `frenchpress` folder contains feedback files from two different versions. No confusion resulted because the name of each feedback file indicates which version of the plug-in wrote it. For these students, Figure 5.1 reflects the higher version used on each project.

Table 5.1. FrenchPress usage by version

Version	Project3	Project4	Project5	Project6
1.2	19	7	2	3
1.3	28	12	8	5
1.4	*	30	39	11
1.5	*	*	*	25
total	47	49	49	44

5.4.3 Student ran plug-in too early

One of my reasons for building an Eclipse plug-in (Section 1.4.2) is to give the Java learner suggestions while he is working on his program, instead of delivering comments several weeks after he submits his solution for a grade. I hoped students would run the plug-in multiple times for the same `.java` file as they modified the class definition. However, my examination of the archive folder shows that running FrenchPress at an early stage of development can produce some unhelpful messages, just as Eclipse displays spurious warnings and errors when the programmer is part-way through a line of code. Rule 2 triggers in situations where the student has declared and initialized a field but has not yet written the method that modifies its value. FrenchPress might suggest that a field named `size` be declared `static final` when it seems obvious from the name that the student intends to increment or decrement `size`. Or FrenchPress recommends a `public` method be made `private` because it is not called outside the defining class, when the student has not yet completed the code for the calling class. These diagnostic messages might be confusing for the student, but they are true positives. The plug-in can only analyze the code as it exists at the moment the user selects the *Run FrenchPress* menu item.

5.4.4 Student ran plug-in on a JUnit test class

A few students tried analyzing one of the JUnit test classes provided to them by the course instructors as part of the starter code for the project. This resulted in spurious warnings from Rule 4 because all the test methods have to be `public` for JUnit to function. FrenchPress is not designed to run on JUnit test classes.

5.4.5 Unused field or method

Rule 2 triggers when the student has declared and initialized a field that is never referenced anywhere else in the program. While it is certainly possible to make such a field `static final` as Rule 2 suggests, doing so would not improve the code. The best

feedback message in this scenario would ask the student to consider eliminating the field all together. Similarly, Rule 4 recommends making a `public` method `private` if it is never called outside of the class where it is defined. But some of these methods are not called anywhere in the entire project (including test code). These might be methods the student wrote and subsequently forgot to delete when he refactored his code. Or perhaps the student included these methods because they are logically part of the class's API even though they are not required by any interface the class implements, and are not used in the current project. The feedback message for a method that is not called anywhere should be different from the feedback message for a method that is called only within its defining class.

The savvy Eclipse user can find out where a field is referenced or a method called by selecting the field or method of interest and choosing *Open Call Hierarchy* from its menu. But the advanced beginner programmers in CMPSCI 187 are not always aware of Eclipse's capabilities or motivated to exploit them. To alert these students about unused fields or methods in their code, it would be more effective to refine the diagnostics and improve the feedback of Rules 2 and 4.

5.4.6 Getter and setter methods

Among the unused `public` methods flagged by Rule 4, getter and setter methods are a special category. Students frequently define getter and setter methods whether or not these methods are called anywhere in the project. Public getter and setter methods are such a standard idiom of the Java programming language that French-Press should simply pass over these unused methods without any feedback.

5.5 Lessons learned

I realized over the course of the classroom trial that the term "flaw" can be offensive to students who are proud of their programming skills. I believe the vocabulary

of “suggesting improvements” instead of “finding flaws” would be more readily accepted by the plug-in’s intended audience. Had I chosen my words more carefully, I might have avoided the misunderstandings around survey questions and response choices that mention “flaws” (Section 5.2.3).

I should have implemented a branching survey [10, p. 213] instead of including “FrenchPress found no flaws in my program” as a possible answer. I could have programmed skip logic so that any student who answered no to question 6 (Section 5.2.2), indicating he received no substantive feedback, would have jumped directly to question 10 and bypassed followup questions about how understandable or helpful the feedback was. Skip logic would have prevented inconsistent responses to survey questions 6–9.

I should have made prompt software updates a requirement of the classroom trial. Of course it would have been better to maintain a single stable version of the plug-in throughout the duration of the study. But since bug fixes were necessary, I should at least have ensured that all participants would update their FrenchPress installation as soon as I announced a new release. Students might have been more conscientious had I told them they would lose extra credit for submitting feedback files from older versions.

I need to expand the FrenchPress usage instructions to clarify that the plug-in is not designed for JUnit test files. The user guide should also explain that running diagnostics at a very early stage of development can lead to misleading feedback from Rules 2 and 4 as described in Section 5.4.3.

CHAPTER 6

RESULTS OF CLASSROOM TRIAL

The classroom trial of the FrenchPress plug-in produced two types of data: program archives recorded in the participant's `frenchpress` folder, and responses to the online user satisfaction survey. The FrenchPress log file gives an objective record of the student's interaction with the tool, both the feedback received and the changes made or not made. The student's survey responses give insight into his subjective experience of using FrenchPress.

The Fall 2014 classroom trial of FrenchPress covered four CMPSCI 187 programming assignments, projects 3–6 in the course numbering. The percentage of students using FrenchPress who got substantive feedback (not a "Good work" message) declined steadily over the duration of the study, from a high of 87% on project 3 to a low of 30% on project 6. This might indicate students benefited from FrenchPress over the course of the semester, as they read the feedback and learned not to make the same mistakes. But many other factors could influence the level of feedback, including the type of coding required for the assignment as well as the student's (lack of) enthusiasm for running the plug-in on many different `.java` files. The program logs show that students did not always modify their code as recommended by FrenchPress. Among students who got suggestions for improvement, the proportion who changed their program in light of the feedback was about half on projects 3 and 5, but less than a quarter for projects 4 and 6. This might reflect the quality of FrenchPress diagnostic messages, which were not always appropriate for the student's program. Higher incidence of off-target messages coincided with the lower uptake on projects 4

and 6. I also examined individual trajectories of subjects who ran the plug-in on all four assignments. I wanted to see for each person whether the frequency of feedback messages from each of the diagnostic rules increased, decreased, or fluctuated over the four projects. This analysis was inconclusive because most students got feedback from a particular rule on none of their projects, or only one. Few trajectories showed any discernible trend up or down.

Survey data showed that over 87% of respondents found FrenchPress's diagnostic messages to be moderately or very easy to understand for projects 3–5, dropping to 60% on project 6. However, the diagnostics were not as helpful as they were intelligible. About 55% of users found the feedback rather or very helpful on projects 3 and 5, but the percentage drops to 32% on project 4 and 40% on project 6, tracking the rates of off-target feedback. Despite this drawback, overall satisfaction with the plug-in was good. The percentage of respondents who said they were somewhat or very satisfied with the performance of FrenchPress varied from a low of 56% on projects 3 and 6 to a high of 66% on project 5.

6.1 Data collection

Table 6.1 lists the topics of the four CMPSCI 187 assignments for which data were collected in the classroom trial. The study did not cover assignments 1 and 2, as the informed consent process had to be completed before I could acquire any student information. Students wrote their programs on their personal machines. They exported the Eclipse Java project as a `.zip` file and submitted the archive file via Moodle, the UMass course management system. I had Moodle instructor privileges for CMPSCI 187 and could download the student submissions.

For each assignment, students enrolled in the trial were asked to analyze at least one, preferably more, of their `.java` files before submitting their program to be graded. After the assignment due date, the students responded to a short online

Table 6.1. CMPSCI 187 assignment topics

Project	Problem/Data structure
P3	Towers of Hanoi
P4	Recursive linked list
P5	Binary search tree
P6	Priority queue

Table 6.2. Fluctuation in classroom trial participation

Project	Ran plug-in	Completed survey
P3	47	43
P4	49	46
P5	49	44
P6	44	43

user satisfaction survey (Appendix E) managed via the SurveyMonkey website. The number of study participants varied slightly from one assignment to the next, as a few students forgot to use the plug-in or skipped the survey for a particular project but then re-engaged on the following one (Table 6.2). A total of 53 distinct individuals tried the plug-in for at least one project. 37 students used the plug-in on all four assignments (although not all of them kept their software up to date). Some students answered the survey even though they had not run FrenchPress on their program for that assignment. I excluded those responses; Table 6.2 reflects only legitimate survey responses. The program archive and survey responses offer different vantage points from which to evaluate the impact of FrenchPress on student learning.

6.2 FrenchPress program archive

Each time the student runs FrenchPress, the plug-in updates its log file in the `frenchpress` folder of the student's Eclipse project. The program archive records all the feedback the student received, and a snapshot of his `src` folder at the time he ran the plug-in. The left-hand bar plot of Figure 6.1 shows among students who ran FrenchPress, what percentage received substantive feedback. By "substantive

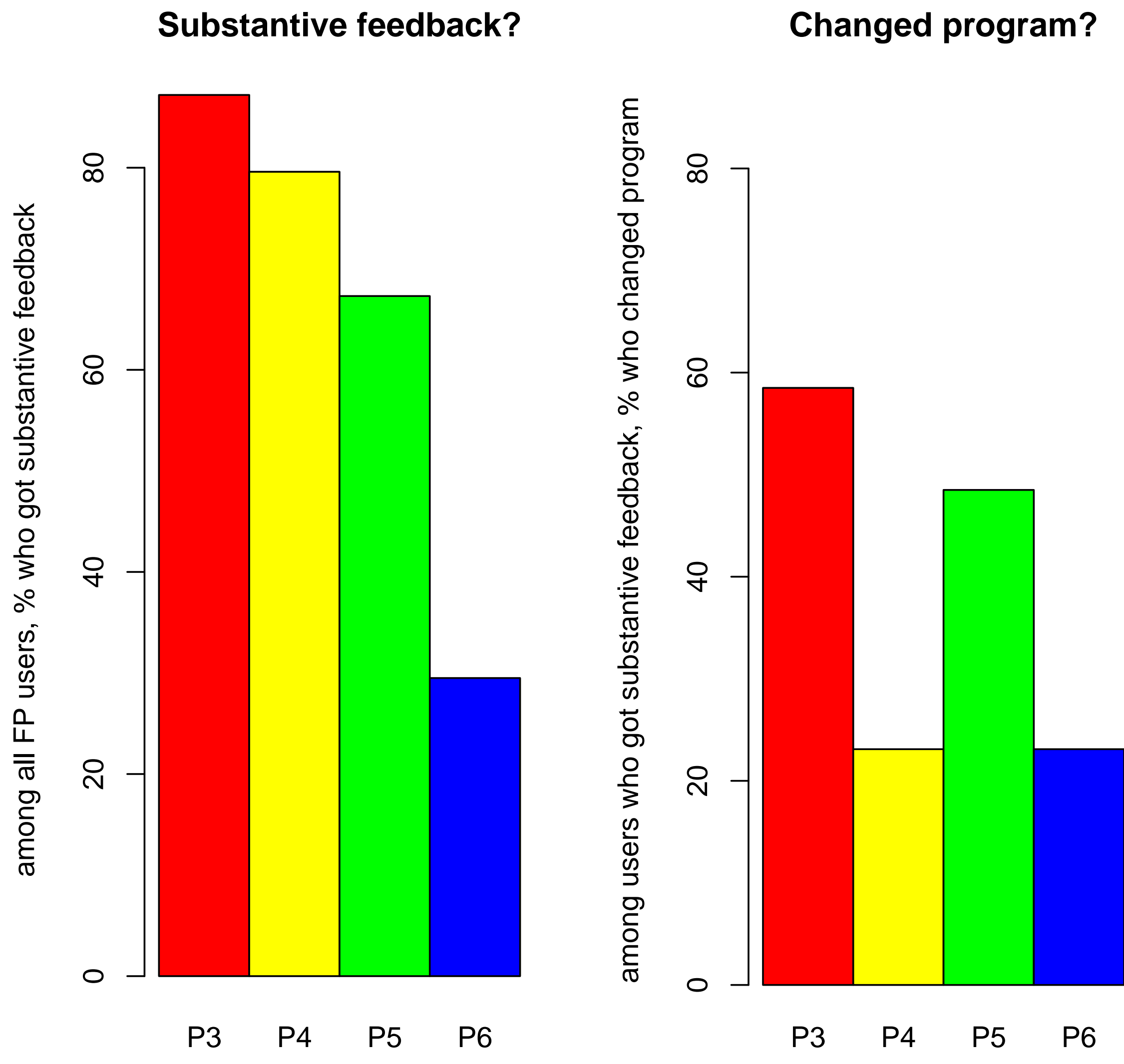

Figure 6.1. Rates of substantive feedback and program modification

feedback" I mean any diagnostic message. Users who did not get any substantive feedback saw only a "Good work" message when they ran the plug-in. The four projects are color-coded in this and subsequent figures. The percentage of users receiving substantive feedback declined from a high of 87.2% on the first project to a low of 29.5% on the last project. Since each student could decide for himself how many of his `.java` files he wanted to examine with FrenchPress, the amount of diagnostic feedback to which the student was exposed depended in part on how zealous he was in running the plug-in. The minimum required to earn extra credit was running FrenchPress on just one `.java` file for each project. I deliberately kept the criteria of participation low to avoid seeing students drop out of the study because they found it too time-consuming.

Regardless of student behavior, project 6 elicited little diagnostic feedback compared to the other assignments. Table 6.3 shows the percentage of substantive feedback messages among all the messages generated by the plug-in on each assignment. Project 6 seems to be qualitatively different from the other three projects. The low rate of substantive feedback on the final project might reflect improvement in student programming practices, due to exposure to FrenchPress, the excellent instruction of Profs. Barrington and Corner, and the coding experience gained over the semester. However, the drop in substantive feedback from project 5 to project 6 is so steep that I suspect other factors are also contributing. The data structure explored in each assignment or the starter code furnished by the professors could render FrenchPress diagnostics more or less relevant. One possible explanation is that project 6 had some symmetrical structure because it asked students to implement a max- and a min-priority queue using a max- and a min-comparator. When one of the pair was working correctly, the student could just copy his code and make minor modifications to finish the mirror class. Less room for creativity means lower likelihood of mistakes, hence less FrenchPress feedback.

Table 6.3. Substantive feedback varies by project

Project	Total msg	Subst msg	% Subst
P3	376	135	36%
P4	141	78	55%
P5	262	87	33%
P6	232	19	8%

Table 6.4. Recap of FrenchPress diagnostics

Rule 1	Field could have been a local variable
Rule 2	Instance variable could have been a static final constant
Rule 3	Instance variable declared public
Rule 4	Non-static method declared public
Rule 5	Integer variable used as a boolean flag
Rule 6	Redundant boolean expressions

6.2.1 Recap of diagnostic rules

For convenience, Table 6.4 recaps the six rules that produced diagnostic messages for students in the classroom trial. Rule 7 *Inappropriate for loop control* did not trigger on any student program. This could possibly be explained by an upward trend in the level of the undergraduates who get admitted to the UMass computer science major over the time it took to propose and develop FrenchPress. It is equally possible that the courses students take before they arrive in CMPSCI 187 are doing a better job of preparing them on this point.

6.2.2 Short-term indicator of student learning

The right-hand bar plot of Figure 6.1 shows what percentage of students who received substantive feedback changed their program in response to the diagnostic message. This measure can be taken as an indicator of student learning in the short term: the student read the feedback message, understood it, and acted upon it. The indicator ranges from 58.5% on the first assignment in the study to 23.1% on the final assignment, but it does not show a smooth decrease in between. The sharp decline

Table 6.5. Off-target feedback varies by project

Project	Subst msg	Off-target msg	% Off-target
P3	133	62	47%
P4	80	57	71%
P5	87	26	30%
P6	19	11	58%

from project 1 to project 2 might reflect the fact that the students' initial enthusiasm for new software has worn off. By the end of the classroom trial, fewer participants were getting substantive feedback from FrenchPress but a good proportion of those who got feedback were still motivated to modify their code. One reason for the variation in student uptake of FrenchPress's suggestions is that the diagnostic messages are not always on target. For reasons detailed in Section 5.4, Rules 2 and 4 generated many messages that were not entirely appropriate for the student's program. Some of these messages were false positives due to implementation errors, while other messages were misleading because the rule itself needed further refinement. Rule 4 was the most prolific of all the diagnostic rules and also produced a large proportion of the off-target messages. Table 6.5 shows the off-target feedback as a percentage of all the substantive feedback FrenchPress doled out. Projects 4 and 6 have higher percentages of off-target messages and lower percentages of students who changed their programs in light of the feedback. It is quite possible that the students were exercising good judgment in ignoring messages that did not make sense for their code.

6.2.3 Longer-term indicator of student learning

If FrenchPress is achieving its educational goals, one would expect the frequency of feedback to decline over time as students learn to avoid the mistakes they were making when they first started using the plug-in. To look for trends over the length of the classroom trial, I limited my attention to the 37 participants who used the plug-in

on all four projects. I tabulated each rule separately because there is no reason to expect that what a student learns from one rule will carry over to other rules. The student might see, and learn from, an instance of Rule 3 on the first project but encounter his first message from Rule 2 on the last project. Although the six rules fall into three categories (as discussed in Chapter 3), even rules of the same category address different stylistic issues; seeing feedback from one would not necessarily help the student avoid triggering the other.

The concept of frequency is understood in relation to the length of the code analyzed by FrenchPress. For each student and each project, the feedback frequency for Rule n is

$$\frac{\text{number of feedback messages generated by Rule } n}{\text{total file length}}$$

The denominator includes all files the student analyzed for that project, whether or not they produced substantive feedback. Each of the 37 students has four feedback frequency numbers for each diagnostic rule, corresponding to the four projects covered by the classroom trial. These four numbers reflect the trajectory of the student through the classroom trial with respect to that rule. His trajectory falls into one of the following categories:

0msg the student received no feedback messages from that rule on any project;

1prj the student received feedback messages from that rule on exactly one of the four projects;

Dec the feedback frequency for that rule follows a decreasing trajectory;

Inc the feedback frequency for that rule follows an increasing trajectory;

Zig the feedback frequency for that rule follows a zigzag trajectory (down followed by up, or vice versa).

Figures 6.2–6.4 show for each rule the distribution of student trajectories in these five categories. I separated out trajectories in which the student received feedback from the rule on only one of the four projects because it is difficult to interpret the significance of the zeros that precede and follow the single positive feedback frequency. Do these zeros reflect student understanding of the concept addressed by the rule, or do they reflect the various types of code required by the different projects? Learning from feedback is only one factor that might influence student trajectories through the classroom trial. The nature of the problem posed in each assignment makes some projects more likely to trigger certain rules and not others. Project 6 produced very little substantive feedback, so almost all trajectories will show a decline at the end of the semester. This seems more a consequence of the project than of any learning that might have occurred.

The evidence for student learning from these plots is inconclusive. For all the rules except Rule 4, so many student trajectories are in the **0msg** or **1prj** categories that the remaining trajectories do not show much of a trend. Rule 4 is the only one that has a significant percentage of decreasing trajectories. This rule generated more feedback messages than any other, as may be seen from the low percentage of trajectories in the **0msg** category. Rule 4 also contributed a large proportion of off-target feedback, which might account for the high percentage in the **Zig** category.

A classroom trial of only four programming assignments is not long enough to see slowly developing trends. The data collected so far are not adequate to draw a firm conclusion about the efficacy (or lack thereof) of the plug-in. The type of learning one might expect from a tool such as FrenchPress will probably not be evident until the semester following the one in which the student is exposed to the feedback, or even later in the student's course sequence. FrenchPress tries to give students a gentle push toward better programming practices. A student might read a feedback message and decide it is not worth the trouble to modify a program that is already

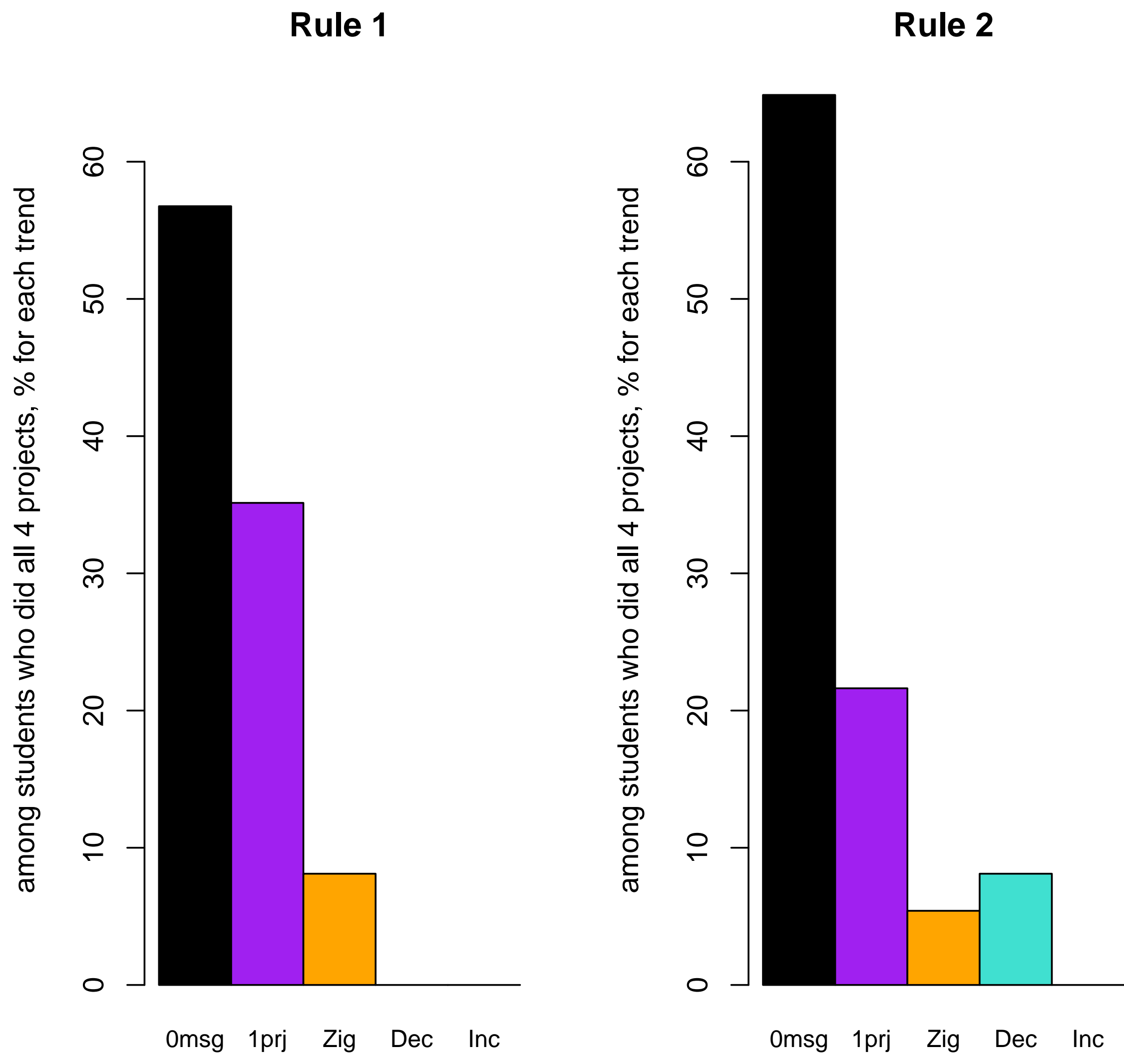

Figure 6.2. Rules 1 and 2 feedback trends

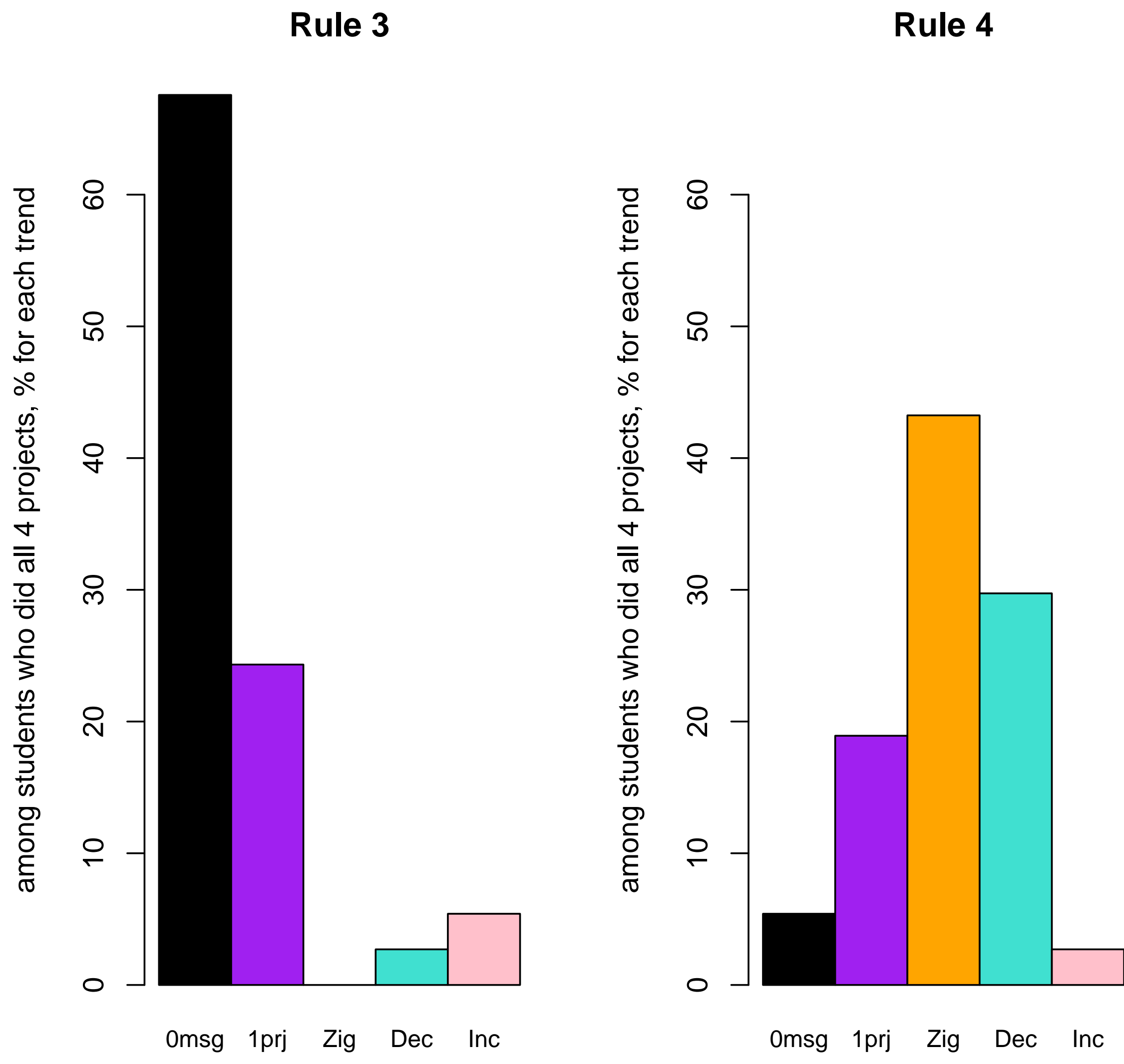

Figure 6.3. Rules 3 and 4 feedback trends

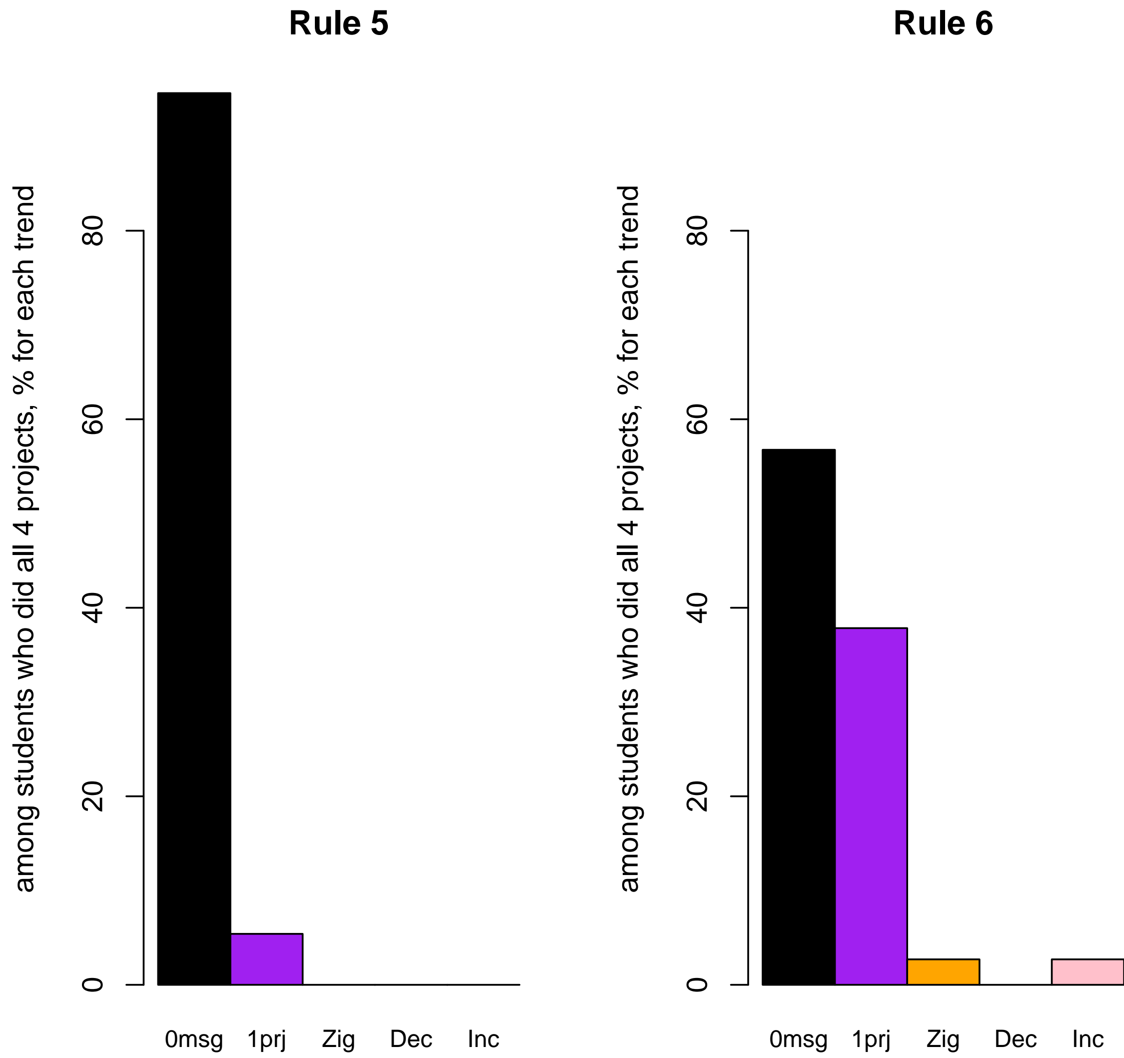

Figure 6.4. Rules 5 and 6 feedback trends

computing the correct output for the assignment. He might still internalize the coding recommendation and adhere to it on subsequent projects. One would have to conduct a longer study to discern the gradual evolution of student programming habits.

6.3 User satisfaction survey responses

An important objective of this research is to give advanced beginner programmers feedback they can understand, eschewing unfamiliar jargon and subtle programming language concepts these students have not yet absorbed. To get a sense of how well FrenchPress achieved this goal, I included several questions on the user satisfaction survey (Appendix E) related to the quality of feedback and the student's overall impression of the tool.

> For this assignment, was the feedback from FrenchPress confusing or easy to understand?
>
> For this assignment, was the feedback from FrenchPress helpful or unhelpful?
>
> Are you satisfied or dissatisfied with the performance of FrenchPress on this assignment?

Figures 6.5–6.7 show the distribution of answers to these questions for the four projects. I standardized the three Likert-style scales so that each bar plot runs from *Bad* (confusing, unhelpful, dissatisfied) on the left to *Good* (easy to understand, helpful, satisfied) on the right. The labels *MB* and *MG* stand for moderately bad, moderately good. In Figures 6.5 and 6.6, the y-axis includes only those students who indicated on their survey that they had received substantive feedback (*sfb*) for that project. These survey questions had a separate response choice ("FrenchPress found no flaws in my program") for students who got only "Good work" messages on the project. In Figure 6.7 the y-axis includes all survey respondents for each project.

These bar plots suggest that most students found the feedback easy to understand, but not consistently helpful. Project 6 feedback seems worse than the others, but the

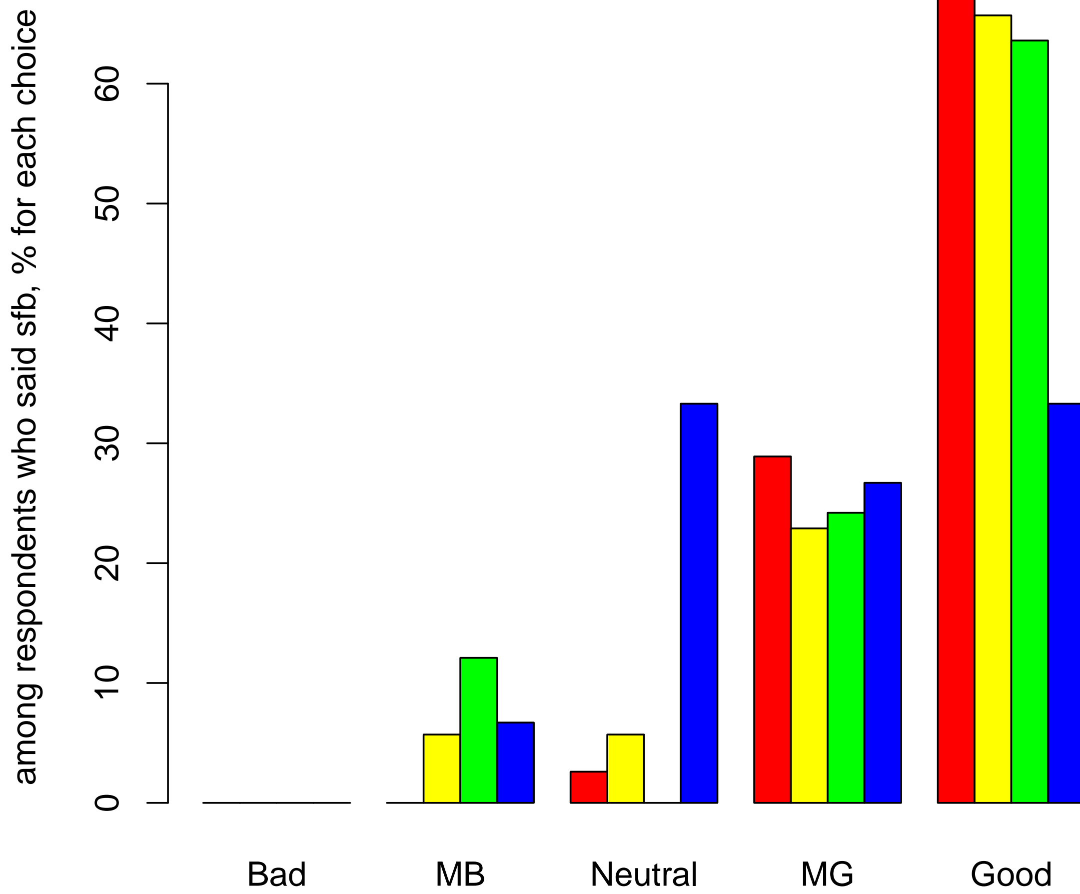

Figure 6.5. Is feedback confusing or easy to understand?

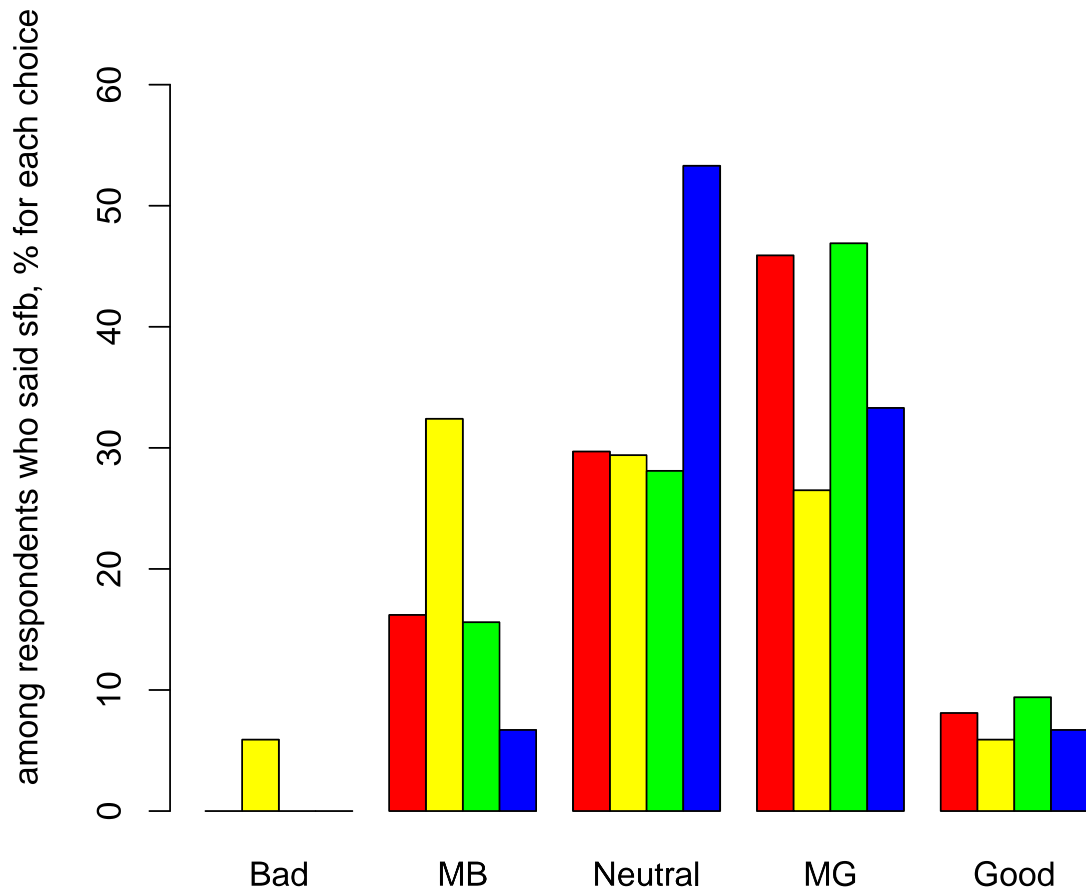

Figure 6.6. Is feedback helpful or unhelpful?

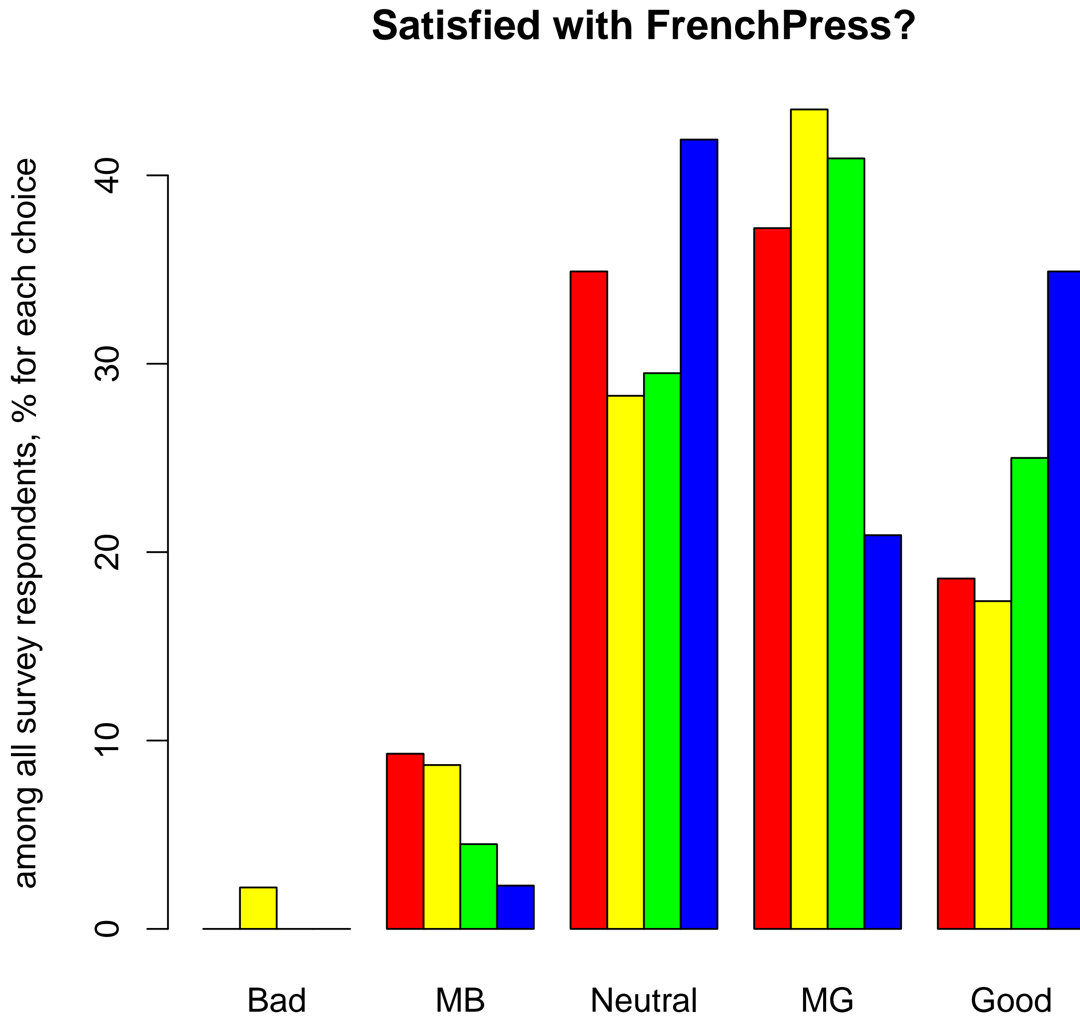

Figure 6.7. Overall satisfaction with FrenchPress

difference might be due to population size: only 13 students got substantive feedback on project 6 (19 messages in total). As noted in Section 5.4, the plug-in's performance was marred by false positives and rules that triggered in situations for which the feedback message was not correctly worded. These shortcomings are reflected in Figures 6.6 and 6.7. Nonetheless, most students rated their overall satisfaction with the plug-in as neutral or positive.

6.4 Selected student comments

Many study participants wrote thoughtful comments in response to the last question on the survey,

> How can we improve FrenchPress?

Some students feel FrenchPress's diagnostics are not relevant for their coursework.

> FrenchPress didn't find any big flaws in program. It might be more useful when I have to write more and bigger classes, so that way FrenchPress can catch mistakes and flaws that I might not have seen before. I just found it rather unhelpful in a small programming assignment such as this one.

> Right now, I just don't think FrenchPress is relevant to the material we are covering. The coding and "flaws" that FrenchPress found within my program aren't really "flaws", e.g. I had a method that was public but wasnt called outside of a class so FrenchPress suggested that I make it private. I guess that's a flaw but it really doesn't help when you're a student just trying to learn how to code. I can see how it would definitely be useful in other contexts though. The thing's we're covering right now are pretty basic and I think FrenchPress refines code rather than "fixes".

Some students had unrealistic expectations of what kind of diagnostics they could expect from FrenchPress. The 15 minutes of lecture time I had to introduce the classroom trial were not sufficient to clarify the scope of the plug-in while also explaining the purpose of an Informed Consent Form (Appendix D).

> Would be better if it can find logical errors in my program

> It finds no flaws while there is logical errors in my program.
>
> This is probably very difficult but I was having a hard time figuring out the Big-O Notation. If FrenchPress could actually determine the performance of codes, that would be amazing. Just a suggestion, something to optimize codes would be a really good tool.

Some students acknowledged they had learned something from using the plug-in.

> Although FrenchPress has really helped me to understand local variables and other such things, I suppose it would be nice to get some feedback on how my code is written in terms of clarity and style perhaps? Nevertheless, it has been working very well.
>
> Using FrenchPress has really helped me improve the way I initially code. Because I now have fewer flaws, it seems harder for me to receive feedback. Nevertheless, there isn't anything I can think of that FrenchPress really needs.

Some comments addressed the problems with Rule 4.

> It tells me that I need to change some "helper" method to private, but they are not really helper methods.
>
> Sometimes French Press detected "flaws" in my programs, but is not really a flaws. For example, in my LlinkList class, even though the setInfo is never call outside of class in this project, It can be call on other project. I hope FrenchPress can detected flaws depend on the method's general use no just base on one project.

Rule 6 is more controversial than I anticipated.

> I find that the phrase if(box.isEmpty() == false) is more readable than if(!box.isEmpty). I would get rid of the corrections to remove this. It seems like its mostly an issue of preference and I don't know that I've heard a convincing argument that the == false is worse in any way. Perhaps it's trivially slower, but that's not the statement made in the report.

Some students asked for a better user interface.

> Put the suggestions in tips on each line which are done automatically rather than running it manually.
>
> Although it is very clear where the error is, FrenchPress could direct you to and highlight the the error.

> It would be cool if like how the compiler underlines syntax errors with red, if FrenchPress could underline the sections in a certain color after running it. I don't necessarily know what that entails for eclipse or if its possible, but would both be visually appealing and also informative in marking the errors.

Some students want FrenchPress to reformat their code, apparently unaware that Eclipse already provides this functionality.

> Perhaps it can fix formatting in the code to make the format look tidier

> I suggest that it be able to fix the formatting of the code to make it be more presentable and more professional.

Some students are ready for an expanded set of diagnostics.

> It runs smoothly, but I still feel like there should be more features, unless I'm doing something wrong.

> not smart enough

> Checking for more problems, and explanations on why existing ones are bad practice.

> Perhaps include more flaws, such as suggesting one break up large methods into smaller sub-methods.

For some students, no news is good news.

> For this project, I did not find any flaws in my programs. So it very good.

CHAPTER 7

FUTURE WORK

Future work on the FrenchPress plug-in falls into three areas: better user experience for students, new features for course instructors, and improved diagnostics. Students need more detailed explanations and a better visual display of feedback. Professors would benefit from summary statistics and more control over how FrenchPress applies its rules. Rules that now generate off-target feedback must be refined to give users appropriate guidance. A longer classroom trial is essential to gather more conclusive evidence of FrenchPress's effect on student learning.

7.1 Student experience of FrenchPress

The feedback FrenchPress currently offers is rudimentary. For the simpler rules such as Rule 6 *Redundant boolean expressions*, perhaps the two- or three-sentence message FrenchPress now displays is adequate. More subtle rules such as Rule 1 *Field could have been a local variable* or Rule 5 *Integer variable used as a boolean flag* demand more explanation. The next phase of FrenchPress development will provide a two-tier system of feedback. In addition to the short message FrenchPress now delivers, students will have the option to get more information by clicking a button. The button will open a window containing a more detailed writeup including an example of the flaw FrenchPress identified and how to fix it.

The FrenchPress prototype displays feedback in a dialog box that closes as soon as the student clicks the OK button. If the student wants to keep the feedback visible while he modifies his code, he has to open the feedback file stored in the `frenchpress`

folder of his project. It would be easier for the student to track FrenchPress diagnostics if the plug-in exploited Eclipse interface mechanisms such as warning symbols in the margin of the editor window, highlighting, and hover text. For example, the student could mouse over a method definition or variable declaration and FrenchPress would display the relevant feedback message (if any).

Most of the code changes FrenchPress suggests to the user will not affect the behavior of the program. However, if the student heeds feedback from Rule 3 *Instance variable declared public* and changes the variable in question to `private`, this will cause compiler errors if the public instance variable is accessed in other classes. The student could get frustrated when he sees that FrenchPress's advice led him to new compiler errors. The appropriate solution is getter and setter methods. FrenchPress should include in the feedback message a list of the direct variable accesses that must be replaced by a call to a getter or setter method.

7.2 New features for professors

I have ordered FrenchPress's diagnostic rules to reflect my judgment of their relative importance for the student's understanding of Java. Another instructor may have a different opinion about the best way to order multiple feedback messages. In a future version of the plug-in I would like to give the course instructor the power to change the order of the rules or to disable rules she does not want her students to see.

FrenchPress feedback could be helpful not only for the students but also for their professor. Once the students have submitted their assignments, the course instructor might want to know which programming issues were highlighted by FrenchPress. If she realizes that many students are making similar types of mistakes, the professor can discuss in class the misconceptions that led to those mistakes. The plug-in could be repackaged as a stand-alone application that would run in batch mode over a directory

of the entire class's homework submissions. This version of FrenchPress would create a summary report showing the distribution of various diagnostic categories. The instructor could then address some of the most frequent or most egregious stylistic errors in her lecture.

7.3 Improved diagnostics

The rule that triggered most frequently in the classroom trial is Rule 4 *Non-static method declared public*. This rule is intended to identify a method declared `public` that is never called outside of the class where it is defined and hence does not have to be `public`. As described in Section 5.4.5, many of Rule 4's feedback messages are not on point because in fact the method is not called anywhere, in the defining class or outside of that class. Students sometimes define getter and setter methods that are not needed in the program, but they include the methods anyway for completeness. There might also be methods that are not called anywhere because the student changed his mind as the program evolved and forgot to delete code that had become useless. I need to refine Rule 4 to distinguish between methods that are called only within the class where they are defined, and methods that are not called at all. Likewise Rule 2 *Instance variable could have been a static final constant* should be split in two to create separate diagnostics for a field that is declared (and possibly initialized) but never referenced anywhere else in the program. In most cases these unused fields are the residue of an abandoned design and should be eliminated.

I could expand the scope of FrenchPress by adding completely new diagnostic rules. These might include the rules discussed in Section 1.7 to look for over-ambitious constructors and inappropriate inheritance relationships between classes. I might also invest the energy to first clarify and then implement the fuzzy ideas for diagnostics listed in the introduction to Chapter 3.

7.4 Extended classroom trial

The classroom trial conducted in Fall 2014 was too short to yield any conclusive evidence that students had learned better programming practices from using FrenchPress. A longer study covering more programming assignments would be required to substantiate claims that the plug-in offers a real benefit for students. If I were planning a new classroom trial I would make two significant changes:

- identify a "control" group of CMPSCI 187 students who signed a consent form but have not run FrenchPress;
- look at the possible correlation of student grades and diagnostic feedback.

I would like to identify a group of students within the study population who are not using the plug-in so I could compare their programs to those of students who have tried FrenchPress. In both of the classroom trials already conducted in CMPSCI 187 (Spring and Fall, 2014), some students signed an Informed Consent Form but never took any further steps to fulfill the requirements for extra credit. I could not tell whether these students still considered themselves to be in the trial, since paragraph 11 *Can I Stop Being In The Study?* of the Informed Consent Form (Appendix D, page 125) says anyone is free to drop out of the study whenever he wishes. The Informed Consent Form grants the researcher authorization to examine programs submitted by study participants, but were these students still participating?

To resolve this confusion, I would revise the Informed Consent Form to require an explicit opt-out by email from subjects who initially sign their form but subsequently decide to leave the study. I would expect to see a group of students who sign a consent form, lose interest and never run the plug-in, but do not communicate any intention to drop out of the trial. I will refer to these subjects as "inactive participants" to distinguish them from the "active participants" who run the plug-in and take the user satisfaction survey after every assignment. I could treat the inactive participants as

an informal “control” group vis-à-vis the “treatment” group of active participants. Of course this would not be a true controlled trial because students self-select for one group or the other. I could run FrenchPress on programs submitted by the “control” group to find out whether the categories and frequency of diagnostic feedback differ between the active and inactive participants.

Exposure to FrenchPress does not promise to raise a student’s grades in CMPSCI 187 or similar programming courses. Most data structures and algorithms courses with large class sizes evaluate student code on the basis of input/output behavior and adherence to the specifications of the assignment. FrenchPress aims to improve the advanced beginner’s programming style, but is not designed to have any direct effect on the factors that determine his grade. Nevertheless, it could be revealing to examine the correlations among programming grades, FrenchPress usage, and feedback received for students who participate in the classroom trial. Do weaker students run the plug-in more often because they value the extra help? Or do stronger students run the plug-in more often because they are generally more motivated to take advantage of learning opportunities? Do weaker students get more feedback messages per line of code than stronger students? Do they get different types of feedback (from different rules) than stronger students? Gaining access to student grade information, even if aggregated over groups of study subjects, would require revision of the Informed Consent Form and compliance with FERPA regulations.

7.5 Conclusion

Many existing automated assessment systems are designed to help students get through their first Java course, as they are struggling with the mechanics of the language. At UMass, students in the introductory Java class rely on an interactive online textbook described in Section 2.1. When students graduate to the next level of instruction they outgrow these tools because they have made, and learned from, most

of the novice's mistakes. Yet students in their second or third Java course are not ready for professional strength diagnostics from FindBugs and comparable program analysis systems. The errors detected and the explanations offered fly over the head of the inexperienced programmer. I developed FrenchPress for the population of advanced beginners in Java who are now dependent on their instructors and teaching assistants for helpful feedback on their programs.

Implemented as an Eclipse plug-in, FrenchPress can be readily incorporated into many different undergraduate programming courses. Researchers at two major conferences in computer science education[1] have expressed interest in deploying FrenchPress in their own classrooms. The system will support student learning in any educational environment, but particularly those in which the teaching staff have difficulty providing individualized attention to all the students. These include community colleges, where instructors have no teaching assistants, and public universities, where large class sizes outstrip limited personnel resources. Automated feedback will also facilitate distance learning: the student can get guidance on his program any time and anywhere he needs it.

[1] *SIGCSE 2015, 46th ACM Technical Symposium on Computer Science Education*, 5–7 March, Kansas City, MO; *ITiCSE 2015, the 20th Annual Conference on Innovation and Technology in Computer Science Education*, 6–8 July, Vilnius, Lithuania.

APPENDIX A

CMPSCI 187 ASSIGNMENTS

A.1 Spring 2008

This assignment is a bit of a reality check. It isn't very hard, but it's also not trivial. It's designed to give you a sense of the Java skills I expect you to have as you enter the course.

The so-called **birthday paradox** is the observation in basic probability theory that if 23 people are in a room, the chances are about 50–50 that two people have the same birthday. Your job for this assignment is to verify this claim. More specifically, consider rooms with a variety of people (10 up to 30 people), and in each case, run 10000 experiments to determine the approximate likelihood that two people have the same birthday. Here is sample output from my implementation; it shows, for example, that with 10 people in a room, people have the same birthday 11.43% of the time. (In other words, I ran 10000 trials with 10 in the room, and a common birthday showed up in 1143 of those trials.)

```
> java BirthdayDriver
people: 10 0.1143
people: 11 0.1414
people: 12 0.1737
people: 13 0.1851
people: 14 0.2247
people: 15 0.2487
people: 16 0.2797
```

```
people: 17 0.3153

people: 18 0.3373

people: 19 0.3827

people: 20 0.4097

people: 21 0.4544

people: 22 0.4671

people: 23 0.505

people: 24 0.5347

people: 25 0.5751

people: 26 0.6023

people: 27 0.6226

people: 28 0.6583

people: 29 0.6836

people: 30 0.7037
```

Write a two class application that computes these values.

Robbie Moll – 29 Jan 2008

A.2 Fall 2011

CMPSCI 187: Programming With Data Structures

David Mix Barrington

Fall, 2011

Programming Project #1: Mazes and Cells

Originally posted 8 September 2011, due at 11:59 p.m. EDT on Monday 19 September 2011, by placing .java files in your cs187 directory on your edlab account. For more information on accessing the edlab (Question P1.2), and answers to other questions on this assignment, see the Q&A page.

Goals of this project:

1. Submit a compilable and correct program to us through the EdLab.

2. Write a program using objects and classes.

3. Begin the code base for later projects involving mazes.

4. Learn (or review) arrays, including two-dimensional arrays.

Many computer games, such as Sid Meier's *Civilization* series, involve pieces moving on a square grid of cells. In this project you will write a Maze class, allowing you to create Maze objects that are rectangular arrays of Cell objects. You will also write the Cell class. In the future we will extend the Cell class to make it more interesting, but for this project a Cell has only three fields: `int x` and `int y` giving its position in its Maze, and `boolean open` telling whether it is open to be moved into.

A Maze has three fields in all. The first two are `int width` and `int height`, giving the number of columns and the number of rows respectively, and the third field is a two-dimensional array of Cell objects.

Each class should have the usual get and set methods, a toString method, and constructors as specified below. The toString method for Cell gives a string such as `(2, 3) open` if the Cell is at position x = 2 and y = 3 and is open, or `(2, 3) closed` if it is in that position and not open. The toString method for a Maze of width w and height h is a sequence of h binary strings, separated by line breaks, where each individual string has length w. Open Cells are represented by ones and closed ones by zeros. For example, if m is a Maze of width 4 and height 3, where exactly those Cells on the boundary are open, then `m.toString()` would return the String `"1111\n1001\n1111"`, which is printed out as:

```
1111
1001
1111
```

The Cell class should have the following two constructors:

- `Cell (int x, int y)` — gives an open Cell with those values for x and y
- `Cell (int x, int y, boolean isOpen)` — gives a Cell with those x and y values, open or not according to isOpen

The Maze class should have the following two constructors:

- `Maze (int w, int h)` — gives a Maze of width w and height h with a Cell in each place, the correct x and y for each Cell, and all cells open
- `Maze (int w, int h, String [ ] init)` — gives a Maze of width w and height h with a Cell in each place as above, but the openness of the Cells is specified by the array `init`, which should be an array of h binary strings, each of length w. For example, we would create the Maze above by

  ```
  String [ ] s = {"1111", "1001", "1111"};
  Maze m = new Maze (4, 3, s);
  ```

The final part of the assignment (necessary to get an A) is to add an instance method `moves` to the Maze class. This method takes two int arguments and returns an array of Cells, so its signature is `Cell [ ] moves (int col, int row)`. The array returned contains from zero to four Cells, and these Cells are to be exactly the *open* Cells in the Maze that can be reached by one move up, down, right, or left from the Cell at (col, row). The method should throw an `ArrayIndexOutOfBoundsException` if (col, row) is a position that does not exist in the Maze. But it should not throw an exception in any other case — this means that you will have to be careful when (col, row) is on the boundary of the Maze.

If m is our example Maze above, the call `m.moves(0, 0)` should return an array of two Cells, the first one having toString "(1, 0) open" and the second one having toString "(0, 1) open". If we call `m.moves(1, 2)` we again get an array of two Cells, which have toStrings "(0, 2) open" and "(2, 2) open". We don't get the Cell at (1,

1) because it is closed and we don't get a Cell at (1, 3) because that is outside the Maze.

Last modified 15 September 2011

A.3 Fall 2013

CMPSCI187-SEC01 Programming w-Data Structures Fall 2013

P0 description

This is the first programming assignment. It is a simple assignment designed to help you get you back in the spirit of Java and to let you practice submitting an assignment for grading. This assignment is worth only a few points, but it is very important that you submit it so that your submission setup can be tested.

This assignment is worth 15 points. You will be allowed to submit it for grading as often as you like before the cutoff date. Submissions after the due date but before the cut-off date will be penalized per class policy.

Summary

For this assignment, write a program that reads lines from standard input (not from a pop-up window) until a line that contains just "end" is encountered. Each line will be a URL, though for this assignment you do not care what the line contains (you will for P1).

After you have encountered the "end" line, print the total number of lines that were encountered, but not including that "end" line. The format of your output must be exactly:

```
>> Got 10 lines
```

The two "greater than" characters must be the first two characters on the line, then a space, then the word "Got", then the number of lines found, and then the word "lines" (if there was only one line, make it singular).

What to submit

Your program should be entirely contained in a file called P0.java and that is the only file you should submit.

To submit, log onto the Edlab computers and change directory to your “cs187” directory. Create a directory called “P0” and put your program there — that is, the program should be in /cs187/P0/P0.java.

Then type the command *submit P0* (an uppercase P) and watch your program be submitted and graded. If you don’t like your grade, you may revise the program, transfer the new version to the Edlab computers, and submit it. You may do that as often as you like for P0; for future assignments, the number and frequency of submissions will be restricted.

Tips

There are several ways to read input from standard input. An easy one is the java.util.Scanner class. The class is described in your textbook on pages 764–5 (in Appendix E). Section 1.3 of your textbook has example code called Days-Between.java that uses Scanner — though it uses it to read numbers rather than complete lines. You can also find lots of detail about Scanner online at places like http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Scanner.html.

Last modified: Thursday, September 5, 2013, 4:34 PM

A.4 Spring 2014

CMPSCI 187 — Spring 2014

Assignment 1

The goal of this assignment is to introduce you to Eclipse and the standard procedure you will use for completing a programming assignment, testing your code, and turning it in through Moodle. In this assignment (and others) you will be starting from some initial code provided by us and then modifying it. Since you will repeat a

similar process for each programming assignment, please follow these steps and make sure you understand each one:

1. Install the Eclipse development environment.

The version to download and install is: **Eclipse Standard 4.3.1** available for multiple platforms at:

http://www.eclipse.org/downloads/packages/eclipse-standard-431/keplersr1

If you already have a version of Eclipse installed, please make sure you upgrade to the latest version. This version of Eclipse will be available on the edlab machines as well.

2. Download the starter code.

Download the provided archive file containing the starter code ("Assignment1.zip") for this assignment from Moodle and save it somewhere where you can find it. You **do not** need to unzip it — Eclipse will handle that for you.

3. Import the code into Eclipse.

A. Open the Eclipse application.

You will be asked to specify a location for your Eclipse workspace. This is a directory on your system where all of your Eclipse projects can be stored. You can use one workspace throughout the entire class, and this is where you will set it up. If you are opening Eclipse for the first time, you may see a welcome screen with links to instructions, tutorials, etc. You are welcome to read these, but eventually you should click on the upper right: there is an arrow that will take you to the "Workbench," the standard view for working in Eclipse.

B. Import the starter code

Choose **File** → **Import** from the menu. A window will come up so you can choose

how to import. Select "General" and within that, "Existing Projects into Workspace". (It may seem strange, but **do not** choose "Archive File"). Then click "Next". Choose the button for "Select archive file" and locate the file you downloaded in step (2) above. Then click "Finish". You should see a **dates** java project in the package explorer window on the left. Click on the triangles to reveal the content of this directory and the src and test directories within it. You will see three java source files among the directories:

Date.java DaysBetween.java DateTest.java

4. Run the starter code

First explore the code as it is. Under src, choose the DaysBetween.java file, then click the green play button, or choose "Run" from the Run menu. A console will appear in the bottom of the Eclipse window. Enter two dates and witness the output. Chapter 1 of the textbook includes a complete description of this code.

5. Test the starter code

Choose DateTest.java in the Package Explorer and run it using the play button or the menu, as above. The package explorer on the left will switch to a JUnit pane, which will show the testing output. You should see a total of 6 tests: 5 tests that pass, and one test that fails. Familiarize yourself with the testing interface. If you select the failed test, you should see the following under Failure Trace:

```
java.lang.AssertionError: Day index of week not correct
expected:<3> but was:<0>
```

(You may have to resize the JUnit pane to see the full message.)

6. Correct the starter code

The failed test indicates a problem with the starter code that you need to fix. Your goal should be to correct the implementation of the **indexDayOfWeek()** method

in Date.java. If you read the testing code, you will find that the failed test checks whether indexDayofWeek returns the right integer for the date of 12/19/1973. The test failure trace shows that 3 was expected, but the function returned 0. The correct solution is not merely to make the function return 3 on the date 12/19/1973. Instead, you should revise the code so that the function will return the right integer for **any** date. When we grade your program, we will test it on other dates to make sure it works properly.

7. Export your completed code

When you have completed the changes to your code, you should export an archive file containing the entire java project. To do this, click on the **dates** project in the package explorer. Then choose **File** → **Export** from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file.

8. Submit your code using Moodle.

Login to Moodle and, on the page for Assignment 1, upload the archive you exported above.

APPENDIX B

STUDENT INTERACTION WITH FRENCHPRESS

This appendix reproduces one student's interaction with the FrenchPress plug-in. The initial version of the class definition (Section B.1) triggered Rules 1 *Field could have been a local variable* and 5 *Integer variable used as a boolean flag* (Section B.2). The student modified his code as suggested by Rule 5 (Section B.3) and ran the plug-in again. FrenchPress then reported a case of Rule 6 *Redundant boolean expressions* (Section B.4), which the student fixed (Section B.5). The final run of the plug-in (Section B.6) indicates that the Rule 1 flaw remains unchanged.

B.1 Initial version

```
1   package structure;
2
3   public class RecursiveList<T> implements ListInterface<T> {
4
5      int size;
6      LLNode<T> first, last, temp;
7      T info;
8      int loc = 0;
9      int prime = 0;
10
11     @Override
12     public int size() {
```

```
13        return size;
14     }
15
16     @Override
17     public ListInterface<T> insertFirst(T elem) {
18        if (elem == null)
19           throw new NullPointerException();
20        LLNode<T> newNode = new LLNode<T>(elem);
21        if (first == null)
22           first = last = newNode;
23        else{
24           newNode.setLink(first);
25           first = newNode;
26        }
27        size++;
28        temp = first;
29
30        return this;
31     }
32
33     @Override
34     public ListInterface<T> insertLast(T elem) {
35        if (elem == null)
36           throw new NullPointerException();
37        LLNode<T> newNode = new LLNode<T>(elem);
38        if (last == null)
39           first = newNode;
```

```
40          else
41             last.setLink(newNode);
42             last = newNode;
43          size++;
44          temp = first;
45
46          return this;
47       }
48
49       @Override
50       public ListInterface<T> insertAt(int index, T elem) {
51          if (index < 0 || index > size)
52             throw new IndexOutOfBoundsException();
53          if (index == 0)
54             insertFirst(elem);
55          else if (index == size)
56             insertLast(elem);
57          else if (size == 2)
58             {
59                first.setLink(new LLNode<T>(elem));
60                first.getLink().setLink(last);
61                temp = first;
62                size++;
63             }else if (size > 2 && index > 1){
64                temp = temp.getLink();
65                insertAt(index-1,elem);
66             }else{
```

```
67                LLNode<T> newNode = new LLNode<T>(elem);
68                newNode.setLink(last);
69                temp.getLink().setLink(newNode);
70                temp = first;
71                size++;
72             }
73          return this;
74       }
75
76       @Override
77       public T removeFirst() {
78          if (isEmpty())
79             throw new IllegalStateException();
80          T info = first.getInfo();
81          first = first.getLink();
82          size--;
83          temp = first;
84          return info;
85       }
86
87       @Override
88       public T removeLast() {
89          if (isEmpty())
90             throw new IllegalStateException();
91          if (size <= 1){
92             info = first.getInfo();
93             temp = first = last = null;
```

```
94              size--;}
95           else if(size==2){
96              info = first.getLink().getInfo();
97              first.setLink(null);
98              temp = first;
99              size--;
100          }else{
101             if(temp.getLink().getLink() != null){
102                temp = temp.getLink();
103                return removeLast();
104             }else {
105                info = temp.getLink().getInfo();
106                temp.setLink(null);
107                temp = first;
108                size--;}}
109          temp = first;
110          return info;
111       }
112
113       @Override
114       public T removeAt(int i) {
115          if (i < 0 || i >= size )
116             throw new IndexOutOfBoundsException();
117          if (prime == 0)
118             temp = first;
119          if (i == 0){
120                info = removeFirst();
```

```
121          }else if (i == size-1)
122             info = removeLast();
123          else if (size == 2)
124             {
125                info = first.getLink().getInfo();
126                first.setLink(first.getLink().getLink());
127                size--;
128             }else if (size > 2 && i > 1){
129                prime = 1;
130                temp = temp.getLink();
131                removeAt(i-1);
132             }else{
133                info = temp.getLink().getInfo();
134                temp.setLink(temp.getLink().getLink());
135                size--;
136             }
137          prime = 0;
138          return info;
139       }
140
141       @Override
142       public T getFirst() {
143          if (isEmpty())
144             throw new IllegalStateException();
145          return first.getInfo();
146       }
147
```

```
148     @Override
149     public T getLast() {
150        if (isEmpty())
151           throw new IllegalStateException();
152        return last.getInfo();
153     }
154
155     @Override
156     public T get(int i) {
157        if (i < 0 || i >= size)
158           throw new IndexOutOfBoundsException();
159        if (i == 0){
160              info = first.getInfo();
161        }else if (i == size-1)
162           info = last.getInfo();
163        else {
164           if (size == 2)
165              info = first.getLink().getInfo();
166           else if (size > 2 && i > 1){
167              temp = temp.getLink();
168              get(i-1);
169           }else{
170              info = temp.getLink().getInfo();
171           }
172        }
173        prime = 0;
174        return info;
```

```
175        }
176
177        @Override
178        public boolean remove(T elem) {
179           if (isEmpty()) return false;
180           if (temp.getInfo().equals(elem)){
181              if (size == 1) {
182                 first = last = null;
183                 temp = first;
184                 size--;
185                 return true;
186              }else{
187                 first = first.getLink();
188                 temp = first;
189                 size--;
190                 return true;}}
191           if (size > 1) {
192           if (temp.getLink().getInfo().equals(elem)){
193              if (size == 2){
194                 first.setLink(null);
195                 temp = first;
196                 size--;
197              return true;
198              }else{
199                 temp.setLink(temp.getLink().getLink());
200                 size--;
201                 temp = first;
```

```
202                 return true;}
203           }else{
204              if (size == 2) return false;
205              else if (temp.getLink().getLink() != null){
206              temp = temp.getLink();
207              return remove(elem);
208              }}}
209           temp = first;
210           return false;
211        }
212
213        @Override
214        public int contains(T elem) {
215           if (temp.getInfo().equals(elem)){
216              temp = first;
217              int toReturn = loc;
218              loc = 0;
219              return toReturn;
220           }
221           else if (temp.getLink() != null){
222              temp = temp.getLink();
223              loc++;
224              return contains(elem);
225           }
226           temp = first;
227           loc = 0;
228           return -1;
```

```
229
230       }
231       @Override
232       public boolean isEmpty() {
233          return size == 0;
234       }
235
236
237
238    }
```

B.2 FrenchPress feedback for initial version

```
FrenchPress 1.4 feedback for RecursiveList.java


Variables such as
     info (7)
are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.


Variables such as
     prime (9)
are declared int but appear to function as boolean flags.
Instead of giving them the values 1 and 0, declare them as boolean
and give them the values true and false.
```

B.3 Student responds to initial feedback

The student took the plug-in's suggestion and made the variable `prime` a `boolean` instead of an `int`. Note changes on lines 9, 117, 129, 137, and 173.

```
1    package structure;
2
3    public class RecursiveList<T> implements ListInterface<T> {
4
5       int size;
6       LLNode<T> first, last, temp;
7       T info;
8       int loc = 0;
9       boolean prime = true;
10
11      @Override
12      public int size() {
13         return size;
14      }
15
16      @Override
17      public ListInterface<T> insertFirst(T elem) {
18         if (elem == null)
19            throw new NullPointerException();
20         LLNode<T> newNode = new LLNode<T>(elem);
21         if (first == null)
22            first = last = newNode;
23         else{
24            newNode.setLink(first);
```

```
25            first = newNode;
26         }
27         size++;
28         temp = first;
29
30         return this;
31      }
32
33      @Override
34      public ListInterface<T> insertLast(T elem) {
35         if (elem == null)
36            throw new NullPointerException();
37         LLNode<T> newNode = new LLNode<T>(elem);
38         if (last == null)
39            first = newNode;
40         else
41            last.setLink(newNode);
42            last = newNode;
43         size++;
44         temp = first;
45
46         return this;
47      }
48
49      @Override
50      public ListInterface<T> insertAt(int index, T elem) {
51         if (index < 0 || index > size)
```

```
52            throw new IndexOutOfBoundsException();
53         if (index == 0)
54            insertFirst(elem);
55         else if (index == size)
56            insertLast(elem);
57         else if (size == 2)
58            {
59               first.setLink(new LLNode<T>(elem));
60               first.getLink().setLink(last);
61               temp = first;
62               size++;
63            }else if (size > 2 && index > 1){
64               temp = temp.getLink();
65               insertAt(index-1,elem);
66            }else{
67               LLNode<T> newNode = new LLNode<T>(elem);
68               newNode.setLink(last);
69               temp.getLink().setLink(newNode);
70               temp = first;
71               size++;
72            }
73         return this;
74      }
75
76      @Override
77      public T removeFirst() {
78         if (isEmpty())
```

```
79            throw new IllegalStateException();
80         info = first.getInfo();
81         first = first.getLink();
82         size--;
83         temp = first;
84         return info;
85      }
86
87      @Override
88      public T removeLast() {
89         if (isEmpty())
90            throw new IllegalStateException();
91         if (size <= 1){
92            info = first.getInfo();
93            temp = first = last = null;
94            size--;}
95         else if(size==2){
96            info = first.getLink().getInfo();
97            first.setLink(null);
98            temp = first;
99            size--;
100        }else{
101           if(temp.getLink().getLink() != null){
102              temp = temp.getLink();
103              return removeLast();
104           }else {
105              info = temp.getLink().getInfo();
```

```
106                 temp.setLink(null);
107                 temp = first;
108                 size--;}}
109           temp = first;
110           return info;
111       }
112 
113       @Override
114       public T removeAt(int i) {
115           if (i < 0 || i >= size )
116               throw new IndexOutOfBoundsException();
117           if (prime == true)
118               temp = first;
119           if (i == 0){
120                 info = removeFirst();
121           }else if (i == size-1)
122               info = removeLast();
123           else if (size == 2)
124               {
125                 info = first.getLink().getInfo();
126                 first.setLink(first.getLink().getLink());
127                 size--;
128               }else if (size > 2 && i > 1){
129                 prime = false;
130                 temp = temp.getLink();
131                 removeAt(i-1);
132               }else{
```

```
133              info = temp.getLink().getInfo();
134              temp.setLink(temp.getLink().getLink());
135              size--;
136           }
137        prime = true;
138        return info;
139     }
140
141     @Override
142     public T getFirst() {
143        if (isEmpty())
144           throw new IllegalStateException();
145        return first.getInfo();
146     }
147
148     @Override
149     public T getLast() {
150        if (isEmpty())
151           throw new IllegalStateException();
152        return last.getInfo();
153     }
154
155     @Override
156     public T get(int i) {
157        if (i < 0 || i >= size)
158           throw new IndexOutOfBoundsException();
159        if (i == 0){
```

```
160                 info = first.getInfo();
161         }else if (i == size-1)
162             info = last.getInfo();
163         else {
164             if (size == 2)
165                 info = first.getLink().getInfo();
166             else if (size > 2 && i > 1){
167                 temp = temp.getLink();
168                 get(i-1);
169             }else{
170                 info = temp.getLink().getInfo();
171             }
172         }
173         return info;
174     }
175
176     @Override
177     public boolean remove(T elem) {
178         if (isEmpty()) return false;
179         if (temp.getInfo().equals(elem)){
180             if (size == 1) {
181                 first = last = null;
182                 temp = first;
183                 size--;
184                 return true;
185             }else{
186                 first = first.getLink();
```

```
187                 temp = first;
188                 size--;
189                 return true;}}
190           if (size > 1) {
191           if (temp.getLink().getInfo().equals(elem)){
192              if (size == 2){
193                 first.setLink(null);
194                 temp = first;
195                 size--;
196              return true;
197              }else{
198                 temp.setLink(temp.getLink().getLink());
199                 size--;
200                 temp = first;
201                 return true;}
202           }else{
203              if (size == 2) return false;
204              else if (temp.getLink().getLink() != null){
205              temp = temp.getLink();
206              return remove(elem);
207              }}}
208           temp = first;
209           return false;
210        }
211
212        @Override
213        public int contains(T elem) {
```

```
214            if (temp.getInfo().equals(elem)){
215               temp = first;
216               int toReturn = loc;
217               loc = 0;
218               return toReturn;
219            }
220            else if (temp.getLink() != null){
221               temp = temp.getLink();
222               loc++;
223               return contains(elem);
224            }
225            temp = first;
226            loc = 0;
227            return -1;
228
229         }
230         @Override
231         public boolean isEmpty() {
232            return size == 0;
233         }
234
235
236
237      }
```

B.4 FrenchPress feedback for revised version

`FrenchPress 1.4 feedback for RecursiveList.java`

```
Variables such as

     info (7)

are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.


Boolean expressions such as

     prime == true (117)

are redundant and can be shortened.  If B is a boolean expression,

     B == true or B != false means the same thing as B

     B != true or B == false means the same thing as !B.
```

B.5 Student responds to new feedback

To avoid repeating a long class definition, I show here a `diff` between the previous version and the final version.

```
117c117
< 117          if (prime == true)
---
> 117          if (prime)
120c120
< 120                info = removeFirst();
---
> 120             info = removeFirst();
160c160
```

```
< 160                info = first.getInfo();
---
> 160             info = first.getInfo();
```

B.6 FrenchPress feedback for final version

```
FrenchPress 1.4 feedback for RecursiveList.java


Variables such as
     info (7)
are declared at the class level but appear to function as local
variables. Each of these variables could be declared locally
in each method where it is used. To find all the places a variable
is used, select the variable name and Eclipse will highlight
every occurrence of that variable.
```

APPENDIX C

MOTIVATION FOR UNIMPLEMENTED RULES

The following example from the Spring 2008 class of CMPSCI 187 illustrates the need for both of the rules discussed in Section 1.7, which were not implemented in the FrenchPress prototype. The student submitted this program for the assignment in Section A.1. `BirthdayDriver.java` has an over-ambitious constructor that runs the entire simulation (lines 4–7). `BirthdayList.java` shows an inappropriate inheritance relationship between class `BirthdayList` and the Java collections class `ArrayList` (line 4).

```
 1   public class BirthdayDriver {
 2      int size;
 3
 4      public BirthdayDriver(int num){
 5         size=num;
 6         run();
 7      }
 8
 9      public void run(){
10         int matches=0;
11         double average=0.0;
12
13         for (int j=0;j<10000;j++){
14            BirthdayList b = new BirthdayList(size);
```

```
15          if (b.hasMatch())
16             matches++;
17       }
18
19       average = ((double)matches)/10000.0;
20
21       System.out.println("People: "+size+"  Probablility: "+average);
22    }
23
24    public static void main(String[] args){
25       for (int j=10;j<=30;j++){
26          new BirthdayDriver(j);
27       }
28    }
29 }
```

```
 1 import java.util.ArrayList;
 2 import java.util.Random;
 3
 4 public class BirthdayList extends ArrayList<Integer>{
 5    Random rand = new Random();
 6    int size;
 7    static final long serialVersionUID=0;
 8
 9    public BirthdayList(int num){
10       size=num;
11       clear();
```

```
12          newList();
13       }
14
15       public void newList(){
16          for (int j=0;j<size;j++){
17             add(rand.nextInt(365));
18          }
19       }
20
21       public boolean hasMatch(){
22          boolean foo=false;
23
24          for (int j=0;j<size;j++){
25             for (int i=j+1;i<size;i++){
26                 if(get(i).equals(get(j)))
27                   foo=true;
28             }
29          }
30
31          return foo;
32       }
33    }
```

APPENDIX D

CLASSROOM TRIAL INFORMED CONSENT FORM

Consent Form for Participation in a Research Study

University of Massachusetts Amherst

Researcher(s): Hannah Blau and Prof. W. Richards Adrion

Study Title: Automated feedback for Java program flaws

---

1. WHAT IS THIS FORM?

This form is called a Consent Form. It will give you information about the study so you can make an informed decision about participation in this research. This consent form will describe what you will need to do to participate and any known inconveniences that you might experience while participating.

2. WHO IS ELIGIBLE TO PARTICIPATE?

All students enrolled in CMPSCI 187 and at least 18 years old are eligible to participate.

3. WHAT IS THE PURPOSE OF THIS STUDY?

The purpose of this research study is to evaluate educational software called FrenchPress. FrenchPress gives you automated feedback on your Java programs as you are writing them.

4. WHERE WILL THE STUDY TAKE PLACE AND HOW LONG WILL IT LAST?

You will install and run FrenchPress on the same computer you use for doing your CMPSCI 187 programming assignments. You are asked to try the software for the rest of the semester.

5. WHAT WILL I BE ASKED TO DO?

If you agree to take part in this study, you will be asked to install the FrenchPress plugin in Eclipse on your computer. The researchers ask you to run FrenchPress at least once for each programming assignment you work on (but you may run it as often as you like). Each time you request feedback from FrenchPress, the software will save a copy of your program on your own hard drive. You will be asked to submit the saved data as part of your Java project for each assignment in the course.

You will be expected to complete a short online survey after each assignment for which you use FrenchPress. The surveys will ask questions about how easy or difficult it was to install FrenchPress, was the software easy or difficult to use, did you get any feedback from FrenchPress, was the feedback easy or difficult to understand, and did you modify your program in response to what FrenchPress told you. You may skip any question you feel uncomfortable answering.

6. WHAT ARE MY BENEFITS OF BEING IN THIS STUDY?

The purpose of FrenchPress is to give you feedback on your program while you are working on it, so you can eliminate the flaws that the software has pointed out to you. This will help you learn to avoid making the mistakes FrenchPress is capable of identifying for you. Fixing these flaws will not have any effect on your grade for the assignment, but it will help you to be a better Java programmer. The researchers want your opinions about the software as expressed in your survey responses. Knowing

what you think about FrenchPress will help them improve it for future CMPSCI 187 students.

7. WHAT ARE MY RISKS OF BEING IN THIS STUDY?

The main risk you run by participating in this study is the possibility that FrenchPress will give you poor guidance. The software is still under development. The researchers conducting this study cannot guarantee that all the feedback you receive from FrenchPress will be appropriate for the program you are writing. You might get frustrated if FrenchPress gives you a confusing feedback message. Another risk of participating in this study is the possibility that your classmates or a teaching assistant or the professor could read the feedback you get from FrenchPress. To reduce this risk, the researchers will never show your feedback or saved data to anyone else without first removing any information that could identify you.

Participation in the study could require modest additional effort. You will have to install the FrenchPress plugin on your computer. For each assignment you will have to submit the saved program data and complete an online survey. These tasks should be simple to accomplish but they will take some time.

8. HOW WILL MY PERSONAL INFORMATION BE PROTECTED?

The following procedures will be used to protect the confidentiality of your study records. FrenchPress will capture your Java program each time you request feedback. These recorded programs will be stored on your own machine until you upload them as part of your finished assignment to Moodle. At the end of the semester, the researchers will match up your FrenchPress stored data with your survey responses, and then remove from these files any information that identifies you (your name or student number). The programs and survey responses will be stored on a password-protected computer to prevent access by unauthorized users. Only the members of the research staff will have access to the password. The professor of the course will

not be permitted to look at any data stored by FrenchPress. At the conclusion of this study, the researchers may publish their findings. Information will be presented in summary format and you will not be identified in any publications or presentations.

9. WILL I RECEIVE ANY PAYMENT FOR TAKING PART IN THE STUDY?

Extra credit of 0.10 will be added to your final grade for CMPSCI 187 if you complete your part in the research study. For example, if your final grade were 3.10, high in the B range, the extra credit would give you a 3.20, in the B+ range. Partial credit may be given for partial completion.

Completing your part in the study means that for every programming assignment, you use FrenchPress at least once, submit the program versions that are automatically saved by FrenchPress, and take the online survey.

Participating in the research study is not the only way to earn extra credit. The Moodle page that gives instructions for this study also describes an alternate assignment to earn the same amount of extra credit. To get extra credit you may choose the FrenchPress study or the alternate activity, but not both.

10. WHAT IF I HAVE QUESTIONS?

Take as long as you like before you make a decision. We will be happy to answer any question you have about this study. If you have further questions about this project or if you have a research-related problem, you may contact the researcher, Hannah Blau at (413) 584-0963 or blau@cs.umass.edu. If you have any questions concerning your rights as a research subject, you may contact the University of Massachusetts Amherst Human Research Protection Office (HRPO) at (413) 545-3428 or humansubjects@ora.umass.edu.

11. CAN I STOP BEING IN THE STUDY?

You do not have to be in this study if you do not want to. If you agree to be

in the study, but later change your mind, you may drop out at any time. There are no penalties or consequences of any kind if you decide that you do not want to participate.

12. HOW DO I FILL OUT THIS FORM?

First, read the form so you know what you are agreeing to. If you are ready to participate in the study, please sign in section 14.

- Sign with a pen, not a pencil.
- Sign on the line that reads Participant Signature; leave the bottom signature line blank.
- Your signature must be hand-written (not typed on the computer). Sign in the first blank, then write (or type) your name legibly in the second blank.
- Fill in todays date, including the year.
- Fill in your student ID number. Be careful to get all the digits right.

13. HOW DO I SUBMIT THIS FORM?

After you have read and signed the consent form, you can submit it in one of three ways:

- Hand it to the researcher, Hannah Blau, in class today.
- Hand it to a staff member in the main office of the School of Computer Science. Ask the staff member to put your form in the secure box for 187 consent forms. The office is located in room 100 on the ground floor of the Computer Science Building.
- Submit electronically. You must first sign a hardcopy of the consent form by hand. You can sign this copy, or print out the .pdf file from the course Moodle

page. Typed signatures are not acceptable. Then scan all three pages of your signed consent form and email the scan to Hannah at blau@cs.umass.edu.

14. SUBJECT STATEMENT OF VOLUNTARY CONSENT

When signing this form I am agreeing to voluntarily enter this study. I have had a chance to read this consent form, and it was explained to me in a language that I use and understand. I have had the opportunity to ask questions and have received satisfactory answers. I understand that I can withdraw at any time. I can get a copy of this Informed Consent Form from the course website.

____________________	_______________	__________
Participant Signature:	Print Name:	Date:

____

Student ID Number:

By signing below I indicate that the participant has read and, to the best of my knowledge, understands the details contained in this document and has been given a copy.

____________________	_______________	__________
Signature of Person Obtaining Consent	Print Name:	Date:

APPENDIX E

FRENCHPRESS USER SATISFACTION SURVEY

Study participants responded to this survey after each programming assignment for which they used the FrenchPress software. Questions 2–4 appeared only on the first survey. The remaining questions appeared on all the surveys.

To get extra credit for participating in the classroom trial, you must provide your student ID number. If you prefer to answer this survey anonymously you may skip question 1, but you will not get your extra credit.

1. Please enter your student ID number for this survey to count toward extra credit.
   [text box response]

2. What operating system are you using? Please include the edition/version (e.g. Windows 7 Professional or OS X 10.9.4).
   [text box response]

3. How easy or difficult was it to install FrenchPress?

   - Very easy
   - Moderately easy
   - Neither easy nor difficult
   - Moderately difficult
   - Very difficult

4. How long did it take you to install FrenchPress?
    - Very quick
    - Moderately quick
    - Neither quick nor slow
    - Moderately slow
    - Very slow
5. How often did FrenchPress crash or "freeze up" on you for this assignment?
    - Very often
    - Somewhat often
    - Neither often nor rarely
    - Rarely
    - Never
6. Did FrenchPress find any flaws in your program for this assignment?
    - Yes
    - No
7. For this assignment, was the feedback from FrenchPress confusing or easy to understand?
    - FrenchPress found no flaws in my program
    - Very confusing
    - Moderately confusing
    - Neither confusing nor easy to understand
    - Moderately easy to understand

- Very easy to understand

8. For this assignment, was the feedback from FrenchPress helpful or unhelpful?

    - FrenchPress found no flaws in my program
    - Very helpful
    - Rather helpful
    - Neither helpful nor unhelpful
    - Rather unhelpful
    - Very unhelpful

9. Did the FrenchPress feedback for this assignment lead you to change your program?

    - FrenchPress found no flaws in my program
    - Yes
    - No

10. Are you satisfied or dissatisfied with the performance of FrenchPress on this assignment?

    - Very satisfied
    - Somewhat satisfied
    - Neither satisfied nor dissatisfied
    - Somewhat dissatisfied
    - Very dissatisfied

11. How can we improve FrenchPress?

    [text box response]

BIBLIOGRAPHY

[1] Amelung, Mario, Piotrowski, Michael, and Rösner, Dietmar. EduComponents: Experiences in e-assessment in computer science education. In *ITiCSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (New York, NY, 2006), ACM, pp. 88–92.

[2] Arnow, David, and Barshay, Oleg. WebToTeach: An interactive focused programming exercise system. In *FIE 1999: 29th Annual ASEE/IEEE Frontiers in Education Conference* (1999), vol. 1, pp. 12A9/39–12A9/44.

[3] ASM 5.0.4. *http://asm.ow2.org/*.

[4] Badros, Greg J. JavaML: A markup language for Java source code. *Computer Networks 33*, 1 (2000), 159–177.

[5] Benford, S D, Burke, E K, Foxley, E, and Higgins, C A. The Ceilidh system for the automatic grading of students on programming courses. In *ACM-SE 33: Proceedings of the 33rd Annual Southeast Regional Conference* (New York, NY, 1995), ACM, pp. 176–182.

[6] Bloch, Joshua. *Effective Java*, 2nd ed. The Java Series. Addison-Wesley, 2008.

[7] Checkstyle 6.9. *http://checkstyle.sourceforge.net/*.

[8] Dawis, Rene V. Likert scale. In *International Encyclopedia of the Social Sciences*, W. A. Darity Jr., Ed., 2nd ed., vol. 4. Macmillan Reference, Detroit, MI, 2008, pp. 447–448.

[9] Dexter, Scott. On automated checking of Java applets. *Journal of Computing Sciences in Colleges 15*, 5 (May 2000), 84–95.

[10] Dillman, Don A., Smyth, Jolene D., and Christian, Leah Melani. *Internet, Mail, and Mixed-Mode Surveys: The Tailored Design Method*, 3rd ed. John Wiley & Sons, 2009.

[11] Edwards, Stephen H. Rethinking computer science education from a test-first perspective. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, 2003), ACM, pp. 148–155.

[12] FindBugs 3.0.1. *http://findbugs.sourceforge.net/*.

[13] Fokaefs, Marios, Tsantalis, Nikolaos, and Chatzigeorgiou, Alexander. Jdeodorant: Identification and removal of feature envy bad smells. In *ICSM 2007: 23rd IEEE International Conference on Software Maintenance* (2007), IEEE, pp. 519–520.

[14] Fokaefs, Marios, Tsantalis, Nikolaos, Stroulia, Eleni, and Chatzigeorgiou, Alexander. Jdeodorant: Identification and application of extract class refactorings. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ACM, pp. 1037–1039.

[15] Higgins, Colin, Symeonidis, Pavlos, and Tsintsifas, Athanasios. The marking system for CourseMaster. In *ITiCSE '02: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (New York, NY, 2002), ACM, pp. 46–50.

[16] Higgins, Colin A., Gray, Geoffrey, Symeonidis, Pavlos, and Tsintsifas, Athanasios. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing 5*, 3 (2005), 5.

[17] Higgins, Colin A., Hegazy, Tarek, Symeonidis, Pavlos, and Tsintsifas, Athanasios. The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies 8*, 3 (2003), 287–304.

[18] Hovemeyer, David, and Pugh, William. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[19] Hristova, Maria, Misra, Ananya, Rutter, Megan, and Mercuri, Rebecca. Identifying and correcting Java programming errors for introductory computer science students. In *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, 2003), ACM, pp. 153–156.

[20] Jackson, David, and Usher, Michelle. Grading student programs using ASSYST. *SIGCSE Bull. 29*, 1 (1997), 335–339.

[21] JDeodorant 5.0.15. *http://www.jdeodorant.com/*.

[22] Joy, Mike, Griffiths, Nathan, and Boyatt, Russell. The BOSS online submission and assessment system. *Journal on Educational Resources in Computing 5*, 3 (2005), 2.

[23] McCabe, T.J. A complexity measure. *IEEE Transactions on Software Engineering 2*, 4 (1976), 308–320.

[24] Moll, Robert. *Interactive Java: An Online Approach to Java Learning*. Edition 3.2, 2015. *http://ijava.cs.umass.edu/*.

[25] Morris, Derek S. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *FIE 2003: 33rd Annual ASEE/IEEE Frontiers in Education Conference* (2003), vol. 3, pp. S3F 1–6.

[26] Murphy-Hill, Emerson, and Black, Andrew P. An interactive ambient visualization for code smells. In *SOFTVIS '10: Proceedings of the 5th International Symposium on Software Visualization* (New York, NY, USA, 2010), ACM, pp. 5–14.

[27] PMD 5.3.3. *http://pmd.sourceforge.net/.*

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

[29] Qiu, Lin, and Riesbeck, Christopher K. Facilitating critiquing in education: The design and implementation of the Java Critiquer. In *Proceedings of the International Conference on Computers in Education (ICCE)* (Hong Kong, 2003).

[30] Reek, Kenneth A. The TRY system -or- how to avoid testing student programs. In *SIGCSE '89: Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, 1989), ACM, pp. 112–116.

[31] Reek, Kenneth A. A software infrastructure to support introductory computer science courses. In *SIGCSE '96: Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, 1996), ACM, pp. 125–129.

[32] Saikkonen, Riku, Malmi, Lauri, and Korhonen, Ari. Fully automatic assessment of programming exercises. In *ITiCSE '01: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (New York, NY, 2001), ACM, pp. 133–136.

[33] Singh, Rishabh, Gulwani, Sumit, and Solar-Lezama, Armando. Automated feedback generation for introductory programming assignments. In *PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), ACM, pp. 15–26.

[34] Suleman, Hussein. Automatic marking with Sakai. In *SAICSIT '08: Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries* (New York, NY, 2008), ACM, pp. 229–236.

[35] Truong, Nghi, Bancroft, Peter, and Roe, Paul. Learning to program through the web. In *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (New York, NY, 2005), ACM, pp. 9–13.

[36] Truong, Nghi, Roe, Paul, and Bancroft, Peter. Static analysis of students' Java programs. In *ACE '04: Proceedings of the 6th Conference on Australasian Computing Education* (Darlinghurst, Australia, 2004), Australian Computer Society, Inc., pp. 317–325.

[37] Truong, Nghi, Roe, Paul, and Bancroft, Peter. Automated feedback for “fill in the gap” programming exercises. In *ACE ’05: Proceedings of the 7th Australasian Conference on Computing education* (Darlinghurst, Australia, 2005), Australian Computer Society, Inc., pp. 117–126.

[38] Tsantalis, Nikolaos, Chaikalis, Theodoros, and Chatzigeorgiou, Alexander. Jdeodorant: Identification and removal of type-checking bad smells. In *CSMR 2008: 12th European Conference on Software Maintenance and Reengineering* (2008), IEEE, pp. 329–331.

[39] Vermeulen, Allan, Ambler, Scott W., Bumgardner, Greg, Metz, Eldon, Misfeldt, Trevor, Shur, Jim, and Thompson, Patrick. *The Elements of Java Style*. SIGS Reference Library. Cambridge University Press, Cambridge, UK, 2000.