FrenchPress Gives Students Automated Feedback on Java Program Flaws

Hannah Blau  
School of Computer Science  
University of Massachusetts Amherst  
blau.hannah@acm.org

J. Eliot B. Moss  
School of Computer Science  
University of Massachusetts Amherst  
moss@cs.umass.edu

ABSTRACT

We created an Eclipse plug-in called FrenchPress 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 runtime 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 datatype. FrenchPress delivers explanatory messages in a vocabulary appropriate to the student’s current level.

This paper reports preliminary results of a formative evaluation of FrenchPress conducted in a Fall 2014 data structures and algorithms course. User satisfaction survey responses indicate that among the students who received substantive diagnostic suggestions from FrenchPress, the percentage who were motivated to modify their program varied from 36% to 64% on four different assignments.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education;  
D.2.5 [Software Engineering]: Testing and Debugging—Diagnostics; D.3.2 [Programming Languages]: Language Classifications—Java

Keywords

Eclipse plug-in, educational technology, program analysis, static analysis

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

ITiCSE ’15, July 6–8, 2015, Vilnius, Lithuania.  
Copyright is held by the owner/author(s). Publication rights licensed to ACM.  
ACM 978-1-4503-3440-2/15/07 ...$15.00.  
DOI: http://dx.doi.org/10.1145/2729094.2742622.

1. INTRODUCTION

Anyone who has taught a 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 instructor 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/graded can spend only a few minutes on each program, checking to see if it produces the expected output for selected test inputs. 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 assistant 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 TAs 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 from the teaching assistant. He might even dismiss these comments as a reflection of the TA’s idiosyncrasies. The student could reach upper level courses before he gets an instructor 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 instructor.

To address this problem we developed FrenchPress, a static analysis tool that partially automates the task of giving students feedback concerning their Java programs. Students get guidance to make improvements without depending on the instructor or the teaching assistant to review their code. Our target population is not complete 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. We chose Java because it has been widely adopted for undergraduate curricula in object-oriented programming. Our goal is not to diagnose compile-time or runtime errors, or logical flaws that produce incorrect output. We are after the silent flaws — flaws the student is unable to identify for himself because he gets no feedback from the pro-
gramming 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. Other diagnostic rules target the all too common misunderstanding of the boolean datatype.

FrenchPress is easy to use for both students and instructors. The instructor does not have to write a model solution or customized diagnostics for each assignment. The analysis applies to any compilable Java program. The Eclipse plug-in integrates seamlessly into the student’s development environment. The student can get feedback as he works, so he can iteratively improve the code before submitting his project. We conducted a classroom trial of FrenchPress in a data structures and algorithms course during the Fall 2014 semester. User satisfaction survey responses indicate that among the students who received substantive diagnostic suggestions from FrenchPress, the percentage who were motivated to modify their program varied from 36% to 64% on four different assignments.

2. RELATED WORK

Advanced beginner students of Java are not well served by the program analysis tools created in the past decade. Existing style checkers (Expresso [5]) and automatic assessment systems (CourseMarker [3], BOSS [6]) developed in academic institutions are aimed at students just learning to program. Expresso is a pre-compiler for Java programs that helps novices avoid common mistakes that can lead to incomprehensible compiler messages or unexpected runtime behavior. Expresso will flag errors such as unbalanced parentheses, braces, and brackets; confusion of = and ==; and misplaced semicolons. The Environment for Learning to Program (ELP [11]) does both dynamic and static analysis, but only on program snippets submitted by students to complete fill-in-the-gap exercises in the introductory Java course. ELP checks for unused variables, unused parameters, redundant logical expressions, numeric literals that should be named constants, and other stylistic blunders. ELP performs a structural similarity analysis between the student’s code and the instructor’s model solution(s) for the exercise. This approach only works with fill-in-the-gap exercises because they are so short there is relatively little room for structural variation. The Java Critiquer [9] addresses localized stylistic issues, including boolean expressions, increment operators, unnecessary parentheses, and floating point data types. The Java Critiquer [8] can analyze only one class file at a time. FrenchPress handles more challenging assignments, involving multiple class definitions, that students would encounter in their second or third Java course.

The MIT system described in [10] 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.

On the other end of the spectrum, program analysis tools such as FindBugs [2, 4] and PMD [8] are geared for large-scale professional projects. Their bug reports assume a sophisticated understanding that college students are unlikely to attain in their first year of exposure to Java. 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. FrenchPress is similar in spirit to systems such as Stench Blossom [7] that alert programmers to code smells, questionable program features that indicate the code should be refactored or redesigned. FrenchPress flaws can be considered code smells specific to advanced beginner Java programmers.

3. OBJECTIVES

The main objectives of the FrenchPress project are:

1. Give the student effective and useful feedback without increasing TA or instructor 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 instructor from the burden of writing customized design checks from scratch for each programming exercise.

3.1 Effective feedback

Many of the programming best practices articulated in books such as *Effective Java* [1] 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 we give this student must be relevant for his situation, not some hypothetical situation of large-scale software development.

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.
3.2 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. We feel it is essential to integrate the tool into the student’s development environment so he can run the diagnostics while still working on his program. We decided to implement FrenchPress as an Eclipse 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. We 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. 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. Figure 1 shows a screenshot of FrenchPress.

3.3 Easy for the instructor

In designing a pedagogic program analysis system, we face a tradeoff between its range of applicability and the quality and usefulness of its feedback. If the diagnostics are tailored to a specific programming exercise, the system can give detailed feedback because the course instructor has a good idea of what she expects to see in a student solution. Some educational tools (for example, [11] and [10]) 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, we sought an approach that would be less labor-intensive for them while still offering a benefit to their students. We opted instead for 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 CS2 courses, as evidenced by the multiplicity of textbooks on offer to teach data structures in Java. FrenchPress diagnostics are not limited to a particular course syllabus. The rules described in Section 4 embody principles of good programming that are appropriate for assignments from any of these textbooks.

4. DIAGNOSTICS

The current version of the FrenchPress prototype comprises seven diagnostic rules. We chose these rules by examining a set of student programs submitted for the data structures and algorithms course. We identified flaws that are amenable to automated analysis and frequent enough to be worth investing our effort. Some of FrenchPress’s rules overlap with the diagnostics of earlier pedagogic systems including the Java Critiquer [9]. We wanted a well-rounded set of diagnostics that would cover commonly occurring flaws in CS2 student submissions, whether or not those flaws had been addressed in previous work. To implement these rules, the plug-in crawls the abstract syntax tree, exploits Eclipse’s type hierarchy and call graph search functions, and performs data flow analysis. Some of these questionable 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. We grouped the rules into four broad categories so that explanatory messages about related concepts would be displayed together. We ordered the categories according to our judgment of their severity. Misconceptions about the use of variables and access modifiers seem to us more serious in their potential consequences than redundant boolean expressions or unexpected for loop control variables. We want to first draw the student’s attention to the issues we consider to be more significant. For each diagnostic we include below an example of the feedback FrenchPress gives the student. If no diagnostic rules are triggered the student gets a message congratulating him on his good work.

4.1 Misuse of fields

Advanced beginner Java programmers 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. Perhaps this habit is a vestige of high school programming classes where they were encouraged to declare all the variables at the top of the program.

4.1.1 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. An example of feedback follows; the numbers in parentheses are line numbers in the .java file.

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.

4.1.2 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.

4.2 Misuse of the public modifier

CS2 students often do not attach much importance to the principle of hiding the details of a class’s data representation and internal methods. They routinely declare instance variables public or make a method public even though it is
not part of the API for the class. This is understandable in the case where the student is working on his program alone (true for most CS2 projects) and has no intention of re-using his code after the due date of the assignment. Nevertheless we try to remind students of better programming practices.

4.2.1 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.

4.2.2 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.

4.3 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.

4.3.1 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.

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

4.4 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
should not be used as for loop control variables. It is preferable to use a separate variable as in, for (int i = ...).

5. FRENCHPRESS CLASSROOM TRIAL

We conducted a classroom trial of FrenchPress in a Fall 2014 data structures and algorithms course. The trial covered four programming assignments. 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 user satisfaction survey. They were rewarded for their cooperation by a small amount of extra credit towards their final grade in the course.

The number of survey 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. We had to weed out responses from students who completed the survey even though they had not run FrenchPress on their program. The number of legitimate survey responses varied during the trial from a low of 43 to a high of 47.

Table 1 summarizes survey responses for the four assignments to the question, “Did the FrenchPress feedback for this assignment lead you to change your program?” No Flaws indicates that the student chose the option “FrenchPress found no flaws in my program”. Certainly we do not expect FrenchPress to influence the student’s coding choices if he did not get any substantive feedback on the files he selected to analyze. 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. We wanted to give students some leeway so they would not drop out of the study because they found the requirements of participation too time-consuming.

The second and third columns of Table 1 show the percentage of all survey respondents who say they did/did not change their program in light of the feedback they received from FrenchPress. The Yes response declined from a high of 54% on the first assignment to a low of 11% on the final assignment. The fourth column of the table calculates the Yes response as a percentage of those students who said FrenchPress found flaws in their code: Yes/(Yes + No). This measure ranges from 64% on the first assignment in the study to 41% on the final assignment, but it does not show a smooth decrease in between. The sharp decline 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. If FrenchPress is achieving its educational goals, we would expect the frequency of feedback to decline over time as students learn to avoid the mistakes they were making when they first starting using the plug-in.

We have not yet scrutinized the trajectory of each individual in the study population to confirm or contradict this explanation of the survey data.

The most frequently triggered rule is 4.2.2, a method declared to be final, which is not an instance method. Our rule would suggest that assigning to an instance variable. Our rule would indicate that any method does not stop at constructing an object but try to run the entire program by calling instance methods. This anomaly is difficult to detect with static analysis but we could at least warn the student if his constructor calls instance methods that are neither private nor final.

We eliminated these two rules from the FrenchPress prototype because they seemed less applicable to CS2 as it is now taught at our institution. The class size of our data structures and algorithms course continues to increase year after year. This has forced a move to automated grading of submissions based solely on input/output behavior. Automated grading requires greater uniformity in the submitted cause in fact the method is not called anywhere, in the class where it is defined 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. We need to refine this rule to distinguish between methods that are called only within the class where they are defined, and methods that are not called at all.

6. FUTURE WORK

The feedback FrenchPress currently offers is rudimentary. For the simpler rules such as for loop control, perhaps the two- or three-sentence message FrenchPress now displays is adequate. More subtle rules such as field should have been local variable or integer variable should have been boolean 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 open a new window and read a more detailed writeup including an example.

We also realized over the course of the classroom trial that the term “flaw” can be offensive to students who are proud of their programming skills. We believe the vocabulary of “suggesting improvements” instead of “finding flaws” will be more readily accepted by our intended audience.

We have ordered FrenchPress’s diagnostic rules to reflect our judgment of their relative importance for the student’s understanding of Java. Another professor may have a different opinion about the best way to order multiple feedback messages. In a future version of the plug-in we 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.

Our original plan 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. Our rule 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 we have observed 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 we could at least warn the student if his constructor calls instance methods that are neither private nor final.

We eliminated these two rules from the FrenchPress prototype because they seemed less applicable to CS2 as it is now taught at our institution. The class size of our data structures and algorithms course continues to increase year after year. This has 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. The CS2 professor provides a starter project for each assignment that the student imports into Eclipse. The student may add classes or methods but must not deviate from the program structure expected by the automated tests.

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. FrenchPress 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. When the plug-in is deployed in a smaller class that can afford to give students more flexibility, we will implement the inheritance and constructor rules.

7. 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. When students graduate to the next level of instruction they outgrow these tools because they have made, and learned from, most of the beginner mistakes. 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 CS2 student. We have 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. 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 anytime and anywhere he needs it. Whether at a community college or a four-year institution, introductory CS courses become “gateway courses” where a significant percentage of women and under-represented minorities quit the discipline—often because they lack confidence in their skills. By improving the delivery of instructional feedback in these courses, FrenchPress will contribute to the retention of at-risk students.

8. ACKNOWLEDGMENTS


9. REFERENCES