Automated Style Feedback for Advanced Beginner Java Programmers

Hannah Blau Samantha Kolovson W. Richards Adrion Robert Moll
College of Information and Computer Sciences
University of Massachusetts
Amherst, Massachusetts 01003-9264
blau.hannah@acm.org skolovso@umass.edu adrion@cs.umass.edu moll@cs.umass.edu

***Abstract*—We created an Eclipse plug-in called FrenchPress that offers students feedback on their Java programming style. It is designed not for novices but for students taking their second or third Java course. Advanced beginner students know enough to produce a program with the desired input/output behavior, but fail to understand it could still be poorly written. Large class sizes in introductory courses make it difficult for instructors to give their students individualized attention. FrenchPress automates a small subset of the feedback students might have received from educators. The system diagnoses issues characteristic of programmers who have not yet assimilated the object-oriented paradigm, such as misuse of the `public` modifier, fields that should have been local variables, and instance variables that should have been class constants. We conducted a classroom trial of the plug-in covering four assignments in data structures and algorithms. Among students whose code triggered one or more 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. On the user surveys following each assignment, 56-66% of respondents said they were satisfied with FrenchPress performance.**

I. Introduction

Holding office hours shortly before the due date of a programming assignment can be a rough ride. Frustrated students line the hallway waiting for a few minutes of the instructor's attention. They have a compile-time or run-time error they cannot decipher and therefore cannot fix, or the program is producing incorrect output but the student lacks debugging skills to locate the fault. While the student is painfully aware of these obstacles, he might be completely oblivious to stylistic blunders the professor or T.A. observes as she examines the program. Even code that is performing as intended might fail to conform to professional Java programming practices. But the student's moment of crisis is not a propitious time to educate him about coding standards. He will not be receptive to suggestions for improving the style of his program when his immediate concern is achieving the desired input/output behavior in time to meet the submission deadline.

When is the right moment to give students stylistic feedback on their code? In many introductory courses the class sizes are so large that automated grading is the only feasible option to achieve a reasonable turnaround on student submissions. When programming assignments are run through a test harness, it can be difficult to incorporate style points into the grading rubric. Where class sizes permit manual grading, the instructor could include comments on coding style in her written feedback to the student. There is, however, no guarantee that the student will ever read those comments. By the time the grader's feedback is available, the student has already moved on to the next project and might have lost interest in learning how he could have improved the previous one.

FrenchPress is our response to the challenge of providing useful and timely stylistic feedback to advanced beginner students. FrenchPress is a plug-in for Eclipse, the IDE required for assignments in our undergraduate data structures and algorithms course. The student can run the plug-in as he is working on his program, whenever he is ready to entertain stylistic suggestions. FrenchPress identifies a limited set of poor programming practices that are characteristic of students who have not yet assimilated the object-oriented paradigm. These issues include misuse of the `public` modifier, fields that should have been local variables, and instance variables that should have been class constants. The plug-in does not check for spacing, comments, capitalization patterns in identifiers, and other conventions of Java coding that are very competently covered by a tool such as Checkstyle [1].

Our target population is students in their second or third Java programming course. Experience suggests that novices taking their first course are not ready to absorb stylistic guidance when they are still fighting with the compiler to get their program to run. FrenchPress is aimed at students who have learned enough Java to produce a working program, but have never participated in a large software project where code quality can have long-term consequences for the productivity of other team members. Our goal is to nudge these students in the direction of professional programming practices, without writing code for them. The plug-in suggests changes in the program but it is up to the student whether he wants to edit his code in light of the FrenchPress feedback.

In [2] we discuss in more detail the objectives that motivated the design of the FrenchPress plug-in. Here we give a brief overview of the tool's repertoire of diagnostics and its user interface. This paper focuses on the results of a classroom trial we conducted in the Fall 2014 undergraduate data structures and algorithms course (CS2).

II. FrenchPress Diagnostics

FrenchPress diagnostics aim to correct programming style blunders that are most common among beginner and advanced beginner Java learners. Experienced programmers are not the intended audience of this pedagogic tool, and some of the rules would be inappropriate for them. The plug-in prototype comprises seven diagnostic rules, one of which never triggered during the classroom trial. We include below examples of FrenchPress feedback messages for the six rules that proved effective. Numbers in parentheses refer to lines in the `.java` file. Some of these questionable coding choices reveal the student's poor grasp of the object-oriented programming paradigm, including concepts of access control and the distinction among class, instance, and local variables. Other rules address the all too common misunderstanding of the boolean data type. Further detail in [2].

A. Misuse of fields

Rule 1. Field could have been a local variable:

```
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.
```

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

```
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.
```

B. Misuse of the public modifier

Rule 3. Instance variable declared public:

```
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.
```

Rule 4. Non-static method declared public:

```
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.
```

C. Misunderstanding booleans

Rule 5. Integer variable used as a boolean flag:

```
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.
```

Rule 6. Redundant boolean expressions:

```
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.
```

III. Running the FrenchPress Plug-in

The plug-in does not run continuously in the background, as this might risk overwhelming the student with confusing feedback on half-written code. The student chooses FrenchPress from a menu in the Package Explorer view of the Eclipse IDE. He can run FrenchPress on on a single `.java` file, or on all the `.java` files in the `src` folder of the project. FrenchPress 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 order of the rules listed in Section II. Figure 1 shows a screenshot of FrenchPress running on the `.java` file displayed in the Eclipse editor window.

IV. FrenchPress in Context

Why create FrenchPress when there are already so many tools available to analyze Java programs? 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 cause compile-time or run-time errors, and common mix-ups that lead to incorrect program behavior (for example, = in place of ==). FrenchPress goes after the silent flaws that do *not* cause error messages or bad output: flaws the student might never notice because nothing in the development process signals a problem.

Table I summarizes the evaluation mechanisms available in some of the systems that have been developed for the automated assessment of programming exercises. 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 [3] and Web-CAT [21] require the student to submit a test suite, which is itself evaluated for completeness.

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 [12], Expresso [14], and Web-CAT aim to discourage bad programming practices such as unused variables or risky side effects. Web-CAT merges diagnostics from both Checkstyle and PMD [24] into a unified report, so the student can inspect his code with all the warning messages displayed line by line.

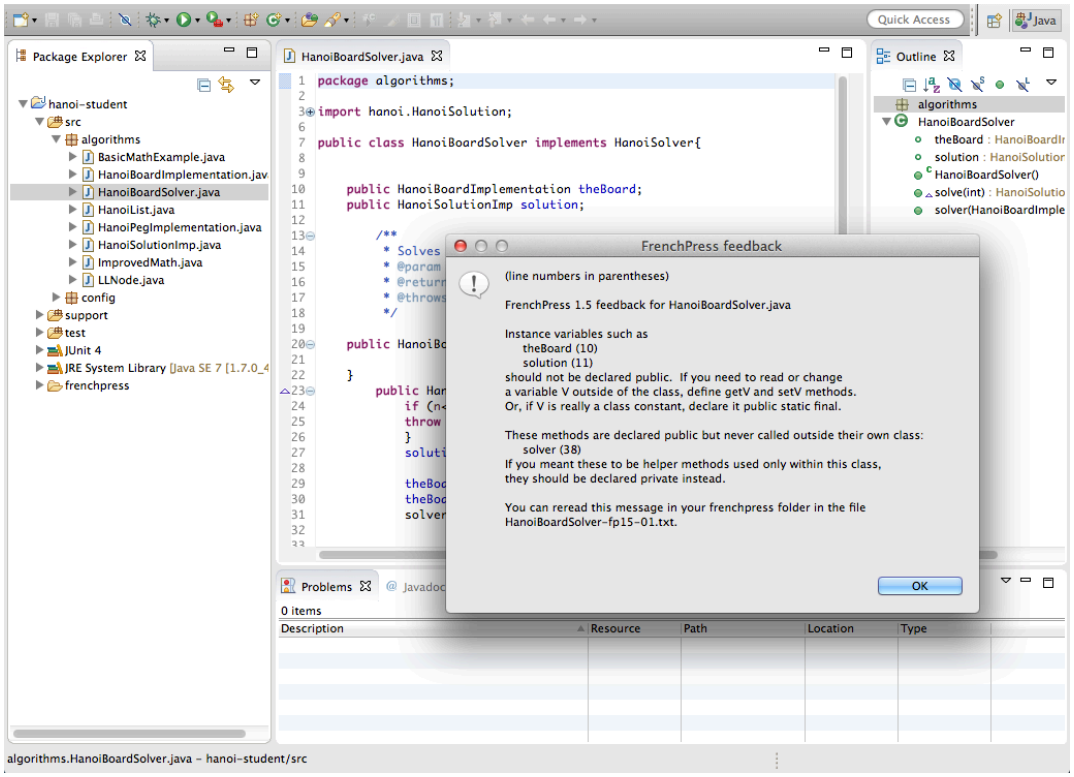

Fig. 1. FrenchPress feedback for a single `.java` file

TABLE I
AUTOMATED ASSESSMENT SYSTEMS FOR INTRODUCTORY PROGRAMMING COURSES

System	Origin	Refs	Languages	T	M	S
ASSYST	U Liverpool	[3]	Ada, C	×	×	
Automatic Marker	U Cape Town	[4]	Java	×		
BOSS	U Warwick	[5]	Java, Perl	×	×	
Ceilidh	U Nottingham	[6]	C, C++, Prolog, Z	×	×	×
CourseMarker	U Nottingham	[7], [8], [9]	C++, Java	×	×	×
EduComponents	U Magdeburg	[10]	Haskell, Lisp, Prolog, Python, Scheme	×		
ELP	Queensland U Tech	[11], [12], [13]	C#, Java	×	×	×
Expresso	Bryn Mawr Coll	[14]	Java			×
Java Critiquer	Northwestern	[15], [16]	HTML, Lisp, Java			×
HoGG	Rutgers	[17]	Java	×		
Scheme-Robo	Helsinki U Tech	[18]	Scheme	×		×
TRY	RIT	[19], [20]	Any	×		
Web-CAT	VA Tech	[21]	Java	×		×
WebToTeach	Brooklyn Coll	[22], [23]	Ada, C, C++, Fortran, Java, Pascal	×		

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

FrenchPress bridges the gap between existing pedagogic systems aimed at novices and static analysis tools geared toward professionals. Bug reports from industrial strength tools such as FindBugs ([25], [26]) and PMD can be intimidating for Java learners. These tools presuppose a familiarity with programming language vocabulary that is still out of reach for most CS2 students. FindBugs does not recognize many of the style flaws FrenchPress identifies, because it is looking for more subtle errors an experienced programmer might make (synchronization of threads, vulnerability to malicious code). FrenchPress 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. As students advance in the computer science curriculum, we expect them to outgrow the FrenchPress plug-in and graduate to professional analysis tools.

FrenchPress is similar in spirit to systems such as Stench Blossom [27] and JDeodorant ([28], [29], [30], [31]) that alert programmers to *code smells*, questionable program features that indicate the code should be refactored or redesigned.

TABLE II
JDEODORANT CODE SMELLS AND CORRESPONDING REFACTORINGS

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

TABLE III
FLUCTUATION IN CLASSROOM TRIAL PARTICIPATION

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

Stench Blossom offers programmers an interactive visualization that warns them of code smells as they are writing Java in Eclipse. It recognizes eight code smells: DATA CLUMPS, FEATURE ENVY, MESSAGE CHAIN, SWITCH STATEMENT, TYPECAST, INSTANCEOF, LONG METHOD, and LARGE CLASS. JDeodorant detects five categories of code smell and can automatically refactor the code to eliminate them (Table II). JDeodorant calculates and ranks multiple candidate refactorings. The user can select one, preview its effects, and have JDeodorant apply it to the program.

FrenchPress flaws can be considered code smells specific to advanced beginner Java programmers. The issues FrenchPress highlights for students are more localized than the smells identified by Stench Blossom and JDeodorant. FrenchPress's suggested repairs are much smaller in scope than JDeodorant's candidate refactorings. Unlike JDeodorant, our plug-in does not perform code modification. If the student accepts FrenchPress's recommended changes, he must make them himself.

With so many options for program analysis, the CS2 professor might want to combine FrenchPress with some other tool that covers different mistakes. Our plug-in relies heavily on Eclipse's rich repertoire of data structures and built-in functions. Incorporating FrenchPress feedback into a standalone static analysis system would require a complete reimplementation of the rules. On the other hand, it would be feasible to integrate FrenchPress with the Eclipse plug-ins for Checkstyle, FindBugs, JDeodorant, or PMD. We must avoid overwhelming the student with too many warnings. The author of an omnibus plug-in could define default rule sets catering to varying levels of programming sophistication, and/or provide a user-friendly dashboard so the student or the instructor could select the most appropriate diagnostics.

V. CLASSROOM TRIAL

In Fall 2014 we conducted a classroom trial in the undergraduate introduction to data structures and algorithms. The study was supervised by the UMass Amherst Institutional Review Board, which approved our research protocol including the informed consent form and survey questions. [32] and [33] were our guides for evaluation. We did not attempt a randomized controlled trial, due to the ethical difficulties of an intervention in which only half the class would have access to a new software tool. Students who volunteered to participate were rewarded with a small amount of extra credit toward their final grade. Students who chose not to participate could earn the same extra credit by completing a short writing task.

The study covered four programming assignments, projects 3–6 in the course numbering. We skipped the first two projects to allow time for the informed consent process. For each project, 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 due date for each assignment, the students responded to a short online user satisfaction survey. There was no overall survey at the end of the semester. We decided to administer multiple surveys for two reasons. First, we 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, we wanted to obtain a more specific evaluation of his experience on each program.

The number of study participants varied slightly from one assignment to the next, as a few students forgot to run the plug-in or skipped the survey for a particular project but then re-engaged on the following one (Table III). 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).

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

VI. FRENCHPRESS PROGRAM ARCHIVE

The first time the student runs the plug-in, it creates an archive folder in his project. The archive folder contains all the feedback files FrenchPress has produced for that project, and a `.jar` file of program history. Every time FrenchPress runs, it records a snapshot of all the `.java` files in the `src` folder at that moment (not just the file under analysis).

The program archive reveals how many students received suggestions for improvement and what if any changes they made to their program in light of the feedback. The left-hand bar plot of Figure 2 shows what percentage of students who ran FrenchPress received substantive feedback. By "substantive feedback" we mean any diagnostic message. Users who did not get any substantive feedback saw only a "Good work"

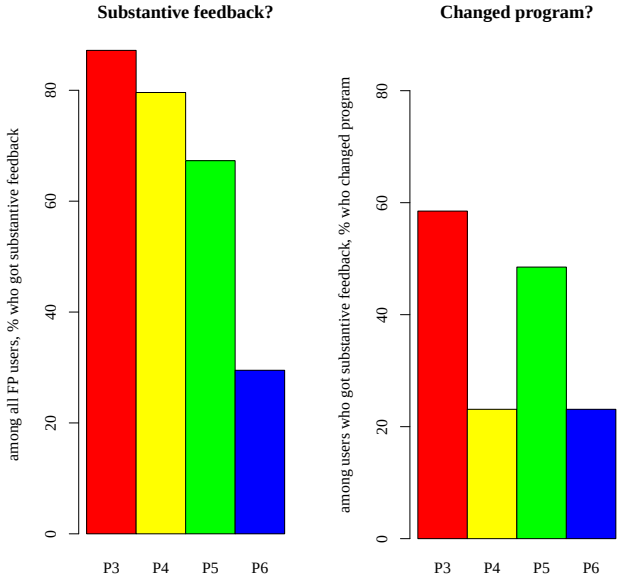

Fig. 2. Rates of substantive feedback and program modification

message. 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 files he wanted to analyze, the amount of diagnostic feedback he got depended in part on how zealous he was. The minimum required to earn extra credit was running FrenchPress on just one `.java` file for each project. We deliberately kept the criteria of participation low to avoid seeing students drop out of the study.

Regardless of student behavior, project 6 elicited little diagnostic feedback compared to the other assignments. Table IV shows the percentage of substantive feedback messages among all the messages generated by the plug-in on each assignment. P6 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 by the professors, and the coding experience gained over the semester. However, the drop in substantive feedback from P5 to P6 is so steep that we 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 P6 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 IV
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%

A. Short-term indicator of student learning

The right-hand bar plot of Figure 2 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 from P3 to P4 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. Student uptake of FrenchPress's suggestions might also have suffered because the diagnostic messages are not always on target. Some messages were false positives due to implementation errors, while other messages were misleading because the rule itself needed further refinement.

Rules 2 and 4 generated many messages that were not entirely appropriate for the student's program. Rule 2 is designed to identify class constants, but it also 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. 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.

Table V shows the off-target feedback as a percentage of all the substantive feedback from FrenchPress. Projects 4 and 6 have higher percentages of off-target messages and lower percentages of students who acted on the feedback. It is quite possible that the students were exercising good judgment in ignoring messages that did not make sense for their code.

TABLE V
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%

B. 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. We interpret frequency in relation to the length of code analyzed. For each student and each project, the feedback frequency of 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. We 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. Although the six rules fall into three categories (Section II), even rules of the same category address different stylistic issues; seeing feedback from one would not necessarily help the student avoid triggering the other.

To look for trends over the length of the study, we limited our attention to the 37 participants who used the plug-in on all four projects. Each of these students has four feedback frequency numbers for each diagnostic rule, reflecting his trajectory through the classroom trial with respect to that rule. The 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 3–5 show for each rule the distribution of student trajectories in these five categories. We separated out trajectories in which the student received feedback from the rule on only one of the four projects. Otherwise, a single positive feedback frequency at the beginning, at the end, or in the middle of the trial would produce a decreasing, increasing, or zigzag trajectory respectively. These distinctions are arbitrary if the student interacted with the rule on just one project.

Learning from feedback is only one factor that might influence student trajectories through the classroom trial. The nature of the problem posed makes some assignments more likely to trigger certain rules and not others. P6 produced very little substantive feedback, so almost all trajectories 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 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

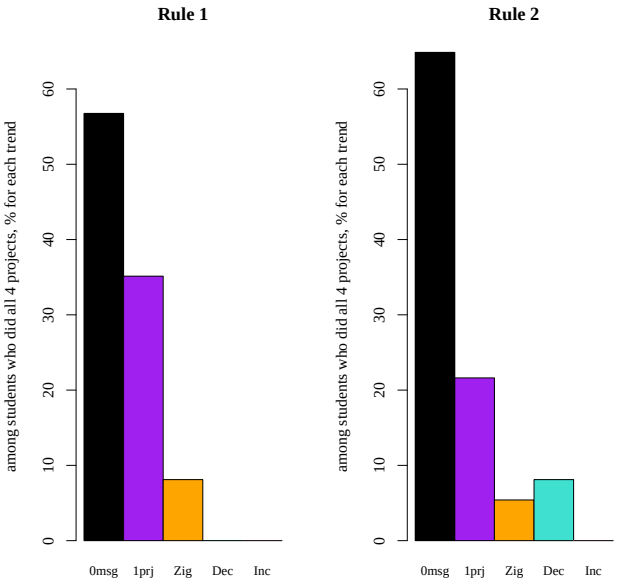

Fig. 3. Rules 1 and 2 feedback trends

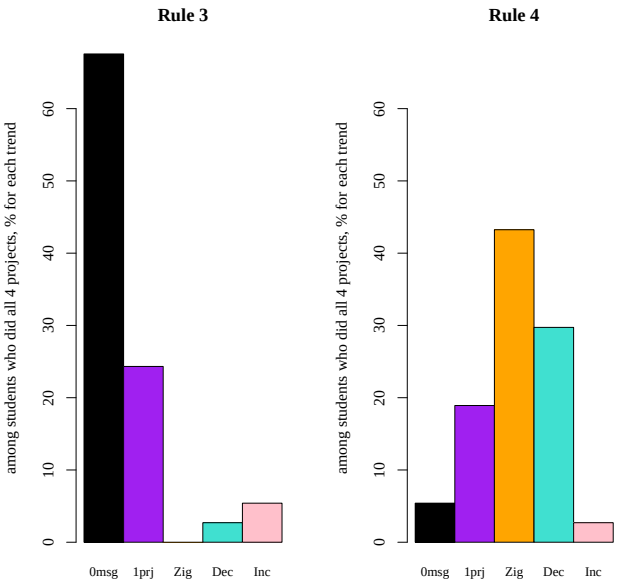

Fig. 4. Rules 3 and 4 feedback trends

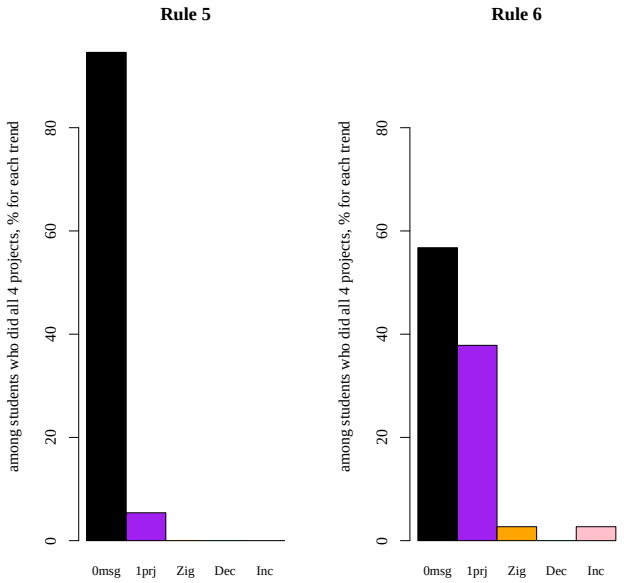

Fig. 5. Rules 5 and 6 feedback trends

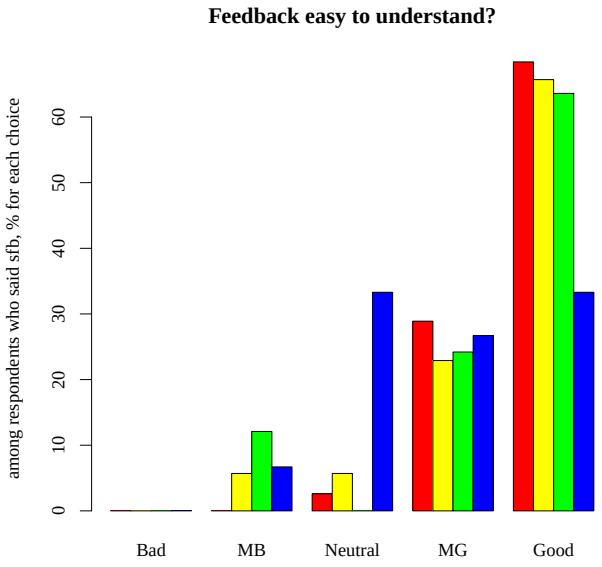

Fig. 6. Is feedback confusing or easy to understand?

generated more feedback messages than any other, as may be seen from the low percentage 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.

VII. User Satisfaction Survey Responses

An important objective of this research is to give advanced beginners 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, we included several questions on the user satisfaction survey 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 6–8 show the distribution of answers to these questions for the four projects. We standardized the three Likert-style scales so that each 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 and 7, the y-axis includes only those students who indicated on their survey that they had received substantive feedback (*sfb*) for that project. In Figure 8 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

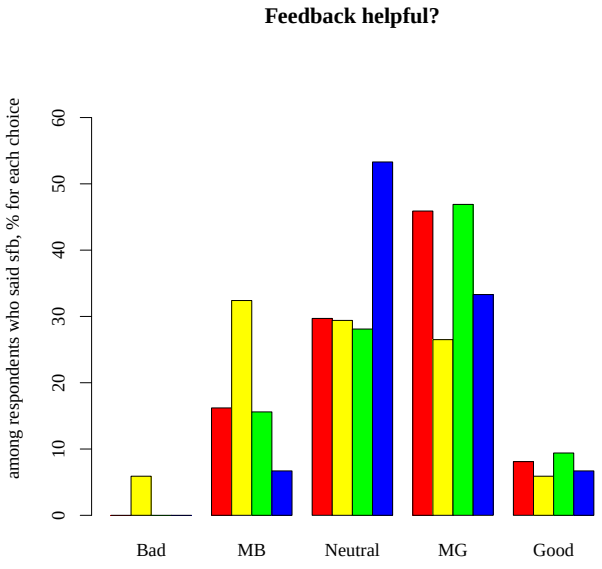

Fig. 7. Is feedback helpful or unhelpful?

feedback seems worse than the others, but the difference might be due to population size: only 13 students got substantive feedback on P6 (19 messages in total). As noted earlier, 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 7 and 8. Nonetheless, most students rated their overall satisfaction with the plug-in as neutral or positive.

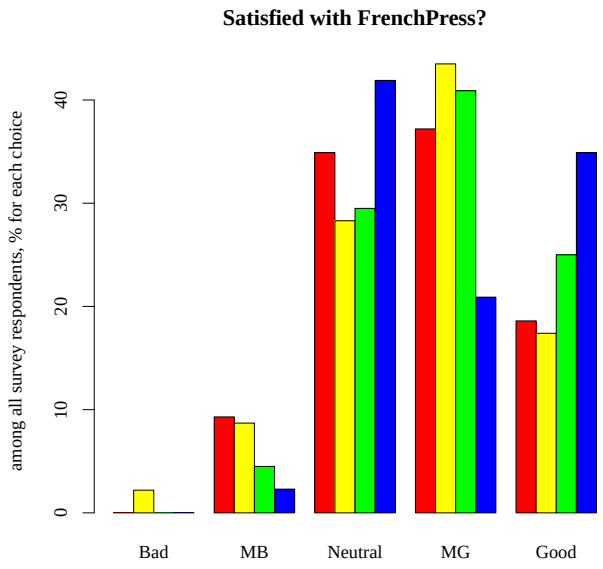

Fig. 8. Overall satisfaction with FrenchPress

VIII. Lessons Learned

Counting feedback messages would seem straightforward, but it requires careful attention to avoid treating duplicate messages as if they were new. Many students ran the plug-in over and over but made no changes to their code in between. The feedback files might look as though a rule has triggered many times, but it is really 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.

We should have made prompt software updates a requirement. Bug fixes and feature enhancements led to the release of new versions during the trial. Updating the plug-in demanded minimal effort, the students had only to select *Check for Updates* on the Eclipse *Help* menu. Nevertheless, some students were slow to update their software and three even reached the end of the study without updating at all. Students might have been more conscientious had we told them they would lose extra credit for submitting feedback files from older versions.

We built an Eclipse plug-in so we could give the Java learner suggestions while he is still working on his program, instead of delivering comments weeks after he has finished it. We hoped students would run the plug-in several times for the same class definition as they modified the file. However, examination of the archive folders 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. In future we will tell students to wait until their project is near completion before they get feedback from the plug-in.

IX. Conclusion

The Fall 2014 classroom trial of FrenchPress covered four CS2 assignments (P3–P6). The percentage of participants who got substantive feedback (not "Good work") declined steadily throughout the study, from a high of 87% on P3 to a low of 30% on P6. 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 multiple `.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 P3 and P5, but less than a quarter for P4 and P6. 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 P4 and P6.

Survey data show that over 87% of respondents found FrenchPress feedback to be moderately or very easy to understand for P3–P5, dropping to 60% on P6. However, the diagnostics were more intelligible than helpful. About 55% of users found the feedback rather or very helpful on P3 and P5, but the percentage drops to 32%–40% on P4 and P6, 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 P3 and P6 to a high of 66% on P5.

We also examined individual trajectories of subjects who ran the plug-in on all four projects, to see whether the frequency of feedback messages from each of the diagnostic rules increased, decreased, or fluctuated. 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 recognizable trend up or down.

A classroom trial of four programming assignments is not long enough to see slowly developing trends. The data collected so far do not support a firm conclusion about the plug-in's efficacy. The 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 users 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 computing the correct output. He might still internalize the coding recommendation and adhere to it on subsequent projects. A longer study would be required to discern the gradual evolution of student programming habits.

ACKNOWLEDGMENTS

We are grateful to D. A. M. Barrington and M. Corner for allowing us to collect data in their classrooms. J. E. B. Moss offered indispensable technical guidance and made substantial contributions to the plug-in implementation. Fruitful discussions with R. Fall and insightful comments from B. Lerner clarified the analysis.

REFERENCES

[1] Checkstyle 7.0, *http://checkstyle.sourceforge.net/*.

[2] H. Blau and J. E. B. Moss, "Frenchpress gives students automated feedback on Java program flaws," in *ITiCSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 15–20.

[3] D. Jackson and M. Usher, "Grading student programs using ASSYST," *SIGCSE Bull.*, vol. 29, no. 1, pp. 335–339, 1997.

[4] H. Suleman, "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: ACM, 2008, pp. 229–236.

[5] M. Joy, N. Griffiths, and R. Boyatt, "The BOSS online submission and assessment system," *Journal on Educational Resources in Computing*, vol. 5, no. 3, p. 2, 2005.

[6] S. D. Benford, E. K. Burke, E. Foxley, and C. A. Higgins, "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: ACM, 1995, pp. 176–182.

[7] C. Higgins, P. Symeonidis, and A. Tsintsifas, "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: ACM, 2002, pp. 46–50.

[8] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," *Journal on Educational Resources in Computing*, vol. 5, no. 3, p. 5, 2005.

[9] C. A. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas, "The CourseMarker CBA system: Improvements over Ceilidh," *Education and Information Technologies*, vol. 8, no. 3, pp. 287–304, 2003.

[10] M. Amelung, M. Piotrowski, and D. Rösner, "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: ACM, 2006, pp. 88–92.

[11] N. Truong, P. Bancroft, and P. Roe, "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: ACM, 2005, pp. 9–13.

[12] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' Java programs," in *ACE '04: Proceedings of the 6th Conference on Australasian Computing Education*. Darlinghurst, Australia: Australian Computer Society, Inc., 2004, pp. 317–325.

[13] ——, "Automated feedback for "fill in the gap" programming exercises," in *ACE '05: Proceedings of the 7th Australasian Conference on Computing education*. Darlinghurst, Australia: Australian Computer Society, Inc., 2005, pp. 117–126.

[14] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "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: ACM, 2003, pp. 153–156.

[15] L. Qiu and C. Riesbeck, "An incremental model for developing educational critiquing systems: Experiences with the Java Critiquer," *Journal of Interactive Learning Research*, vol. 19, no. 1, pp. 119–145, 2008.

[16] L. Qiu and C. K. Riesbeck, "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.

[17] D. S. Morris, "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*, vol. 3, 2003, pp. S3F 1–6.

[18] R. Saikkonen, L. Malmi, and A. Korhonen, "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: ACM, 2001, pp. 133–136.

[19] K. A. Reek, "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: ACM, 1989, pp. 112–116.

[20] ——, "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: ACM, 1996, pp. 125–129.

[21] S. H. Edwards, "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: ACM, 2003, pp. 148–155.

[22] D. Arnow and O. Barshay, "WebToTeach: An interactive focused programming exercise system," in *FIE 1999: 29th Annual ASEE/IEEE Frontiers in Education Conference*, vol. 1, 1999, pp. 12A9/39–12A9/44.

[23] S. Dexter, "On automated checking of Java applets," *Journal of Computing Sciences in Colleges*, vol. 15, no. 5, pp. 84–95, May 2000.

[24] PMD 5.5.0, *http://pmd.sourceforge.net/*.

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

[26] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[27] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *SOFTVIS '10: Proceedings of the 5th International Symposium on Software Visualization*. New York, NY, USA: ACM, 2010, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879216

[28] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *ICSM 2007: 23rd IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 519–520.

[29] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *CSMR 2008: 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 329–331.

[30] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 1037–1039. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985989

[31] JDeodorant, *http://marketplace.eclipse.org/content/jdeodorant*.

[32] J. Frechtling, "The 2002 user-friendly handbook for project evaluation," NSF 02-057, Arlington, VA, January 2002.

[33] ——, "The 2010 user-friendly handbook for project evaluation," *https://www.westat.com/sites/westat.com/files/2010UFHB.pdf*, December 2010.