Automated Program Analysis for Novice Programmers

Tim Blok
University of Twente
7511CW Enschede
The Netherlands
t.blok@student.utwente.nl

ABSTRACT

This paper describes how to adapt a static code analyzer to help novice programmers realize their mistakes , as current analyzers give feedback which is not very useful for novice programmers. An extension to PMD was made so that feedback messages appear which are easier to understand for novice programmers. Firstly, the current limitations of PMD were looked at by looking at their feedback of common programming mistakes. Afterwards, these limitations were filled in by creating custom rules. Lastly, the effectiveness of these rules was measured by noting the difference of errors returned by PMD on a number of projects using the default ruleset of PMD, and one with an extended ruleset.

Keywords

Programming; Code analysis; Teaching

1. INTRODUCTION

This paper describes how to adapt static code analysis tools (tools which check code for errors) to the needs of novice students. Improving in this context means that the feedback from such tools should directly point to the possible misunderstanding of certain programming concepts, instead of simply saying what may not compile correctly. This is done by changing the feedback messages.

The reason for using static code analysis tools is that these kind of analyzers are capable of giving feedback without compiling and running the code. Therefore, feedback can be given as quickly as possible to the code writer.

The feedback currently given by tools is meant for experienced developers. Because of this, the current errors refer to the specific compile or run-time problems that they encounter. However, these problems may not directly relate to the concepts which the students know. As a result, students may get stuck trying to fix a bug which is caused by something entirely different than the IDE, for example, currently gives as an error.

Additionally, as tools such as Coverity and FindBugs are meant for experienced developers, the focus of these tools lies in more advanced concepts, such as security, and assumes the user already knows how to program (fairly) well [2] [3]. This means that some errors are assumed to be known and assumed that the programmer does (barely) not make, which is not the case for novice students.

```
char reply;
char Y;

do {
    System.out.println("Do you want to continue?");
    reply = (char) System.in.read();
} while(reply != Y);
                 ^
  Variable 'Y' might not have been initialized
```

Figure 1. Example of a tool giving feedback

A static code analysis tool is extended with custom rules. This is done so that new and old versions' feedback can be compared easily, and to incorporate previous research (such as analyzing code itself) as much as possible. The choice is made to only look at Java code, as the University of Twente mainly uses Java in their programming courses. PMD is a java code analysis tool, and gives programmers the option to add their own rules (error detection with feedback messages) to it.

Afterwards, old code of programmers (early assignments, projects) is analyzed with the normal version of PMD, and then with the new, extended version meant for novice programmers. Code made by professionals is looked at as well, to see how many 'beginner mistakes' professionals still make in their open source projects.

Finally, these results are looked at and discussed to form a general opinion of changing PMD in a way to make it more understandable for novice students.

2. BACKGROUND

Static code analyzers are tools usually used to automatically check for programming errors in code and report their severity and location. [1] This can range from simple errors (using a variable which has not been initialized) to security flaws (hashing passwords without a salt). Often integrated in IDEs by default, but there are specialized programs that can be used in combination with IDEs to catch as many errors as possible (such as FindBugs). The other kind of code analyzers are dynamic code analyzers, these also look at code while it is executing. Static code analysis, however, does not execute a program to look for errors.

Whenever 'tools' are mentioned in this paper without further explanation, static code analyzer(s) are meant.

Feedback (messages) are the messages which appear when hovering over an error (in case of tools with a visual in-

25th *Twente Student Conference on IT* July 1st, 2016, Enschede, The Netherlands.

terface), or the messages listed after a tool has gone over code (tools without visual interface).

The rule set (or rules) of a tool is the list containing what kind of feedback messages should be given when a certain error is found. Rules can be customized to support personal preference [6].

Errors are programming mistakes made while coding. This can range from syntax errors which make code uncompilable to small errors which are mainly seen as bad practice.

Abstract Syntax Tree (AST) a tree representing syntactic objects in implementations of systems that manipulate programs, formulas, rules, etc. In this paper, ASTs will represent the source code of Java classes.

The founders (Seth Hallem, Dawson Engler, amongst others) of Coverity, Inc. made a paper [2] about the creation of a static code analyzer and building a company around making a static code analyzer. This reviews mostly the process of making such a tool and what problems arose while testing it with users.

Using this paper and the paper about general use, strengths and limitations mentioned in a paper written by Alexandru Bardas [1] more knowledge is gained about the general use and knowledge behind static code analysis tools.

FindBugs is a static code analyzer tool for Java, and is able to give insight in the way a tool looks for faulty code patterns. One of the creators of FindBugs has made a paper on the creation of FindBugs [3]. This paper contains all the different bug patterns which FindBugs looks for. This list could be used to know which errors students currently make are already able to be tracked without problem. This paper, however, along with the ones mentioned previously, do not mention the use of tools in a classroom.

Lastly, a paper by A. Sen reviews the usage of code analysis in class [6]. There are two points to look at, namely looking through assignments of students, and the usage of tools by the students themselves. There are no real results aside from saying that code analysis can help in the classroom, and that custom rules help with this. Custom rules are also mentioned (and used), however not explored.

3. RESEARCH

An example of something to be solved can be seen in Figure 1. If this were written by a student, it would point to the student failing to understand the difference between character variables and character literals. They meant to check if the input is the character 'Y', but instead compared it to the (undefined variable) Y. They addressed the error on the undefined field by declaring char Y. This results in the code in Figure 1, and the current feedback message is about a variable not being initialized, not giving the student enough information to know what they did wrong. What a student then could do, is do something which removes the error, such as initializing the variable with something useless (char y = 0; or char y = '0';). This removes the error, but now the input is being checked to 0 or '0', instead of the intended 'y'. This pattern of confusing literal characters with character variables is common among novice programmers. To counteract this, the error message should point towards the concept that the student may not understand, instead of the problem the compiler encountered.

Research purpose.

The goal of this paper is to see what different kinds of errors are made by students and to make an extension to a currently existing static code analysis tool. This is to make students learn from their errors by changing the feedback of such tools. This extension could be used by both students and their teachers. Students are able to immediately see what they did wrong. Teacher can use this tool to speed up the process of looking through assignments, as a broader range of errors are looked at.

Research question.

The question which relates to the goal above, is: How can feedback in the form of messages of static code analysis tools be changed to let novice students know what concepts of Java they do not understand? To answer this question, multiple sub-questions have to be answered first.

- What kind of information has to be relayed to a novice programmer to know what they did wrong?
- How can current code analysis tools be used to know what mistakes a novice programmer has made?
- What are the effects of changing feedback messages on the student's coding?

These questions also indicate the process of the research. Firstly look at what kind of information is useful for students in the first place. Afterwards, use the gained knowledge to extend a tool with new feedback.

Lastly, use this new version of the tool to measure the increased amount of errors on existing projects. When all this has been done, the main research question can be answered.

Research method.

Firstly, literature research is done to create a list of errors. These errors can be divided in 2 different categories: Errors that are already able to be identified using tools, and those which cannot be identified yet. Each error will then be one of two kinds depending on the category: if it is already identifiable, whether the feedback is sufficient enough, or if the error is not yet identifiable, whether it is possible to identify or not. which are mainly bad practice (bad code style). The example given in figure 1 is categorized into 'already identifiable', but does not yet have sufficient feedback.

The next step is to correlate certain ways of coding to known programming mistakes. Using figure 1 again, if a tool sees both the creation of a variable character with only one character as its name combined with a 'not initialized' error, this could mean that the student tries to use a character variable as a literal.

Afterwards, PMD is extended with custom rules to make sure the category 'unidentifiable errors' is as empty as can be. Additionally, some feedback messages which are currently given back may be changed to give the user more useful feedback.

Lastly, the categorized errors are looked at by going through code, and comparing the number of issues found by the standard tool and the extended tool. This is done on two sets of projects, namely projects made by novice programmers and projects made by professional programmers. The difference of additional errors found with the extended tool are compared to give a good indication on how effective the extension is.

When all this information is gathered, the research questions are able to get answered correctly.

4. STATIC ANALYSIS APPROACH

4.1 Examples of solutions

Some examples will help understand what this paper looks to accomplish. First, a small piece of code is shown. Then the current feedback of a standard tool (usually the IDE itself) is shown. Afterwards, a proposed way of changing the message is shown. Lastly, it is explained why this proposed feedback was chosen.

Example 1 (variables).

Code:

```
char q; return q;
```

Current feedback: Variable 'q' might not have been initialized.

Proposed feedback: Variable 'q' has no value. Are you sure you want to use a variable instead of a literal value?

Explanation: Similar to figure 1, a student might think 'char q' is the correct way to say that 'q' is the value of a character. Therefore 'return q' should return a character with value 'q'. However, this is incorrect, as q in the code is now simply a variable with the name q.

Example 2 (C++, syntax).

Code:

```
if(a = b){...}
```

Current feedback: Statement is always true.

Proposed feedback: If statement does not have a standard boolean operation. Did you mean 'a == b'?

Explanation: a = b is an assignment, which normally returns 1 (true). The student most likely meant a == b, as this is a normal boolean operation. It may be hard to discover that they used '=' instead of '==', since it could be that the student is not actively looking for this kind of error when debugging.

Example 3 (code style).

```
if(a == 0){a = 1;}if(a == 1){a = 3};
...if(a==13){return 2}
```

Current feedback: Nothing.

Proposed feedback: Many checks are made of the same variable. Try using a switch statement?

Explanation: While the code may be correct, it does not mean that it is clean. Teachers may have certain code styles that they want the students to follow. If switch statements have only recently been explained, it may not be obvious to the student that a switch statement would be perfect for this situation.

Example 4 (switching 'or' and 'and' statements).

```
if(string1 != null || string1.equals(string2)){...}
```

Current feedback: The variable 'string1' can only be null at this location. This error is given at the second occurrence of 'string1' in this line.

Proposed feedback: Null check followed by 'or'. Did you want to use an 'and' (&&) statement?

Explanation: This one is fairly simple, the student switched up the syntax for 'or' and 'and'. There is the possibility of the student not getting the concept of logic gates, this could also be addressed in the feedback.

From these examples one can see that the proposed way, in addition of saying what is wrong, also gives a way to fix it, and makes certain problems easier to understand. During the implementation more types of errors are looked at and researched to know what feedback is the most useful to give.

4.2 Tool choice

As mentioned before, PMD has been chosen as the tool to extend. However, this is far from the only tool available for java code analysis. The three most popular tools are FindBugs, PMD and Checkstyle, based off Google's results and suggestions given around the internet. The choice was made to evaluate these three tools and see which one would fit this research.

Checkstyle is a tool mostly made for correcting certain code styles. While not bad in itself, this research also aims to look at more problems in the code, not purely style.

FindBugs is a tool which looks at the java byte code and bases its detectors off of that [3]. This is very useful for detecting code patterns which may be problematic, but leaves little room to look for style issues, as variable names for example cannot easily be retrieved while checking the byte code.

Lastly, PMD is a good hybrid between the two previous tools. It utilizes a generated abstract syntax tree (AST) from the source code, which is then descended. With this, both code patterns and bad practices can be tracked.

The only downside which is shared by all tools is that code which cannot be parsed can also not be checked by tools. Therefore, bad code which is picked up by the IDE is not able to be given feedback by the tool.

Additionally, Eclipse is used as the IDE in which the errors are tested. This is because the University of Twente uses this IDE as the default IDE in their programming courses.

4.3 Limitations of PMD

After trying out creating custom rules in PMD, is has become apparent that there are still some flaws which can cause certain rules to be harder to implement.

The flaws of PMD lie mainly around creating rules rules which look at multiple classes or projects as a whole. In PMD, there is no method which is called after the entire class is analyzed using a certain rule. Therefore, if it is needed to know everything about a class, it is necessary to do this in the very first node of a class (ASTCompilationUnit) and then traverse the tree manually.

Another thing is that it is not possible to always know everything about another class (fields, methods, etc$\cdots$). While it is possible to save information after having gone through a class, there is no way to gain information of a class which is yet to be analyzed, while it is possible that this class is used in some way in the class which is currently analyzed.

4.4 False positives

For novice programming rules, false positives are not of great concern. This is because the current way of giving feedback is mainly suggesting changes to be made in code. For example, if someone creates a character c (not unusual), it does mean that they for certain do not know the difference between a variable and a literal. While 'c' is probably not a good choice of name, as it makes code harder to read and understand, if it is used properly there is not a big problem at hand. PMD allows the user to select errors and mark them as reviewed or remove the violation entirely, if they think the current error shown is not actually a problem.

4.5 Feedback messages

One part of helping the programmer is detecting errors, and another part is giving feedback back so that they know they did wrong in the first place, so the error can be avoided in the future. To make this feedback give all the information the programmer needs, the feedback consists of two or three elements, depending on the error.

Firstly, what part of the code generates an error. This typically consist of a line number (or, in Eclipse, a small arrow in the side column to indicate there is something wrong on this line) and an explanation of the error. Example: (line xy) A null-check is followed by a conditional OR (||).

Secondly, a suggestion is done to help fix the error. This could be one suggestion, or multiple. For example, if the same variable is often checked using if/else, the suggestion could be to replace the current structure with a switch statement. An example of multiple suggestions: if an instance variable is only used in one method, one suggestion could be to make that variable local to that one method. However, another suggestion could be to make the variable 'final', to indicate it being a constant.

Lastly, a source of information on the concept is given, if applicable to the error. As the University of Twente uses "Programming and object oriented design using Java" by Nino and Hosch [5], this book is used as the main source of information.

These elements were also looked at when testing the current tool feedback. If it is clear what part caused the error, what kind of error caused it, and the suggestions to fix it are correct, it means that the tool currently gives sufficient feedback.

5. RULES FOR NOVICE PROGRAMMERS

Now that the tool and all of its surroundings have been established, it is important to look at what rules should be changed or added. A list of 20 errors which are often made by beginning programmers was created and discussed by M. Hristova [4]. Some of these errors cause the file to be unparsable, such as misaligned parentheses. These errors are ignored, as the tool cannot look at unparsable code.

Additionally, known common programming errors by experts (teachers and student assistants) are added to the total list of errors as well. These have been gathered by talking to experts and noting what they have experienced while teaching students how to program.

The list of errors is as follows:

(A) Confusing the assignment operator (=) with the comparison operator (==).

(B) Use of == instead of .equals to compare strings.

(C) Unbalanced parentheses, curly brackets, square brackets and quotation marks, or using these different symbols interchangeably.

(D) Confusing short-circuit evaluators (&& and ||) with conventional logical operators (& and |).

(E) Incorrect use of semi-colon after an if, while or for statement.

(F) Wrong separators in for loops (using commas instead of semi-colons).

(G) Inserting the condition of an if statement within curly brackets instead of parentheses.

(H) Using keywords as method names or variable names.

(I) Invoking methods with wrong arguments (e.g. wrong types).

(J) Forgetting parentheses after a method call.

(K) Incorrect semicolon at the end of a method header.

(L) Getting greater than or equal/less than or equal wrong, i.e. using => or =< instead of >= and <=.

(M) Trying to invoke a non-static method as if it was static.

(N) A method that has a non-void return type is called and its return value ignored/discarded.

(O) Control flow can reach end of non-void method without returning.

(P) Including the types of parameters when invoking a method.

(Q) Incompatible types between method return and type of variable that the value is assigned to.

(R) Class claims to implement an interface, but does not implement all the required methods.

(S) Confusing character variables as literals

(T) Null check followed by or (||)

(U) Many if/else checks on the same variable.

(V) Instance variable not being used globally within the class. I.e, an instance variable can be reduced to a local variable.

(W) Switch statement does not contain a break.

(X) Switch statement without default case.

(Y) Out of array bounds by using <= instead of <.

Categorization.

As stated above, not all errors are detectable using PMD, as some errors may cause the code to be unparsable. For this paper, only the errors with certain properties are looked at.

The properties that the errors should have are:

(1) The code is still parsable

(2) The current tools (PMD or default Eclipse settings) already find the error and give sufficient feedback.

From this, four main categories can be deducted. Every error has been tested in the current tool environment, to see which error falls in which category.

- Unparsable (C, F, G, H, L)
- Not found (A, D, J, N, S, T, U, V, Y)

- Found, but has insufficient feedback (B, E, W, X)
- Found, and has sufficient feedback (I, K, M, O, P, Q, R).

In this paper, the rules in the second and third categories are looked at. Errors which make the source code unparsable are simply not detectable by PMD, as it relies on creating an AST of the source code. Errors which are already detected and give sufficient feedback have no need to be improved.

6. IMPLEMENTATION

PMD has an option to implement custom made rules and rulesets. A ruleset "Novice" is made, with custom rules in it. Eleven custom rules are created to cover as many rules as possible. To create rules which do not yet exist, an abstract class (AbstractJavaRule) is extended. This class is given as part of the PMD source code.

The extended class will override a visit() method. This method has two arguments: the current node of the AST it is at, and the current context of the class it is analyzing. Using this, actions can be done depending on the kind of node it is currently visiting. It can then inspect the structure of the tree around the given node, and add a violation if the structure looks like the structure when a certain error is made. After the analyzing is done, all the violations are shown.

The feedback message which is shown when hovering over an error is as concise as possible. The programmer can click on 'show details' to get more information about the error, such as an example of the error occurring, and how to fix it.

(A) Confusing the assignment operator (=) with the comparison operator (==)..

This rule checks for assignments being made inside structures that usually expect a boolean value, such as an if-statement. In Java, an assignment inside such a structure can be made only if the left-hand side of the assignment is a boolean.

It does the checking by going through the assignments made in a program. If the parent of this assignment is either an if, while or for statement, a violation is added.

The feedback message suggests to make sure you are not confusing = with ==.

(D) Confusing short-circuit evaluators (&& and ||) with conventional logical operators (& and |).

This rule checks for two booleans being compared using conventional logical operators. This is done by simply using the visit() method on a standard 'AndExpression' or 'InclusiveOrExpression', and making sure that both sides are of type boolean. If this is true, add a violation.

The feedback message suggests to make sure the user is not confusing operators with short-circuit evaluators.

(J) Forgetting parentheses after a method call..

This rule has not been implemented, as there is ambiguity while checking. Maybe the intention was to actually do something with a field which happens to have the same name as a method (which may be wrong for other reasons).

Additionally, one would have to go over all the classes and gather all the information regarding methods and variables to make sure the error is actually forgetting parentheses, instead of something else. This can prove troublesome, as PMD analyzing is done on a class-by-class basis, thereby not guaranteeing that knowledge of some other classes is available, as explained in section 4.3.

(N) A method that has a non-void return type is called and its return value ignored or discarded..

This rule has currently not been implemented, for the same reason as the last rule for not implementing rule J.

Furthermore, it is possible that the intention of ignoring a return is to use the functionality of a method, rather than working with the returned value (such as the .put() method of a Map implementation).

(S) Confusing character variables as literals.

This rule is fairly straightforward, it simply looks for declarations of variables of type 'char' with only a single character as their name. This could mean that the user tries to compare something to the variable's name, instead of its value (see example 1).

(T) Null check followed by OR (||).

This rule checks for null checks on an object which are followed by an OR (||), and then trying to use a method on said object. This is done by looking through conditional or expressions, and checking whether a null check is made. If this is true, it looks for the checked object to be used afterwards, which would lead to a NullPointerException.

The feedback suggests changing the || to &&.

(U) Many if/else checks on the same variable.

This rule looks for an if-statement with multiple else if-statements. If all these statements are looking at the same variable, this means it could be replaced with a switch statement.

The feedback suggests to replace the if/else structure to a switch.

(V) Instance variable not being used globally..

This rule is a bit more complex to create, as some assumptions have to be made. For example, when is a variable intended to be used as an instance variable? It has been decided to only look at private variables which are not final. This is because protected or public variables are likely intended to be used by other classes as well, and final variables are constants, therefore logical to create as an instance variable.

To detect instance variables that could be made local, everything has to be done as soon as a new class is starting to get analyzed, to make sure that all information is ready when it is needed. Firstly, all the instance variable declarations are gathered, and their names saved into a list. Afterwards, it goes through all the methods to see which variables are used. Lastly, it looks at the variables which only have been used in a single method, indicating it could be made local, and adds a violation.

The feedback suggests making the variable local to the only method it is used in, or make the variable final if it is intended as a constant.

(E) Incorrect use of semicolon after an if, while or for statement..

This rule checks for incorrect usage of a semicolon after an if, while or for statement. This is done by simply looking for the use of a semicolon after the condition part of an if, while or for statement instead of either a statement or an opening curly bracket.

The suggestion is to remove the semicolon.

(W)/(X) Switch statement does not contain a break/default case.

This rule already existed in PMD's default ruleset. However, it did not give a sufficient feedback message.

The feedback message now suggests adding a break and/or default case, depending on what is missing.

(Y) Out of array bounds by using <= instead of <.

This rule looks for less or equal than (<=) being used on size/length checks inside a if/while condition, as this has a good chance of throwing an ArrayIndexOutOfBoundsException. This is achieved by looking through the less or equal checks being used in if statements or while statements, and looking at the check being done on a standalone .size() or .length (without '- 1').

The feedback message suggests changing the <= to a <.

7. RESULTS

As stated in the research method, multiple projects made by novice and professional programmers alike are analyzed using the standard ruleset, and an extended one with the custom rules mentioned in the previous section. The standard ruleset has the rules which were changed by the custom rules (B, W and X) disabled to still be able to see how much the 20 rules are actually made by novice programmers.

The novice projects are mostly finished assignments and final projects from the module 'Software systems' of the University of Twente. When students created these projects, they had about 8 to 9 weeks of Java experience. These projects and assignments all had a requirement to be compilable, therefore some rules may not be effective on these projects.

The professional code is taken from six parts of the 'org.apache.commons.codec' library: .codec itself, .binary, .digest, .language, language.bm and .net.

Table 1. Number of errors found per project per ruleset

Project	Novice rules	Standard	Lines of code
Novice Errors/line	592 0.0066	59462 0.67	89056
Professional Errors/line	16 0.0025	3679 0.57	6485

For the results of novice programmers code, 24 projects (mixed assignments and projects) were tested with the default ruleset minus the rules mentioned in the above section, and with solely the extended ruleset.

7.1 Discussion

It was expected that the rules which looked at errors that made the code non-compilable were not going to come up. This is because the projects currently tested were all either finished projects, or assignments which were signed off (and therefore correct). For the same reason, the errors which change the entire behavior of the program (A, B, Y for example) are not as common either.

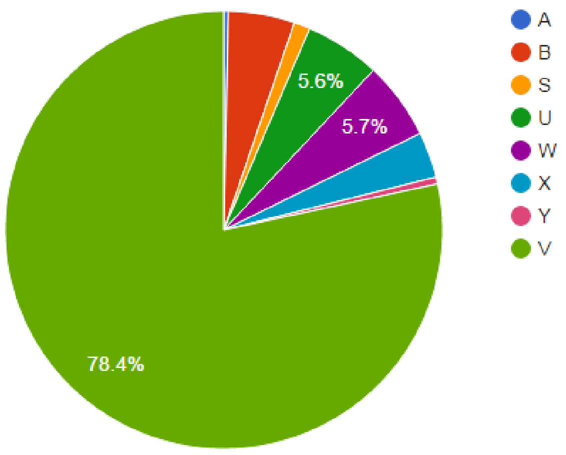

Figure 2. The percentage of rules occurring in the total set of projects

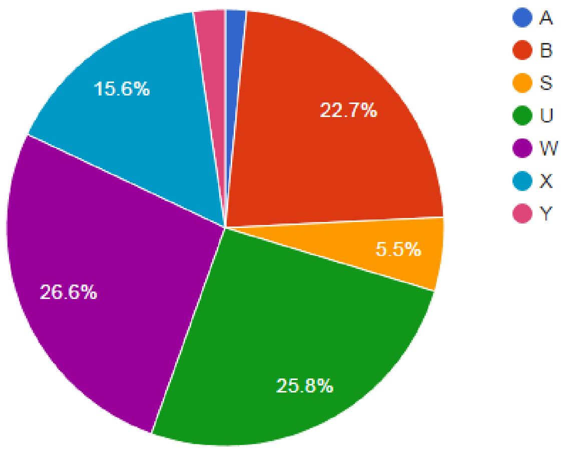

Figure 3. The percentage of rules occurring in the total set of projects, without rule V.

The prevalence of rule V (instance variable can be made local) is interesting to look at (see Figure 7, as it was not expected that this rule would find this amount of the total errors. An explanation for this would be that the types of code which cause certain errors (switches for example) are simply scarcely used compared to the creation and usage of variables. It could also mean that the concept of the other errors are generally understood better, or that the consequences of misusing instance variables are not as obvious as the other errors.

When leaving V out, a more balanced outcome of given errors can be noticed (see Figure 7). This figure also shows that the errors A and Y are rarely made. An explanation for this would be that when making such errors, the program's behavior changes considerably. Most of the given projects were working pieces of software, therefore not containing many behavior-changing bugs.

Another thing to look at, is the amount of errors generated with the custom ruleset compared to the standard ruleset. As said, around 1% of the total current errors are given by the extended part of the ruleset. This is a bit less than expected, as the custom ruleset has about 3.4% the number of rules compared to the total ruleset. This could be because of the sheer amount of errors the LawOfDemeter rule and other rules which have little impact on the code itself (such as CommentRequired at constants) return. These rules, while good to have in general, were not necessary or even graded on the projects themselves. After further testing, it seemed that on average more than 64,7% of the errors were generated by these rules. After disabling these rules, the final amount of extra errors generated by the extended ruleset amounted to around 2.8%, coming close the the expected amount of additional errors.

The weak points of this testing is that the testing was done on projects and assignments which were already finished, meaning that they were compilable. The best test scenario would be to use this tool in a live environment, i.e. while the students are actually being busy making their projects or assignments. Then all errors which would lead to unexpected events, such as using '==' to compare instead of '.equals()', would perhaps occur more often.

As seen by the data in Table 1, the new ruleset returns nearly no errors compared to the standard ruleset. This strengthens the point that the created ruleset has been mainly created for novice programmers, since the professional programmers (in this case, the Apache Commons programmers) barely make these mistakes anymore. Also, from the six professional projects, only two contained errors from the novice ruleset, as opposed to the novice project, where every project had errors generated by the novice ruleset.

8. CONCLUSIONS AND FUTURE WORK

With this new ruleset, students can quickly know what kind of errors they are making and how to fix them. Student assistants can use this as well to quickly look over the errors of a code file when having to grade work. However, this is only useful when the students do not get to use PMD.

The research questions have been answered throughout this paper. The way to relay information to a novice programmer is by using the feedback messages currently implemented by PMD, and fill them with information. This information contained three parts: what caused the error, what could be done to fix it, and where more formal information about the concept behind the error could be found.

The current analysis tool (PMD) was able to be extended using java classes to traverse an AST in which the structure of source code was contained. This structure was then used to locate certain patterns which may indicate the existence of an error.

The effects of the new ruleset, while not tested, is that the student is further educated while they are programming. How effective this education is was not able to be tested, however, and will have to be done in a future research.

This paper was not able to analyze all the errors presented, so the custom ruleset can still be extended. Other misunderstandings of concepts, such as parameters or control flow, have not yet been touched on either because of time restraints. And there are of course other errors not discussed in this paper where rules could be made for.

Future research can look at adding more rules to spot more errors, or try to test the new ruleset in a real environment, to see how often the errors currently not seen often appear.

9. REFERENCES

[1] A. G. Bardas et al. Static code analysis. *Journal of Information Systems & Operations Management*, 4(2):99–107, 2010.

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[3] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[4] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, volume 35, pages 153–156. ACM, 2003.

[5] J. Nino. *An introduction to programming and object-oriented design using JAVA*. John Wiley & Sons, Inc., 2007.

[6] A. Sen. Using code analysis tool in introductory programming class. *Issues in Information Systems*, 15(1), 2014.