A Comparison of Bug-Finding Tools

John R. Hott

April 2, 2007

Abstract

Software Engineering has shown us that it is much cheaper to find errors in code as early as possible, and that found early, they are much easier to fix [4]. This has led to the development of many compile-time tools for finding bugs early.

This paper compares some of the tools provided on sourceforge.net. Student projects were tested with each of the tools to see if the tool was helpful.

Of the tools discussed, PMD was the most powerful. It allows the creation of custom code rules. PMD also allows the code to be checked for duplicate code. Student projects from CS 301, Fall 2005, were tested with PMD and analyzed for duplicate code, and trends were scrutinized. PMD found many bugs in the code, showing its usefulness.

1 Introduction

Sourceforge.net currently lists 408 tools under the category of Quality Assurance. These tools range from bug finding tools to issue trackers for developers of large systems. Of these 408, only a few dealt specifically with finding bugs in Java source code. However, for most of these tools, the tool was not mature enough to test. Ultimately, there were four tools that met our requirements.

1. PMD: PMD is a static checker that parses the Java source code into an abstract syntax tree. It scans the abstract syntax tree looking for violations of the rules supplied. It provides helpful output, but also provides some false positives [11].

2. JCSC: JCSC is a style checker that determines if the code has followed all coding conventions described in its rules. This tool provides a helpful GUI interface that allows the configuration of custom coding standards [8]. This tool was not successful in finding bugs.

3. SISSy: SISSy, Structural Investigation of Software Systems, is an automated static analysis tool that determines the maintainability of object oriented code, written in Java, C++, or Delphi. It can detect violations of many common object oriented principles in code, and reports them [12]. We did not pursue SISSy further because it only writes its output to SQL databases and was unable to run on our systems.

4. Purity: Purity checks Java VM bytecode for pure and impure methods. It lists all the methods in the bytecode, their parameters, return values, and whether or not they are *pure*, as described in Section 5. It provides only one service, purity checking, but it performs this job well.

The next section describes the code suites used to exercise the tools. Then we discuss the use of PMD, JCSC, and Purity on these code suites.

2 Test Suite

2.1 Student Solitaire Projects

The major test suite for these tools was student solitaire projects from CS 301, Software Development, in fall 2005. Students, by the final project, implemented three distinct solitaire games written in Java. This was done by having the students create a framework in which they could implement new games without rewriting much code.

There were six projects that built up to the final project. They were:

1-3. Canfield: Students were required to build the model of the Model-View-Controller framework in projects 1 and 2. They also were required to implement the GUI in project 3.

4. Freecell: In this project, students were required to add a *Freecell* game to their project. Their goal was to also split their code into shared classes and classes unique to each game. This also arranged shared classes into a `solitaire` package.

5. Klondike: Students were required to add the *Klondike* game to their projects. Students were given a new class for this project, `WMSolitaire`, which provided the `main` function and started their games. They were allowed to modify this class. Project 5 created the `gamegui` package to contain view-controller classes of their projects. This package separated the GUI from the model, making testing of the model easier since the GUI cannot be systematically tested.

6. Extra Features: For project 6, students were required to implement a toolbar, unlimited undo, a timer, and a card counter into their solitaire game framework.

Eventually, students were required to have complete test coverage of their model classes, except the instructor-supplied `Deck.java`. This did not include the GUI classes, since GUI code contains elements that are drawn on the screen, such as cards, which cannot be tested in an automated manner with JUnit. Therefore, the GUI classes were separated into the `gamegui` package. The rest of the code was migrated into two packages, `solitaire` containing the model classes, and a tests package that includes JUnit test classes. Students were required to use JUnit [9] for their tests.

The requirement of complete test coverage of the model was intended to motivate the students to reduce duplicate code and increase code quality. If students wrote and designed their code well, it will require less test cases to ensure proper functioning. Also, reducing duplicate code removes the need to produce identical test cases for both copies. The students, however, did not have a tool such as PMD to detect duplicate code, so reducing the number of test cases was the only motivator for students to reduce code size.

The final grading criteria for the project included economy of code in non-test classes, good object-oriented code in non-test classes, and quality of code. The project specifically stated to avoid magic numbers and `if` and `switch` statements if possible, especially nested ones. It also required good use of abstract classes such as `Pile` and `Game`, good use of

the strategy pattern, and code reuse. Lastly, documentation and a FindBugs test with no serious violations reported were required. FindBugs is a bug finding tool in Java, based on *bug patterns*, code idioms that that are often errors [6].

2.2 Linked-List Implementation in Java

Purity required more time to test code, since it checks for the purity of methods and any methods invoked by them. This can produce the tracing of a large call-tree, requiring a longer runtime. Therefore a smaller suite was needed to exercise Purity to get a manageable runtime and output.

Purity was tested on student code from CS 241, Data Structures, from spring 2006. The project used was an implementation of a phone directory as a linked list. Since this project dealt mainly with manipulating memory, it was a perfect testbed for Purity.

In this project, students were required to implement a one-way linearly-linked list. Students were given an stripped-down class, `KWLinkedList.java`, with two private fields:

- `size`, the number of currently stored elements, and
- `head` which is a pointer to the head of the linked list.

The students were required to implement

- a `size` function to return the size of the list,
- an `indexOf` method which searches the list for an element and returns the index of that element in the list or -1 if not found,
- `get` and `set` methods to get and set elements, respectively, at a certain position in the list, and
- `add` and `remove` methods to add an object or remove an object, respectively, at a certain position.

Also required was the implementation of an iterator for `KWLinkedList.java`, with the usual functions `hasNext`, `next`, and `remove` defined by the iterator interface..

Students were not allowed to add any new instance variables, constructors, classes, or public methods to any of the classes. This prevents the students from accessing public fields, such as `int size`, to avoid the required functions, such as the `size()` function. Their code was also required to not produce any output in any method.

3 PMD

PMD (which is not an acronym) is a powerful static checker of Java source code. It comes from Tom Copeland, lead developer, as well as over 100 other contributors. PMD may be invoked in two ways. First, it can be run in its normal mode, which tests included code against the rules. The second mode allows for checking of duplicate code.

3.1 PMD Static Checker

PMD is built on the model of extensible rule sets. Its current version, 3.9, comes with 27 rulesets, ranging from 0 to 42 rules, each containing an average of 5 rules. Each ruleset checks the source code for specific problems or non-conformities. PMD's rulesets include checks for both semantic and syntactic problems. It can check for

- unused code,
- naming convention problems,
- excessive coupling of methods,
- code size problems,
- and problems in JUnit tests.

It even contains tests that some developers do not approve of in a *controversial* ruleset. This ruleset includes tests for: unnecessary constructors, null assignments, having at least one constructor, unnecessary parentheses, a field used by only one method (which could be made local), not using explicit scoping of classes and methods, and an analysis of dataflow anomalies [11].

In addition to the large collection of bundled rulesets, PMD allows the writing of custom rulesets to suit specific needs. These rulesets can be written in Java or in XML documents. Java rules are more complicated to write and were the original format for PMD rules. An XML parser was added to PMD to allow for *Xpath* rules to be included. These rules are written in XML and do not require quality assurance to understand the PMD Java code which implements the rules.

PMD uses a JavaCC parser which parses the Java source code into an Abstract Syntax Tree. Figure 1 contains a small sample of source code which PMD parses into the abstract syntax tree found in Figure 2. The Java example contains only one class, with two fields and one method, `main`. This 12 line source code is transformed into the 96 line abstract syntax tree in Figure 2.

We can see that the abstract syntax tree is rooted at a `CompilationUnit`. Under that entry, we can see `ClassOrInterfaceDeclaration`, which is a class declaration with the name *MyClass*. In the body of this class, the first entry is a field declaration, declared as *package private*, of a referenced type, class *Vector*. Those first 11 lines of the abstract syntax tree describe the first two lines of our sample source code.

PMD then browses the abstract syntax tree, and at each element in the tree checks the rulesets given for a rule that applies to that element. If a rule applies, PMD will execute the rule code for that element.

The authors provide a GUI that shows Java code parsed into the abstract syntax tree, giving a picture of the program. This GUI also allows the visualization of an element's position in the tree, giving aid in writing a custom rule. The most complicated task for writing rulesets is accessing the elements in the abstract syntax tree, since each element is a Java `class` type. This makes finding the elements of the tree, such as `Expression` difficult because the class type of the object must be found.

PMD's static checker can be executed over a Java source file, a directory, or a `.jar` file containing multiple source files. The rulesets are included using command line arguments. If no rulesets are specified, PMD runs the basic rules.

```
class MyClass {
   Vector v;
   Integer a;

   int main() {
      for (int i = 1; i < 25; i++) {
         int b = 5;
         int c = 56;
         b = b + 34 + c;
      }
   }
}
```

Figure 1: Sample Java code.

To test student code, PMD was invoked using the following command,

```
pmd.sh solitaire.jar text basic,imports,unusedcode,controversial,clone,coupling,design,braces,codesize
```

This includes the basic, import usage, unused code, coupling, brace usage, cloning, code size, and controversial rulesets in testing. The code from the `solitaire` package for each student was stored in `solitaire.jar`.

PMD was run in normal mode, using the rulesets listed above. GUI code and students' test cases were overlooked in testing for bugs since that code was cut for prior testing, and due to an oversight, was not reintroduced.

PMD found a significant number of problems in the code tested. These problems included some serious risks as well as false positives. The false positives are usually cautious behavior that should be considered. A complete listing of the errors found in Project 6 and number of occurrences of each are shown in Figure 3. This figure shows each of the error messages received from PMD in rows, with the columns representing the students, A through P. Each column shows the number of times that student received every warning message. The last two columns show the total number of times the warning appeared over all the students code and the total number of students that received each warning message at least once.

In this list, we see many errors and warnings. The most common warnings tended to be interesting, but mostly of little concern. From the table, we can see that the most common warnings given by PMD are calling `super()` in a constructor and avoiding using `if`s and `for`s without curly braces. These warnings were given for every student. The warning "Avoid using if statements without curly braces," for example, was produced for every student's code, and a total of 238 times. Student O's code only produced this warning 2 times, while Student K's code produced the warning 38 times.

PMD warns that a private field could be made final since that field was only initialized in its declaration or the constructor. Fifteen student projects produced this warning a total of 188 times.

We also see that 3 students imported, using `import`, code from their current package and 3 students imported files they did not use with Student C in both groups. We can also see that 4 students (A, E, N, and O) have deeply nested `if` statements.

Most of these warnings are important to cover, but not of much concern. They are good practices, but code will still work correctly if we do not follow these practices. However,

```
CompilationUnit
 TypeDeclaration
  ClassOrInterfaceDeclaration(MyClass)(class)
   ClassOrInterfaceBody
    ClassOrInterfaceBodyDeclaration
     FieldDeclaration:(package private)
      Type
       ReferenceType
        ClassOrInterfaceType:Vector
      VariableDeclarator
       VariableDeclaratorId:v
    ClassOrInterfaceBodyDeclaration
     FieldDeclaration:(package private)
      Type
       ReferenceType
        ClassOrInterfaceType:Integer
      VariableDeclarator
       VariableDeclaratorId:a
    ClassOrInterfaceBodyDeclaration
     MethodDeclaration:(package private)
      ResultType
       Type
        PrimitiveType:int
      MethodDeclarator:main
       FormalParameters
      Block
       BlockStatement
        Statement
         ForStatement
          ForInit
           LocalVariableDeclaration
            Type
             PrimitiveType:int
            VariableDeclarator
             VariableDeclaratorId:i
             VariableInitializer
              Expression
               PrimaryExpression
                PrimaryPrefix
                 Literal:1
          Expression
           RelationalExpression:<
            PrimaryExpression
             PrimaryPrefix
              Name:i
            PrimaryExpression
             PrimaryPrefix
              Literal:25
ForUpdate
 StatementExpressionList
  StatementExpression
   PostfixExpression:++
    PrimaryExpression
     PrimaryPrefix
      Name:i
Statement
 Block
  BlockStatement
   LocalVariableDeclaration
    Type
     PrimitiveType:int
    VariableDeclarator
     VariableDeclaratorId:b
     VariableInitializer
      Expression
       PrimaryExpression
        PrimaryPrefix
         Literal:5
  BlockStatement
   LocalVariableDeclaration
    Type
     PrimitiveType:int
    VariableDeclarator
     VariableDeclaratorId:c
     VariableInitializer
      Expression
       PrimaryExpression
        PrimaryPrefix
         Literal:56
  BlockStatement
   Statement
    StatementExpression
     PrimaryExpression
      PrimaryPrefix
       Name:b
     AssignmentOperator:=(simple)
     Expression
      AdditiveExpression:+
       PrimaryExpression
        PrimaryPrefix
         Name:b
       PrimaryExpression
        PrimaryPrefix
         Literal:34
       PrimaryExpression
        PrimaryPrefix
         Name:c
```

Figure 2: AST PMD generated for the Java code in Figure 1.

	Student																	
Error Message	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Total	# of Students with Error
It is a good practice to call super() in a constructor	4	3	6	16	4	3	6	3	8	2	5	2	6	4	1	3	76	16
Avoid using for statements without curly braces	18	18	2	3	20	20	10	2	17	11	1	1	18	14	9	10	174	16
Avoid using if statements without curly braces	16	18	9	12	6	13	22	18	24	3	38	6	11	26	2	14	238	16
Avoid using if...else statements without curly braces	24	23	24	8	36	21	4	13	5	4	16	3	30	4	16	10	241	16
Private field could be made final; it is only initialized in the declaration or constructor.	17	13	13	3	17	2	27	14	25	4	24		1	7	15	6	188	15
Avoid if (x != y) ..; else ..;	3	6	3		1	9	2	6	1	1	1	1	6	1	4	8	53	15
A method should have only one exit point, and that should be the last statement in the method	6	10	9	53	16	24	28	17	25		42	15	27	16	6	5	299	15
Each class should declare at least one constructor	2	1	9	1	8	6	10	12	13	6	2	1	12			8	91	14
Use explicit scoping instead of the default package private level	3	2	8	3	3	12	2	1	1			1	24	1	1		62	13
Assigning an Object to null is a code smell. Consider refactoring.	1		3	2	1	5	1		1	1	1		4	4		1	25	12
Perhaps could be replaced by a local variable.		3	4		8		2	4	12		5	1	1	5	2	4	51	12
Document empty method	2	3		3	3		1	1	4	1	1	1		3			23	11
These nested if statements could be combined	3	1		3	3			1				1		1	1	2	16	9
Avoid using while statements without curly braces	7				5				5	1			2	6	3	2	31	8
Avoid unnecessary constructors - the compiler will generate these for you	5	7				1	2				6	6		11	6		44	8
Avoid modifiers which are implied by the context					1		1	1			1		6	9	1	2	22	8
Avoid unnecessary comparisons in boolean expressions	11	3	2								2			10	2		30	6
When doing a String.toLowerCase()/toUpperCase() call, use a Locale		1					1	1	1		1					2	7	6
This statement may have some unnecessary parentheses					3	1				1			1	1		5	12	6
Avoid unnecessary if..then..else statements when returning a	2		5		5	1							3				16	5
Avoid unused local variables such as			1			4					3		3	1			12	5
Avoid unused private fields such as						1			2		3		1	1			8	5
Document empty constructor							2		2		1	2		1			8	5
Deeply nested if..then statements are hard to read	1				1									2	1		5	4
Avoid reassigning parameters such as	5				1				1		2						9	4
This final field could be made static		1							1	1		1					4	4
Overridable method called during object construction				6				3	1		3						13	4
The class has a Cyclomatic Complexity	4	1												1			6	3
The method has a Cyclomatic Complexity	7	1												1			9	3
Switch statements should have a default label	5		1									1					7	3
Use equals() to compare object references.	5								1					3			9	3
Use bitwise inversion to invert boolean values		1													1	1	3	3
Avoid unused imports such as			2			1					2						5	3
No need to import a type that's in the same package			1											2		1	4	3
Consider simply returning the value vs storing it in local variable								5					2	1			8	3
Possible unsafe assignment to a non-final static field in a constructor.										1	3		2				6	3
All methods are static. Consider using Singleton instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.	1				1												2	2
Avoid duplicate imports such as		1										1					2	2
An empty statement (semicolon) not part of a loop			2								1						3	2
Avoid unused constructor parameters such as						2							3				5	2
This abstract class does not have any abstract methods							1		1								2	2
Avoid unnecessary return statements									2				3				5	2
Avoid empty if statements											1				1		2	2
Too many fields												1		1			2	2
Avoid empty while statements									1								1	1
Avoid empty catch blocks										1							1	1

Figure 3: PMD results for project 6.

some examples are more interesting, in which errors can be clearly seen and in which PMD becomes very helpful to the programmer. However most of the interesting cases are those that PMD only finds in one student's work or that is found only a few times.

Student K's project was given the error "Avoid empty if statements," which seems like a harmless error, however, upon surveying his code, we find

```
if(!dpile.isEmpty());{
   tpile[num].push(dpile.pop());
   GameMove gmove = new GameMove(dpile, tpile[num]);
   this.addToUndoStack(gmove);
   this.addToStackBool(0);
   return true;
}
```

which is clearly an error, since this code is always executed, no matter the outcome of the if test.

Another less severe example comes from Student N's code, which received the warning message "too many fields" assigned to their Freecell game code. Upon inspection, we see the following list of fields in their `FreecellGame` class, which extends the `Game` class, seen on the left in Figure 4.

```
public FreecellHomecell homecell1 = new FreecellHomecell();
public FreecellHomecell homecell2 = new FreecellHomecell();
public FreecellHomecell homecell3 = new FreecellHomecell();
public FreecellHomecell homecell4 = new FreecellHomecell();

public FreecellTableau tableau1 = new FreecellTableau();
public FreecellTableau tableau2 = new FreecellTableau();
public FreecellTableau tableau3 = new FreecellTableau();
public FreecellTableau tableau4 = new FreecellTableau();
public FreecellTableau tableau5 = new FreecellTableau();
public FreecellTableau tableau6 = new FreecellTableau();
public FreecellTableau tableau7 = new FreecellTableau();
public FreecellTableau tableau8 = new FreecellTableau();

public FreecellPile freecell1 = new FreecellPile();
public FreecellPile freecell2 = new FreecellPile();
public FreecellPile freecell3 = new FreecellPile();
public FreecellPile freecell4 = new FreecellPile();
```

```
public Homecell homecells[4];
public Tableau tableaus[8];
public Pile freecells[4];

...
for (i = 0; i < 4; i++) {
    homecells[i] = new FreecellHomecell();
    freecells[i] = new FreecellPile();
}

for (i = 0; i < 8; i++)
    tableaus[i] = new FreecellTableau();
```

Figure 4: Left: Student N's FreecellTableau code. Right: Improved version of the same code snippet.

While this does not produce an error during runtime, it is a concern because it increases the complexity of the student's code. This method of declaration also does not allow the student to loop over the *homecells* and *tableaus*, as implementing the game would necessitate. The code could be correctly, and more easily be written as the code on the right above. This version creates the card piles in arrays and initializes them in loops. From later code in the Freecell game, seen in Figure 5, the student is required to include multiple identical statements in the constructor, such as the code on the left. The fields could, then, be declared as arrays or alternatively could be declared in the `Game` class, with the Freecell game class extending those it needs.

This example is also not correct, since the students were required to deal one card per pile until all cards were discarded. The code above deals all cards to a pile, then repeats per pile. An example of how this code could correctly and better be written is shown in the example on the right, which uses the arrays created in the last code segment and loops over them.

```java
for (int i = 0; i < 7; i++)
     tableau1.addCard(deck.getCard());

for (int i = 0; i < 7; i++)
     tableau2.addCard(deck.getCard());

for (int i = 0; i < 7; i++)
     tableau3.addCard(deck.getCard());

for (int i = 0; i < 7; i++)
     tableau4.addCard(deck.getCard());

for (int i = 0; i < 6; i++)
     tableau5.addCard(deck.getCard());

for (int i = 0; i < 6; i++)
     tableau6.addCard(deck.getCard());

for (int i = 0; i < 6; i++)
     tableau7.addCard(deck.getCard());

for (int i = 0; i < 6; i++)
     tableau8.addCard(deck.getCard());
```

```java
int count = 0;
for (int i = 0; i < 7; i++) {
    for (int j = 0; j < 8; j++) {
        tableaus[j].addCard(deck.getCard());
        count++;
        if (count == 52)
            return;
    }
}
```

Figure 5: Left: Student N's card dealing code. Right: Correct version of the same code snippet.

Therefore, we can see that PMD is a powerful static checker and can be very helpful in finding code *smells* [7], which are practices that usually indicate possible bugs. It can even discover bugs in relatively large projects.

Even though it produces a large volume of warnings to the programmer, some that the programmer may wish to overlook for ease of programming, it does identify some serious problems. All of the messages should be considered in finding problems.

3.2 Custom Rulesets

I have implemented two rulesets in Java for PMD. One provides type checking for PMD, and another allows a programmer to write simple text files describing methods or fields as pure or immutable. I will describe what *pure* methods are in the discussion of Purity in Section 5. An immutable field is a variable that should never be changed. A description of these rulesets can be found in the Appendix in Section 7.1.

Understanding how to write rulesets in Java gave a clear view of how PMD works. Writing rulesets required the knowledge of how PMD would step through the abstract syntax tree, and what methods it would execute. It also required understanding how to derive the type of a `class` object. When PMD steps through the abstract syntax tree, at each element it parses the rulesets given for a method that should be called. During the execution of that method, the entire abstract syntax tree is available. For most rules, it is necessary to parse through the children or direct parents of the current element of the tree to correctly execute the rule. This is tedious and difficult in most cases.

3.3 PMD for Duplicate Code

PMD can also check for duplicate code segments. Its duplicate code mode is invoked as

```
java -cp pmd-3.7.jar net.sourceforge.pmd.cpd.CPD --minimum-tokens 50 --files solitaire/ --format text
```

This tests the projects for duplicate code with a minimum of 50 tokens that match.

A token is any series of characters that form a logical item in a Java program such as `for`, `class`, or method name `helloWorld`. Comments are not included in token counts. Consistent with standard compiling technology [1], CPD also skips import statements, package statements, and semicolons [11].

This will single out true duplications in a project, however, if there are three or more sections of code duplicated, PMD will list each pair separately. Therefore, PMD may report code duplicated more than once many times in its output, once for each pair of duplicates. This is acceptable for our purposes, since it allowed us to identify if some of the repetitions were removed.

Students were required to have complete test coverage of their projects outside of the GUI classes. Therefore, duplicate code tests are appropriate to show whether students pared down their code in order to write fewer test cases. These tests would show that the class impacted the students. Because the project did not require test coverage of GUI code, student GUI and test code were not considered.

PMD counts duplications of code in both number of duplicate tokens and the number of lines duplicated. Using 50 as the minimum number of duplicate tokens, PMD was run over all the students' projects for project numbers 3, 4, 5, and 6, with the results given in Figure 6.

This table shows each student's results as a row, for students A through O, and the results for each project as columns. Each project is further broken down into duplicate lines and duplicate tokens. The averages and maximum duplicate lines and tokens are shown for each student, as are the number of students that have zero lines or tokens of duplicate code. The last column displays the number of duplicate tokens in project 6 subtracted by the student's maximum number of duplicate tokens over all four projects.

As an example, consider student G. In project 3, student G had 22 lines of duplicate code containing 51 tokens. By project 4, the student reduced their lines of duplicate code to 14, but increased the number of duplicate tokens to 57. Tokens are a better indication of duplicate code, since they do not include white space. For project 5, student G's duplicate code exploded to 402 tokens and 111 lines of code. This is most likely due to the new requirements of project 5. However, project 6, which required complete test coverage of the non-GUI code, shows student G reducing his duplicate code by 100 tokens, even though the number of duplicate lines increased by 8 to 65. Therefore, we can see that requiring complete test coverage was effective in getting this student to reduce the amount of duplicate code in his project. We can see similar results from many other students in this table.

Overall, we see from the right-most column that most students reduced their total number of duplicate tokens by project 6. Of the 15 students tested, 11 reduced their amounts of duplicate code or kept it consistent across the four projects. Only students E, H, J, and L significantly increased the total number of duplicate tokens by project 6. This clearly shows that requiring complete test coverage was effective at persuading students to clean up code and increase their code reuse, as was expected.

Therefore, we can see that PMD is an effective tool, both as a static checker and a way of measuring duplicate code. PMD is very effective at identifying problems and potential bugs in Java code, while at the same time it can be used to identify duplicate code and help reduce code complexity and the number of tests required for complete test coverage of that code.

	Proj 3		Proj 4		Proj 5		Proj 6		Project 6 - Max	
STUDENT	**Lines**	**Tokens**	**Lines**	**Tokens**	**Lines**	**Tokens**	**Lines**	**Tokens**	**Tokens**	
A	0	0	44	135	110	497	117	444		-53
B	159	334	265	665	272	724	288	757		33
C	0	0	0	0	32	78	0	0		-78
D	19	146	120	554	212	760	229	801		41
E	23	56	0	0	24	97	124	350		253
F	0	0	0	0	19	85	20	95		10
G	22	51	14	57	111	402	65	302		-100
H	64	200	0	0	37	163	117	510		310
I	33	121	22	51	0	0	0	0		-121
J	25	94	28	97	108	419	141	510		91
K	0	0	0	0	0	0	0	0		0
L	0	0	0	0	63	251	64	344		93
M	0	0	43	232	212	833	75	302		-531
N	0	0	0	0	12	109	4	55		-54
O	0	0	40	239	76	395	19	61		-334
Average	**23**	**67**	**38**	**135**	**86**	**321**	**84**	**302**	**Average:**	**-29**
Zeros	**8**	**8**	**7**	**7**	**2**	**2**	**3**	**3**	**Zero or below:**	**8**
									Less than 50:	**11**
Max	**159**	**334**	**265**	**665**	**272**	**833**	**288**	**801**	**Maximum:**	**310**
									Minimum:	**-531**

Figure 6: PMD duplicate code results.

4 JCSC

JCSC, the Java Coding Standard Checker, is a good static checker to ensure that code meets a required set of coding guidelines. This checker is a highly configurable tester of coding guidelines, but is not as effective at finding bugs. It will check for syntactic violations of the coding guidelines.

For quality assurance personnel, this tool provides an easy interface for ensuring coding standards. PMD allows for quality assurance to write rules in XML or Java that will enforce standards; however, JCSC contains a GUI that makes standards enforcing much easier. The GUI can be seen in Figure 7. In this screenshot, the Field tab is shown with rule options about fields of Java classes. Some of the examples from this screenshot include that only one field declaration is allowed per line, protected and package fields are allowed while public fields are not allowed, and rules about naming conventions of fields. The right pane shows a description and an example of the currently selected rule. A user of this GUI can simply edit the definition of the rule or use the yellow diamonds to turn the rule on or off, without having to know how to write code.

JCSC was run with the default coding standard rules. It performed as expected, listing many syntactic violations, but nothing as significant or as influential as PMD. The most common violations JCSC found were `\t` tab characters inserted by Eclipse which are not allowed, that spaces are required after a statement keyword, that lines were too long, and that required javadoc tags were missing for elements. All of these errors are listed in Figure 8, which shows error messages in the rows, with the number of instances of each message for students A through P in the columns. The last two columns show the total number of times the message was given across all students and the number of students who received that message.

From this figure, we can see that JCSC does not allow tabs, which all students used.

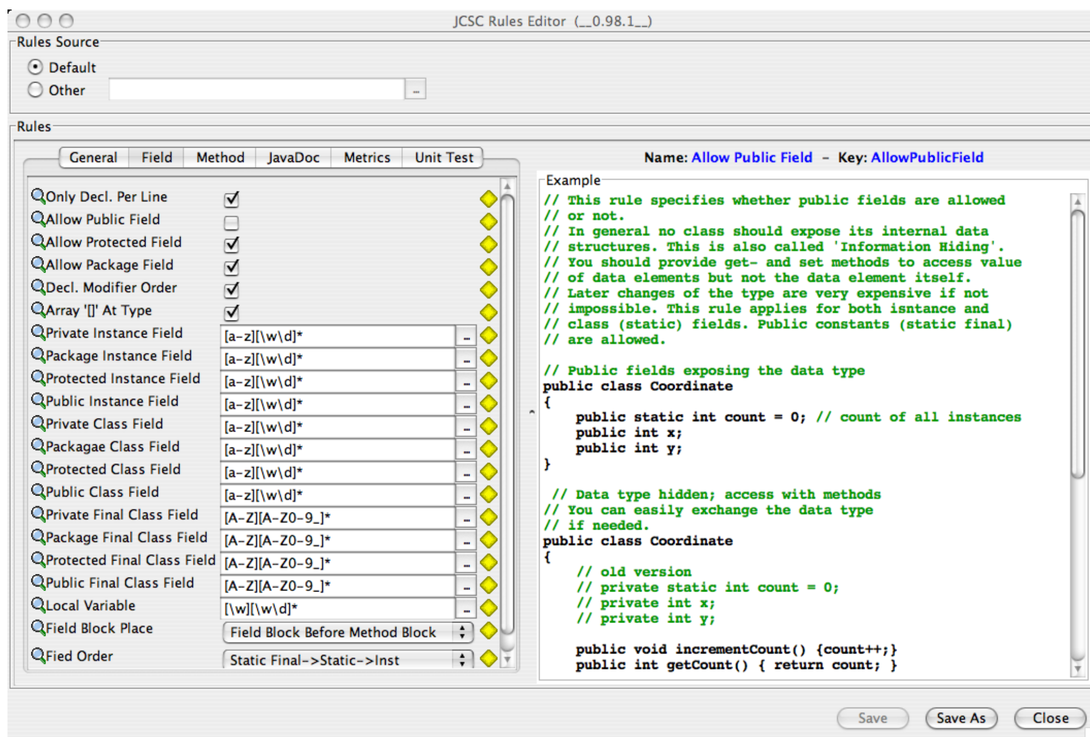

Figure 7: Configuration GUI for JCSC, found in jcsc.jar.

JCSC found 18,395 tabs, which were inserted by Eclipse automatically, and reported each as an error. Ten students declared more than one field per line, with student O repeating it five times. The violations identified by JCSC tend to be stylistic concerns or problems with JavaDoc documentation. For example, it did not report the empty `if` statement or give warning to the bug that was in Student K's code.

Overall, JCSC is not useful for finding bugs as a project is compiled, however, its easy-to-use interface allows it to be implemented into quality assurance tasks more easily than PMD, and it provides enough power to check for coding standards. This checker will find some minute bugs, but not the larger ones that PMD identifies. Therefore, the next step for a team using JCSC would be to incorporate some manner of testing using a more powerful static checker, such as PMD.

5 Purity

Purity is an interesting tool, and should prove helpful at finding methods and classes that change fields or objects that they should not. These changes can cause issues such as memory access problems due to bad pointers, bad indexing such as changing the size variable of an array in the `getSize()` method, and others.

Before going into Purity in detail, we must first discuss the meaning of *pure* and *impure*. A method is considered *pure* if, within the scope of that method, it does not change any field or call other functions that change fields created outside of that method. For example, the function `foo()` in Figure 9 is pure. Alternatively, if a method changes a field declared outside its scope, then it is considered *impure*. As an example of an *impure* method, consider the

JCSC Error	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Total	Count
Not allowed general import statement (import java.util.*)	3	1		3	1		1		6	3	2	2	1	3		1	27	12
is a tab character which is not allowed	2004	1736	1497	23	1591	31	549	1556	744	931	1173	664	1447	2107	1058	1284	18395	16
A space after statement keyword is mandatory	98	71	1	77		67	11	47			87	27	73	13	6	1	579	13
line is too long 0..80 characters are allowed	48	34	18	31	26	6	12	40	5	7	7	15	17	34	23	9	332	16
switch statement doesn block which is required	5		1									1					7	3
this complex loop expression is not allowed	23		3	9	6	1	7	3	15	5	10		18	13	8	11	132	14
JavaDocs Tags are in wrong order. Order	10	28	6		16		11	35	1		11	27		7		1	153	11
Not allowed general import statement (import javax.swing.*)	2	3	1	1	5	1	2		2	1	2		1	3	3	3	30	14
only 1 field declaration per line is allowed	1		2		3	3		4		4	1	1		1	5		25	10
method declaration	1																1	1
public method declaration JavaDoc does not provide the required tag	13	3	12	4	16	33	7	3	47	21	7		50	3	22	15	256	15
public method doesn't provide any JavaDoc	16	1	15	4	3	37	9		49	2			66	4	21	15	242	13
class Declaration JavaDoc does not provide the required tag	6	4	16		4		28		18	2			8	2	4	2	94	11
public constructor declaration does not provide any JavaDoc	4		6			8	2		8		6	2	14	1	1	1	53	11
public method declaration JavaDoc does not provide the required tag for all parameters	6		8	1	6	23	7	4	24	7	4		30	5	3	8	136	14
public abstract method declaration JavaDoc does not provide the required tag for all parameters	3						1	2	3								9	4
public abstract method declaration JavaDoc does not provide the required tag	8					1	2	1	6							1	19	6
public abstract method doesn't provide any JavaDoc	9					1	2		6								18	4
Not allowed general import statement (import java.awt.event.*)	1						1				1	1					4	4
public static method declaration JavaDoc does not provide the required tag for all parameters	1					1									1		3	3
public constructor declaration JavaDoc does not provide the required tag for all parameters	1	1	4	2		6			6		4	1	8	3	1	4	41	12
field declaration modifiers should have the following order[pub|pro|pri]->[abstract]->[static]->[final]->[transient]-		2				2	3	1	3		3			1			15	7
static final field decalration is in the wrong order. Order		2		7			3	1	5		3			1			22	7
field declaration [A-Z][A-Z0-9_]*		2		2			1	1	4	1	1			1			13	8
Not allowed general import statement (import gamegui.*)		1										1					2	2
field declaration [a-z][\w\d]*		1	2					2		1				2			8	5
protected field declaration does not have a JavaDoc			3	5		13		20		8		15		3	1	14	82	9
public static field declaration does not have a JavaDoc			1												1		2	2
field declaration having visibility are not allowed.			2	1		1	1	2		1		7		54	9		78	9
static field decalration is in the wrong order. Order			1			1							2		1		5	4
Not allowed general import statement (import java.awt.*)			1											1			2	2
public field declaration does not have a JavaDoc			1	1		1	1					7		52	8		71	7
public static final field declaration does not have a JavaDoc				47					1	2							50	3
protected static final field declaration does not have a JavaDoc				2						1							3	2
nested type in class must not be declared between fields and methods , you have to place them at the end of the class				1													1	1
method declaration modifiers should have the following order[pub|pro|pri]->[abstract]->[static]->[final]->[synchronized]->[native]->[strictfp]					6				2					8		2	18	4
is implicit for interface, it may be used but Java language spec says it is obsolete					1		1	1			1			1	1	1	7	7
public constructor declaration JavaDoc has more tags as parameters existing					1						1			1			3	3
class modifiers should have the following order [public]->[abstract]->[final]->[strictfp]					1									2			3	2
public static method doesn't provide any JavaDoc						1									2		3	2
static public final field declaration does not have a JavaDoc							3	1	3		3			1			11	5
public final field declaration does not have a JavaDoc								2		1				2			5	3
protected method declaration JavaDoc does not provide the required tag for all parameters									15								15	1
protected method declaration JavaDoc does not provide the required tag									10	6							16	2
protected method doesn't provide any JavaDoc									14								14	1
protected method declaration JavaDoc has more tags as parameters existing									1								1	1
abstract protected method declaration JavaDoc does not provide the required tag for all parameters									1								1	1
abstract protected method declaration JavaDoc does not provide the required tag									1								1	1
abstract protected method doesn't provide any JavaDoc									1								1	1
abstract public method declaration JavaDoc does not provide the required tag for all parameters									1					5			6	2
abstract public method doesn't provide any JavaDoc									1					7			8	2
JavaDoc tag									1		2			1			4	3
protected abstract method declaration JavaDoc does not provide the required tag for all parameters									2								2	1
protected abstract method declaration JavaDoc does not provide the required tag									2						2		4	2
protected abstract method doesn't provide any JavaDoc									2						2		4	2
empty catch block, this is normally an error										1							1	1
public method declaration JavaDoc has more tags as parameters existing											1			3			4	2
protected final field declaration does not have a JavaDoc												2					2	1
public abstract method declaration JavaDoc has more tags as parameters existing												1					1	1
interface Declaration JavaDoc does not provide the required tag													2		2	2	6	3
Not allowed general import statement (import solitaire.*)														2			2	1
abstract public method declaration JavaDoc does not provide the required tag														4			4	1
public static method declaration JavaDoc does not provide the required tag															1		1	1
fields in class must not be declared before and after the method definitions, you have to place them before the method definitions															1		1	1
protected static field declaration does not have a JavaDoc																1	1	1

Figure 8: JCSC results for project 6.

```
public class Sample {
    int i;
    String name;

    public int foo(int max) {
        int counter;
        int j;
        for (j = 0; j < max; j++)
            counter += max * j + i;
        return counter;

    public String bar(String input) {
        name = input;
        return name;
    }
}
```

Figure 9: Examples of pure and impure methods. *foo* is pure, while *bar* is impure.

function `bar()` in Figure 9. This method changes the value of `name`, which is not declared locally to `bar()`.

Purity uses the `.class` files to test code. It must know the path of the project and the class containing the main method in order to run. It systematically goes through every class and method and checks to see if that method is pure. It is helpful since it does not rely on JML or JavaDoc documentation to detect impure methods, but it systematically checks each one, and therefore each run of Purity will take some time.

In order to determine whether or not a method is pure, the method itself must be checked as well as any methods that are invoked from it. Purity traces and checks all methods called by the user and recursively any methods that those methods call. This includes any Java[1] library methods. Therefore, it can take a long time to run over a project due to the amount of code checked. For example, Purity checks 57 methods for the simple program in Figure 10 that imports `java.lang.System` and calls `System.out.println("test\n")`, including `fillInStackTrace()` from `Java.lang.VMThrowable`, `java.io.OutputStream`'s `flush()` method, and `java.lang.Integer`'s `toString()` method.

```
import java.lang.System;

public class MyClass {
        public static void main(String[] args) {
                System.out.println("test\n");
                return;
        }
}
```

Figure 10: Simple Java example.

Outside of this slowdown, Purity is very helpful. As the tool matures, currently version 0.09, we expect that it will be enhanced to produce less output about included files and use less time to check imported methods. One approach for the authors to improve the tool is to include a dictionary of Java library methods and classes and whether they are *pure* or *impure*. A sample output from one student's code can be seen in Figure 11.

Purity's output lists every class, method, and element that it has checked, and whether they are "PURE," meaning that it did not change any other elements that were not declared inside its scope, or impure, such as "NOT HEAP PURE" and a one-line description as to

[1]java.sun.com

```
KWLinkedList {
  public void [ro] KWLinkedList()
  PURE

  public boolean add([ro] java.lang.Object lv1_0)
  NOT HEAP PURE
        "this": mutation on this.(size|head)

  public java.lang.Object [ro] get(int lv1_0)
  PURE

  public int [ro] indexOf([ro] java.lang.Object lv1_0)
  PURE

  public java.util.Iterator [ro] iterator()
  PURE

  public java.lang.Object remove(int lv1_0)
  NOT HEAP PURE
        "this": mutation on this.(size|head)

}
```

Figure 11: Purity output for Java class KWLinkedList.

why. Figure 11 is an example of the Purity output from the KWLinkedList project from CS 241 described in Section 2.2. It shows that the method `public Object remove(int lv)` removes an object and is not pure, since it changes the `size` field and possibly the `head` field. Similarly, `public Object get(int lv)` returns an element at item *lv* and does not modify any fields; therefore, Purity denotes it as "PURE".

Purity can be very helpful. Consider the case of a method that should be pure but is discovered not to be. This is undoubtedly a bug in the user's program. For example, `get()` should return an element without changing any elements or variables external to the method. If it is identified as impure, there is a problem with this `get()` function.

Consider the case where a student's code is modified to introduce an error, `size++;`, in the `get()` function. Purity clearly identifies the function now as "NOT HEAP PURE".

```
public java.lang.Object get(int lv1_0)
NOT HEAP PURE
      "this": mutation on this.size
```

As seen from this example, the output is also comprehensive. Purity tells us that the public `get()` function takes an integer parameter and returns an Object. Furthermore, it is not pure because it modifies the `size` variable in the current class. Therefore, to fix our mistake, we should look into the method `get()` for a change to `this.size`.

As another example, Purity also helps the user find extraneous print statements left due to debugging or for other reasons. A call to `System.out.println()` changes the state of the system; therefore, we would expect Purity to tell us that a method containing a print statement would be impure. I changed the student's code above, replacing `size++` with `System.out.println()` to print out the current element, as would be used for debugging where `get` does not return the correct element. Purity returned the following result for `get`:

```
public java.lang.Object [ro] get(int lv1_0)
HEAP PURE BUT DOES IO
```

Purity told us that it is pure, that it does not change any of our fields, but it also correctly informed us that the method does IO. This would be a handy check for extraneous debugging information left in the code after it has been working correctly.

The test suite for Purity was different from that of PMD. The CS 301 projects were larger and the GUI code was removed from the source directories, making compilation complicated. Therefore, student code from the CS 241 Linked List project, described in Section 2.2, was checked with Purity. Each student's code was compiled under `javac`, and then checked with Purity.

However, Purity was not run over every student's code due to time limitations, but of the students that it was run on, it returns exactly what we would expect for the purity of methods. That is, we did not find any new problems in the student code from Purity. This is due to the fact that with such a small project, the students were required to have their projects working correctly, which means that purity issues must be fixed. We conjecture that these type of issues are usually the first to be fixed, or produce the most obvious errors upon runtime. However, introducing these problems, they are easily picked up by Purity, as the example above shows.

Since Purity is so narrowly focused, it will not be helpful for finding a broad variety of errors, such as those found by PMD, but it provides concrete answers to a method's purity much better than PMD and all other checkers discussed. I recommend that Purity be used, in addition to the use of other, more broad checkers.

6 Conclusion

Each of the three tools examined in this paper have different strengths, all useful. PMD was, however, the most powerful of the three examined. PMD identified out more real bugs than the other two in the code tested. It gives some false positives, but has a wider base of checks than the other two checkers combined.

JCSC was the least powerful of the three, since its main purpose is to check coding style. It performs this task as well as PMD, with the benefits of an easy-to-use GUI that the other two tools do not include. This tool is the most practical of the three tools at finding style problems, but not as powerful as the others in finding true bugs in the code.

Purity has a niche separate from the other two tools examined. We hypothesize Purity would be a good tool to have when writing the code, but once it is written and debugged, it may not be that helpful. This is because impurity issues are mostly caught in debugging, since they generally produce errors in the running of the program.

Therefore, for the tools discussed, I would recommend that they be used simultaneously. Purity is a powerful checker for impure methods, and would complement PMD's power to test for serious errors in the code. JCSC would complement both by giving quality assurance an easy setup for coding standards, without the need to learn PMD's Java or XML structure. Therefore, all three tools are useful.

The results of the study on CS 301 student projects showed that most students reduced the amount of duplicate code over the projects. We concluded that requiring complete test coverage of the `solitaire` package is effective at giving students an incentive to clean up their code [10].

7 Appendix

7.1 PMD Custom Rulesets

Two custom rulesets were written to expand the power of PMD. They took two basic directions: type checking and simple rule handling.

7.1.1 Type Checking

The first attempt was to construct a ruleset to check type issues. The typechecking ruleset was written to include explicit casts and implicit casts through function calls. The function call rule code is run every time the AST encounters a function declaration, and the parameters are researched. The explicit cast rule code is run every time the AST encounters a Cast expression. Both rules look back through the tree and obtain the original declaration of the variable and function call rule checks the original definition of the function. They are compared and PMD outputs a warning if one is needed.

This ruleset worked for all of the simple cases, but was too complex to run on larger test cases. However, this set is mainly used to give the programmer a heads up because the javac compiler will print messages to the user warning of loss of accuracy or unacceptable calling of functions with improper types. Downcasting is a normal procedure for Java, especially in older version when adding to a Collection class provided by Java. Therefore, the casting functionality is useful in large software that an updater might want to see how variables are being downcast, but not useful for the small programmer that must downcast to use Java's built in list and collection functionality.

7.1.2 Simple Rule Handling

A ruleset and supplementary material was written in Java to handle an external file as input to PMD. The file allows a user to declare methods as pure or fields that should not change in certain methods. A sample file can be seen in Figure 12. This example states that the method `main` in class `MyClass` should be found pure. Similarly, method `getInfo` in clas `MyClass` should not change the field `a`.

The ruleset reads in this sample file and checks at each class to see if there is a method that is declared pure and if so, checks that no variables created outside the method are changed within the method. Similarly, if a variable is declared unchanged in a method, the method is checked to see if that variable is assigned something different. If the variable has a function called on it, we cannot guarantee that the variable is unchanged, so we will note that a function has been called on it. The example produces output similar to that in Figure 13, which tells us that the method `getInfo` changed field `a` in line 14.

Simple rule files were written and this ruleset was run on sample student code from the Software Development course. This method correctly identified impure methods.

```
pure MyClass:main
pure MyClass:getData
pure MyClass:getInfo
unchanged MyClass:getInfo:a
unchanged MyClass:getInfo:c
```

Figure 12: Sample simple rule file to test the purity of methods using my ruleset.

```
jrhott% pmd.sh test.java text simplerulefile
test.java:14     Warning: Variable a changed in method MyClass:getInfo.
```

Figure 13: Sample PMD output using the simple rule file to test the purity of methods using my ruleset.

Bibliography

1. Alfred V. Aho, Ravi Sethi, and Jeffrey Ullman, *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.

2. Peter Amey, Praxis High Integrity Systems, "Yours faithfully: an everyday story of formality," Invited keynote address, in "Practical Elements of Safety", Proceedings of the Twelfth Safety-critical Systems Symposium, Birmingham, UK, February 2004. Copyright Spinger-Verlag 2004.

3. W Amme, N Dalton, J von Ronne, and M Franz, "SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form," PLDI 2001.

4. Fred Brooks, "No silver bullet: Essence and accidents of software engineering," IEEE Computer, 20(4):10-19, April 1987.

5. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe, "Extended static checking," Technical Report #159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

6. Findbugs. University of Maryland. `findbugs.sourceforge.net`, January 2007.

7. Martin Fowler, *Refactoring: Improving the design of existing code*. Addison-Wesley, 2000.

8. Ralph Jocham. JCSC, `jcsc.sourceforge.net`, 1999-2005.

9. JUnit, `junit.org`.

10. Robert E. Noonan and John R. Hott. "A course in software development." *Thirty-eighth SIGCSE Technical Symposium on Computer Science Education*, March 2007, pp. 135-139.

11. PMD, `pmd.sourceforge.net`, 2002-2006.

12. SISSy. `sissy.sourceforge.net`, 2006.