Analysis Tool Evaluation: PMD

Final Report
April 24, 2007

Allen Hsu
Somakala Jagannathan
Sajjad Mustehsan
Session Mwamufiya
Marc Novakouski

School of Computer Science
Carnegie Mellon University

Table of Contents

Introduction

1. Introduction to the PMD tool

PMD is an open-source, rule based, static source code analyzer that analyzes Java source code based on the evaluative rules that have been enabled during a given execution. The tool comes with a default set of rules which can be used to unearth common development mistakes such as having empty try-catch blocks, variables that are never used, objects that are unnecessary etc. A more complete list of these default rules is presented in part (3) below. Additionally, PMD also allows users to execute custom analyses by allowing them to develop new evaluative rules in a convenient manner. The tool comes with a relatively easy to user command line interface and at the same time, can also be integrated with popular development environments such as Eclipse. It is also a fact that PMD is quite popular in the Java development community and has also achieved substantial industry acceptance. Currently it only supports Java and as can be ascertained from the second section of this document, while there is room for improvement, the tool is generally quite effective and has proven itself as a useful static analysis tool for both large and small code bases.

2. Motivation for using PMD

Regardless of how experienced and talented a programmer may be, he/she is likely to make mistakes while writing programs. Unfortunately, while for smaller programs these mistakes are usually few and often have limited impact only; as programs grow, even small mistakes start having unexpectedly severe impacts. Consider a small program that creates a few unnecessary objects or variables that are never used in the program. In the realm of this smaller code base, the performance impact of these objects or variables will be negligible. However, in a much larger code base, these unnecessary objects will be large and many in number and can thus greatly impact the system's performance. Therefore, identifying and eliminating them beforehand is vital. This is where tools like PMD, which can statically analyze both large and small code bases, can be used to detect such critical issues well in advance. Therefore, given this usefulness of the tool, our team decided to evaluate the effectiveness of PMD for this report.

3. Brief overview of PMD internals

PMD operates in a manner that is very similar to conventional static analysis tools. This naturally means that the tool involves the generation and traversal of an abstract syntax tree. In summary, based on the information obtained from PMD project's official website at sourceforge.com, PMD's internal operation can be summarized as follows: (1) User supplies the location of the source code that has to be analyzed along with the rule or rules that he/she would like to execute; (2) the tool opens a data read stream to read in the source code and supplies it to a Java based code parser which in turn generates an Abstract Syntax Tree (AST); (3) the AST is then returned to PMD which in turn gives it to the "symbol table layer" that identifies scopes, declarations, and various usages; (4) if a particular rule (that is enabled) involves data flow analysis, the AST is given by PMD to the deterministic finite automaton (DFA) layer that in turn generates control flow graphs and data flow nodes; (5) With all this data obtained, each rule traverses the abstract syntax tree as needed and detects issues based on this traversal, rules can also

utilize symbol tables and nodes within the generated DFA; (6) the issues identified in step 5 are then printed out to the console or an associated file in a number of different formats.

There are three ways in which PMD can be used: as a command line, an Eclipse plugin, or an Ant target element. We focused our analysis and report on the tool being used from the command line. As an Eclipse plugin, the plugin comes with a PMD perspective (refer to the image in Appendix A). In the Package Explorer, the files with violations are marked with error marks, but those error marks are a bit confusing because they are identical to compilation error marks. In the source editors, the violations are shown with markers. There is a Violation Overview window that is meant to provide a summary of violations, and it also provides the ability to toggle severity levels showed in the view; however, this functionality doesn't seem to be working yet. As an Ant target element, Ant automates the running of PMD, pretty similarly to what a batch file would do.

4. PMD Results Analysis

While PMD supports custom rules, it comes with a set of 22 default rule based analyses. Each of these rules covers a different aspect of the Java source code and unveils issues surrounding that aspect. In general, these rules look for security issues (rule *sunsecure* is an example of this), performance/size issues (rules *optimizations*, *unusedcode*, *coupling,* and *codesize* are examples of this), good practices (rules *naming, design*, *import*, *basic*, *finalizers*, *braces*, *logging*, *strictexception*, and *javabeans*) and correctness related issues (rule clone is an example of this). This section covers analysis of PMD results from two view points. Quantitative analysis looks at the rulesets and the results of the rulesets on different projects. Qualitative analysis looks at usability, performance, scalability, modifiability, documentation, automation and the limitations of the tool. The projects considered for the analysis are as follows

- JBoss Application Server
- Eclipse BIRT
- Struts 2
- Applications based on RCP (the studio project is going to use RCP)
    - jLibrary
    - PaperDog
    - Nomad PIM
- Crystal3
- ZEN prototype code

4.1 Rulesets Analysis

PMD uses rulesets to evaluate code. Our approach was to evaluate each qualitatively to identify how useful they are. The rulesets and the results observed are listed below:

- **Basic (rulesets/basic.xml)** – These are basic rules which need to be followed. Across BIRT, Nomad PIM, jLibrary and PaperDog the following conditions were observed which we caught by PMD and which need to be addressed
    - `Catch` blocks shouldn't be empty,
    - Override `hashCode()` anytime `equals()` is overridden
    - Nested if statements should be avoided

- If statements evaluating to only true or only false should be avoided
- Avoid temporary variables when converting from primitive datatype to String
- Empty statements in loops should be avoided
- Avoid instantiating data type values which have existing constants defined for ex. instantiating Boolean type with constructor, instead of using Boolean.true, instantiating BigDecimal (0) instead of using BigDecimal.ZERO and so on
- Final modifier in a final class is redundant
- Overridden methods contain only call to super() and nothing else.
- Avoid unnecessary return statements
- Do not start a literal by 0 unless it's an octal value

There are some false positives in the above cases. In many cases, catch blocks were known to the developer and contained comments explaining why the exception was applicable or not. For final modifiers, the code had final modifiers for all the methods inside the final class and each line was shown as a warning. This though can be avoided is not an error. Some methods in classes were overridden but did not have any special task so it just had a call to super. This was flagged as error in the PMD results but this again does not cause erroneous behavior. 0 was used to denote octal values, so that also turned out to be a false positive.

Overall this ruleset is useful and because it catches basic errors which will be missed during reviews, inspection or testing. Hence we will be using this ruleset for analysis during studio.

- **Naming (rulesets/naming.xml)** – This ruleset tests for the standard Java naming conventions. Some typical errors which were observed during this analysis were
  - Abstract classes should have the name AbstractXXX
  - Variable names should not be too short
  - Field names matching class names lead to confusion
  - Variable names and method names should not be too long or too short
  - Variable names which are not constant should not contain "_" (underscore)
  - Class names should begin with an uppercase letter, method and field names should begin with a lowercase letter
  - The field name indicates a constant but its modifiers do not

  The code is strictly checked against Java naming conventions. This resulted in a lot of false positives. The results for all the four projects were filled with the entries about variable names being too short or method names being too short or too long.

  The one error message which said "The field name indicates a constant but its modifiers do not" sounded to be a valid error situation but on analyzing the code it was found that a variable name was completely in uppercase and had a underscore character which according to Java naming convention is to be used only for constants. Hence it was not an error in code but an error in naming.

  Considering that there will be 5 developers in the project and it is not cost effective (both time and effort based) to insist on strictly following naming conventions or correcting the errors which this ruleset throws, it might not be relevant to use this ruleset. Instead it would be better to define a coding standard for basic rules like class names and method names and each developer can use this while writing code and a reviewer can look for deviations during code review. We will not be considering this ruleset.

- **Unused code (rulesets/unusedcode.xml)** – This ruleset checks for unused code in the project. The following are typical error messages which occurred for the projects considered
    - Private fields and local variables that are never read,
    - Private methods that are never called
    - Unused method parameters and constructor parameters
    - Unreachable statements,

  With the widespread use of IDEs for Java like Eclipse and JBuilder, this ruleset is not very useful. For example in Eclipse, unused fields, variables and private methods are automatically highlighted with warning messages. Unused method parameters and constructor parameters are not enabled by default in Eclipse but settings can be changed to ensure that this check is included. Unreachable statements are treated as errors in Eclipse and the editor will not allow compilation till the error has been fixed. Hence this ruleset is not necessary.

- **Design (rulesets/design.xml)** – This ruleset covered various good design principles. The errors observed in this case were many. Few important ones are listed below
    - switch statements should have `default` blocks,
    - if conditions of the form a!=b should be avoided
    - private fields which are initialized only in the constructor and are not modified anywhere else should be made constant (final)
    - deeply nested `if` blocks should be avoided,
    - parameters should not be reassigned,
    - replace calls like size() == 0 with isEmpty()
    - overridable methods should not be called in the constructor
    - if all methods in a class are static then the class can be converted to Singleton
    - Caught exceptions should not be rethrown as the stack trace may be lost.
    - unnecessary comparisons in Boolean expressions should be avoided
    - equals() should be used for object comparison
    - it is better to use block level synchronization than method level synchronization
    - literals should be used first in string comparisons
    - Values can be returned directly instead of assigning them to temporary variables and
    - resources like connections should be closed after use

  This rulesets identifies many conditions which need to be cleared in the code. This cannot be covered exhaustively using manual reviews or inspection. There is one suggestion for conversion to Singleton when static methods are used, this may or may not be used based on the architecture. Other than this, all conditions highlighted by the ruleset needs to be corrected. Design rulesets based on the data gathered on all the projects do not have instances of false positives. This ruleset is useful for analysis.

- **Import statements (rulesets/imports.xml)** – This ruleset checks for minor issues with import statements. The following were the three conditions observed across all packages
    - Avoid duplicate imports
    - There is no need to import a class which resides in the same package
    - Avoid unused imports

Using IDEs makes this ruleset redundant. Eclipse has an option to reorganize imports. This automatically corrects imports, removes duplicate and unused imports, removes imports of the type java.io.* and includes individual classes. Hence this ruleset need not be used.

- **JUnit tests (rulesets/junit.xml)** – This ruleset looks for specific issues in the test cases and test methods. The issues observed were
    - assertions should have a message
    - correct spelling of method names, especially JUnit keywords like setUp() and tearDown()
    - JUnit tests should contain an assert or fail
    - classes which contain JUnit test cases should end with Test
    - Use assertSame(x, y) instead of assertTrue(x==y)

  There were two main problems observed with this ruleset. This ruleset should be run only on the JUnit source files. If it is run on the main project, then each java file is considered as a JUnit test and the report contains a higher percentage of false positives. Especially the messages "Assertions should have a message" and "Incorrect method names setUp() and tearDown()" occur when method names setup() is used in a normal class file or assert statement is used within non-JUnit code.
  There is another false positive while running the code on JUnit test cases. If a test method calls another method which does the assert() or fail() and does not have assert or fail in the method body then the ruleset is not able to recognize it and generates errors. The ruleset does not verify if the test case is valid or not. It checks for syntax errors. Hence it does not add any value. Code review for test cases should help in verifying errors. This ruleset need not be used.

- **Strings (rulesets/string.xml)** – This ruleset identifies problems that occur while using String and StringBuffer. The common errors observed are
    - Avoid duplicating string literals
    - Appending characters to StringBuffer should be avoided
    - Calling String.valueOf() to append to a string is not necessary
    - String.trim().length() == 0 is inefficient to check if string is empty
    - Constructor for StringBuffer is initialized with a smaller number and more characters are appended (buffer overflow)
    - Using equalsIgnoreCase() is efficient instead of converting strings to upper or lower case and then comparing.
    - indexOf(char) is faster then indexOf(String)
    - calling the `String` constructor, and calling `toString()` on `String` objects is unnecessary

  This ruleset covers various conditions for String and StringBuffer usage. Since String objects are used heavily, it is beneficial to run this test case and correct the related errors. Almost all errors are relevant and there were no false positives observed. This helps in avoiding buffer overflows and improving performance and memory usage by proper allocation of strings and calling appropriate methods.

- **Braces (rulesets/braces.xml)** – This ruleset checks `for`, `if`, `while`, and `else` statements and the usage of braces. All the results were of the same type, which is

- Avoid using if and else without curly braces

IDEs like Eclipse create templates for if-else-elseif statements which include the curly braces using the auto complete option (this is not a default option and needs to be enabled). Hence this ruleset need not be used.

- **Code size (rulesets/codesize.xml)** – This ruleset checks for the following conditions
  - Overly long methods,
  - Methods with too may parameters
  - Classes with too many methods,
  - Cyclomatic complexity and NPath complexity

  This is a useful ruleset which helps in simplifying code and forces adoption of appropriate object oriented programming concepts to reduce complexity. This combined with code review should be helpful in generating modular code which can be tested easily and is easy to maintain.

- **Javabeans (rulesets/javabeans.xml)** – This ruleset inspects JavaBeans components. The typical errors observed are
  - Non-transient and non-static members need to be marked as transient or accessors should be provided
  - Classes which implement Serializable should have a serialVersionUID

  The first error indicates non compliance with JavaBean coding standard. This can be caught during reviews. The second error is usually shown as a warning in Eclipse IDE. This ruleset can be used if JavaBeans is used for coding. If not this does not add any value.

- **Finalizers** – This ruleset identifies two types of errors overall
  - If finalize is used it should be protected
  - Last call in finalize should be a call to super.finalize

  Since finalize is being used very rarely in current implementations it is not necessary to use this ruleset. For the few situations where it is used, it is easy to remember the two conditions and can be checked during reviews.

- **Clone (rulesets/clone.xml)** – There are only a few rules for `clone()` methods. They are
  - classes that override `clone()` must implement `Cloneable`,
  - `clone()` methods should call `super.clone()`
  - `clone()` methods should be declared to throw `CloneNotSupportedException` even if they don't actually throw it

  The main problem with clone() method is in the case of deep copy and shallow copy. This ruleset does not verify if that is achieved. Since there are only three rules, it is fair to assume that it should be included as a part of coding standard and should be an item in the review checklist. This ruleset can be run once at the end to verify if the standard is met and need not be run always.

- **Coupling (rulesets/coupling.xml)** – The errors which are generated with this ruleset are
  - Too many imports or too many different objects indicate coupling
  - Avoid usage of subclass types like Vector, ArrayList and HashMap and use the supertype or interface instead

There are false positives when this ruleset is run. In projects which use various other projects for functionality like using Tomcat, log4j, JBoss, Hibernate, Eclipse RCP, Struts 2 and BIRT which are typical for any web applications, there are usually too many imports and too many different objects. Since the report is full of such warnings it is difficult to find the useful messages. Usage of subclasses again should be a coding convention and should be included in code reviews. This ruleset does not add value.

- **Strict exceptions (rulesets/strictexception.xml)** – The errors generated with this ruleset are
  - Raw exception types should not be thrown
  - methods should not be declared to throw `java.lang.Exception`,
  - Avoid throwing null pointer exceptions
  - Catch should not throw the exception caught (this is also found in design ruleset)
  - `Throwable` should not be caught

  This ruleset is helpful because exception handling is something which needs to be done well, else the code may crash due to unforeseen errors. Since PMD generates exhaustive analysis it is easy to not miss out conditions. The analysis results observed did not contain any false positives.

- **Controversial (rulesets/controversial.xml)** – This ruleset has some conditions which cannot be followed in practice. The typically observed warnings are as follows
  - Each class should have atleast one constructor
  - A method should have only one exit point
  - Avoid unnecessary constructors
  - It is good practice to call super() in constructors
  - Captures data flow anomalies
  - Use explicit scoping rather than default package private scope
  - Don't assign null to an object

  It is not possible to always have one exit point. It is often considered that assigning an object to null as initial state is a good practice. Some of PMD's rules are valid and some are arguable hence the name controversial ruleset. This does not add value to the code analysis because it does not catch errors which might break the system or might cause the system to be insecure. Hence the ruleset need not be used.

- **Logging (rulesets/logging-java.xml, rulesets/ logging-jakarta-commons.xml)** – This ruleset checks for usage of logging. The errors identified are as follows
  - logger variables should be static and final
  - System.out.print and println, and printStackTrace should be replaced with calls to logging

  This ruleset is not very helpful. There are too many false positives which are generated. In most cases System.out.println and printStackTrace are used reliably. Moreover print messages are easily observable during unit testing or integration testing and hence can be corrected.

- **J2EE (rulesets/j2ee.xml)** – This ruleset checks for compliance with J2EE architecture. Since none of the projects used for this report followed a J2EE architecture there was only one

error which was produced throughout which was usage of getClassLoader(). This ruleset is not applicable for non-J2EE projects

- **Optimizations (rulesets/optimizations.xml)** – This ruleset covers certain optimization conditions. The typical conditions referred to here are

    - Parameters, fields or variables not assigned should be declared as final
    - ArrayList can be used instead of Vector
    - Use StringBuffer instead of += for concatenating strings

  There are many false positives in this ruleset. The first condition results in almost 90% of the report. The rest of the conditions are very rare. The rare conditions are valuable ones and hence they should be added to the customizable list for verification.

- **Type resolution (rulesets/typeresolution.xml)** – This ruleset captured only two types of error conditions
    - classes that override `clone()` must implement `Cloneable`,
    - Avoid usage of subclass types like Vector, ArrayList and HashMap and use the supertype or interface instead

  This ruleset covers conditions which are already covered in Cloning and Coupling rulesets. These conditions do not add specific value and they can be covered as a part of coding standards and review checklists.

- **Unsecure code (rulesets/sunsecure.xml)** – This ruleset checks for array assignments. In particular it generates the following conditions
    - Internal arrays being stored directly
    - Return variables which expose internal arrays

  These are conditions which need to be checked to ensure that arrays are handled correctly. Hence this ruleset needs to be executed.

Custom Ruleset

- **Favorites (rulesets/favorites.xml)** – This ruleset contains rules from the different rulesets discussed above which are used quite often. This also shows how to customize rulesets in PMD. It comes by default with PMD, as an example of how to customize rulesets.

Based on the analysis above the following rulesets should be adopted.

- Basic (rulesets/basic.xml) – This needs to be customized to remove some false positives
- Design (rulesets/design.xml)
- Strings (rulesets/string.xml)
- Code size (rulesets/codesize.xml)
- Strict exceptions (rulesets/strictexception.xml)
- Optimizations (rulesets/optimizations.xml) – This needs to be customized to remove false positives
- Unsecure code (rulesets/sunsecure.xml)

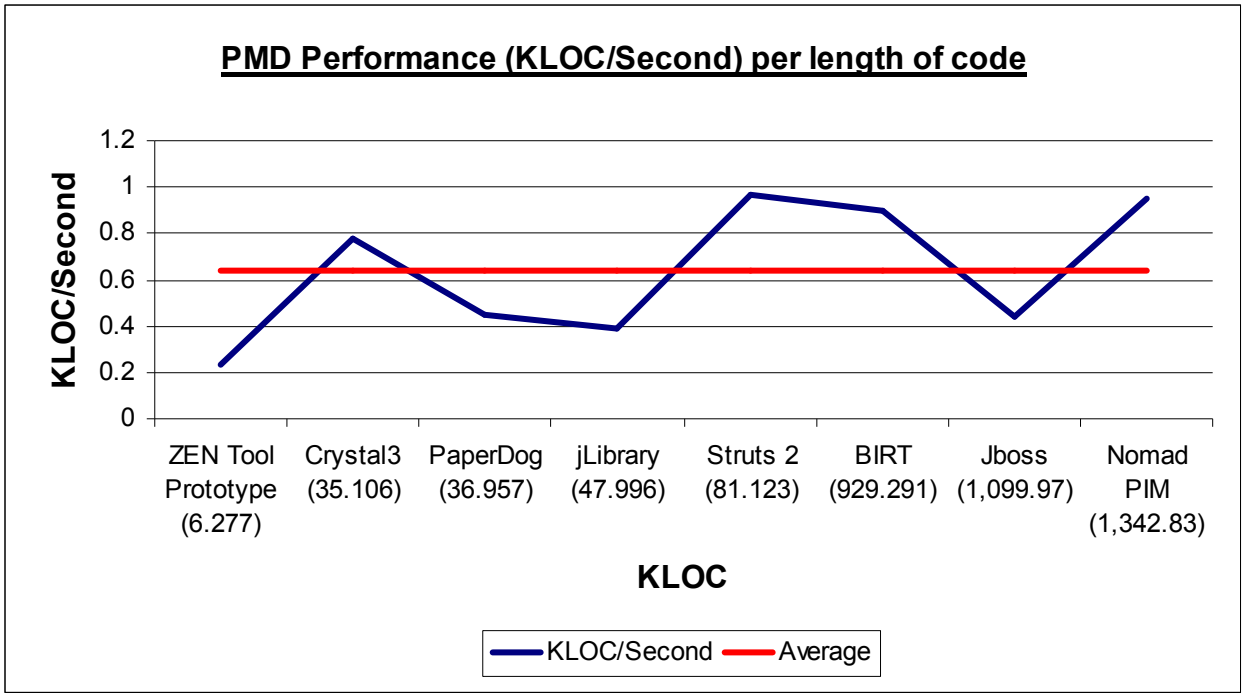
4.2 Tool Usage Analysis

In addition to collecting ruleset data, we will also conduct a more qualitative evaluation based on usability, performance, scalability, modifiability, limitations and documentation.

- **Usability**
  - Pros:
    - Command line is easy to execute
    - Lots of options for execution with ant tasks, maven plugins, ide plugins
    - Easy to run many rulesets at once
    - Many options to run on files, directories, jars, or even zip files
    - Many options for output formats very useful
    - Ability to automatically search all .java files from a directory root makes running on an entire code base very simple
  - Cons:
    - When running rulesets on a file or a set of files, there is no way to tell which ruleset generated which error without direct examination of the ruleset itself. This means you can run in 2 primary modes: 1) All rulesets on a single file or 2) One ruleset on all files. Each paradigm requires multiple runs and does not lend itself to consolidation. If you try to run all rulesets on a large codebase, you end up overwhelmed by the amount of data you get.
    - PMD *CAN* fail (there was a failure on jLibrary project which did not yield any information as to the error) without explanation of why.
    - There are other usability problems related to documentation and are listed in the discussion on documentation

- **Performance**
  - Average Run Times: (All 22 rulesets were run individually using a batch file; note that the times would be much lower if they were all run from a single command line call)
    - about 1:23 minutes to run on PaperDog
    - about 2:03 minutes to run on jLibrary
    - about 0:45 minutes to run on Crystal3
    - about 1:24 minutes to run on Struts 2
    - about 17:13 minutes to run on BIRT
    - about 23:29 minutes to run on Nomad PIM
    - about 0:27 minutes to run on ZEN Tool Prototype
    - about 42:01 minutes to run on JBoss
  - SLOC: (Using antcount to automatically count the SLOC in the source directories)
    - PaperDog: 36957 SLOC
    - jLibrary: 47996 SLOC
    - Crystal3: 35,106 SLOC
    - Struts 2: 81,123 SLOC
    - BIRT: 929,291 SLOC
    - Nomad PIM: 1,342,830 SLOC
    - ZEN Tool Prototype: 6,277 SLOC
    - JBoss: 1,099,970 SLOC

- We obtain the following performance (KLOC/Seconds) graph from our data:



  - The non-linearity of our results is due to two factors:
    - We ran PMD on different machines with different hardware settings for pairs of two tools being evaluated.
    - The SLOC value is not a perfect indication of the complexity of an application, because it includes white spaces, braces, and comments that do not get compiled at run time (so a very thoroughly documented source code may appear to be bigger than a scarcely documented but more complex source code).
  - The average performance is of 0.64 KLOC/Second, and performance seems to improve a bit as the source code base increases, without significantly deteriorating with a very large code base.
  - Further analysis would be required to obtain a better picture of the overall performance of the tool, such as running all tests on a single machine, and mapping the results.

- **Scalability**
  - We could tie in scalability with the performance graph we've obtain above, and, though the graph is not entirely accurate, we get a general sense that performance improves to a certain extent as the number of KLOC increases. So the tool seems to be quite scalable, and it runs well for very large source code bases.
  - For very large code bases, generating html (at least) requires significant room. On the other hand, for well-maintained code bases on which PMD is run every build and the problems are fixed, scalability could be a low priority issue because the report files would always be very small. Also, we ran virtually all rulesets; perhaps that would not be a necessity for a real development project.

- **Modifiability**
  - Pros: You can add/modify your own rulesets, run mixtures of rulesets for each run, use optional arguments to run PMD on codebases using new (and old, for regression testing) JDKs, etc.
  - Cons: Adding code to create a new rule requires an understanding of ASTs, since that's what it's based on.

- **Documentation**
  - Pros:
    - Good online documentation of installation procedures
    - Good online documentation of basic commandline running and other methods of running
    - Good online documentation (for the knowledgeable) of how to write your own rulesets
    - Good referencing to the meaning of each error in each report (for html, at least)
  - Cons:
    - Does not tell you that running on a directory will run on all subdirectories as well (this cost me 2 hours!)
    - Does not give a list and examples of the various types of output you can get (and how to record it)
    - Does not give an explicit list of what the ruleset names actually are (not the general names, but the actual commandline parameters)-you have to go into the .jar files to figure that out.
    - Does not give examples of how to best run PMD on large codebases such that you aren't overwhelmed by a single report.

- **Automation**
  PMD allows the analysis mechanism to be integrated as an ant task. Since we are going to use ant for the build process in the studio project it is easy to automate the generation of the results. However parsing the information in the reports and identifying areas of concerns will be time consuming atleast in the initial stages of usage. After a few iterations we can use a diff tool to see the new messages generated and concentrate on them alone.

- **Limitations**
  - Only works for code. Cannot find runtime issues. Quite a few rulesets find things that inspections or even eclipse will find (like obsolete methods). These rulesets therefore can be used more as a sanity check.
  - Does not validate synchronization mechanisms and threading concepts.

5. Conclusion

Based on the information above, it is obvious that PMD is a useful tool to apply to projects to analyze code and catch errors which otherwise can be missed with code reviews and inspections. Since it is flexible allowing customizations, it is advantageous to apply specific rules to projects. There is no learning curve required to know how to use the tool and it lends itself to automation so that it can be incorporated in any Java project. All these make PMD a valuable tool to use. However PMD cannot be used alone and needs to be combined with effective coding and testing practices like using coding standards, inspections and code review processes.

6. Appendix A: View of PMD as an Eclipse plugin