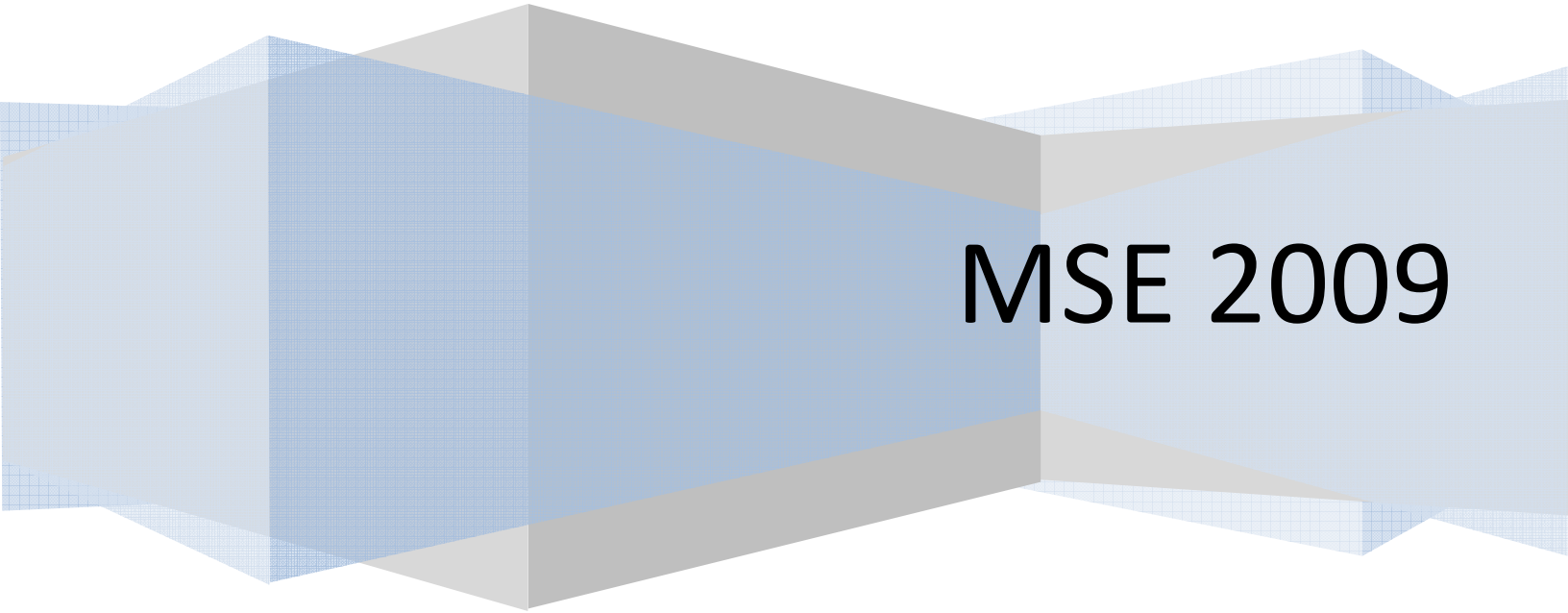
Carnegie Mellon University

Tool analysis report - PMD

Analysis of Software Artifacts

Group 2 - Abhishek Minde, Bhanu Sistla, Jeff Salk, Nan Li, Yuki Saito



MSE 2009

Table of Contents

1. Introduction

PMD is an open source, static code analysis tool. It can facilitate code reviewing through multiple sets of rules. It scans Java source code and looks for potential problems including:

- Violation of coding standards
- Unreachable code - unused local variables, parameters and private methods
- Violation of optimization standards
- Overcomplicated (complex) methods and classes
- Duplicate code - copied/pasted code means copied/pasted bugs
- Dataflow anomalies

It currently consists of 225 rules and new rules can be added with the help of a feature request. In order to keep the number of false positives reported to a minimum number, a suite of unit tests is associated to every rule that ensures that false positives are kept as low in number as possible.

There are many ways to use PMD, such as command line, ant task, maven plug-in, IDE plug-ins (Eclipse plug-in). PMD plug-in allows developers to automatically run the PMD code analysis tool on their project's source code and generate a report with its results. It also supports the separate Copy/Paste Detector tool (or CPD) that is distributed with PMD. The plug-in accepts configuration parameters that can be used to customize the execution of the PMD analysis tool.

2. PMD Workflow

The fundamental concept on which PMD is built upon is ***Abstract Syntax Tree*** (AST) which is a finite, labeled tree with nodes that represent the operators and the edges form the operands of the operators. An AST of every checked source file that is created and each rule from enabled rules sets is run against the created tree. PMD then collects all the violations and presents it in a report. Following are the steps that get executed when PMD is invoked from Eclipse:

- A file name, a directory (or project) is passed to the core PMD by Eclipse PMD plugin. The RuleSets defined in the PMD preference page are then used by the engine to check for the file(s) for violations.

- Following steps are executed in case of a directory or project containing multiple source files.
  1. Java language parser is obtained by PMD using JavaCC
  2. A source file is parsed by the parser
  3. A reference of an Abstract Syntax Tree(AST) is returned by the parser back to the PMD plugin
  4. AST is traversed by each rule in every RuleSet and violations are checked.
  5. Based on the list of Rule Violations, a report is generated. The Rule Violations are displayed in the PMD violations view or they get logged in the form of an HTML report, or TXT , CSV or XML.

- New rules can be added to PMD in two formats:
  1. Java based rules that involves writing a Java implementation class as a rule
  2. XPATH rules that involves defining a rule in an XML file.

  Both the ways can be used to add new rules and they are equally applicable. Therefore choice of one of the two formats depends upon personal preference and complexity of the rule.

3. Experimental Setup

Before starting on with the application of the tool, we decided the strategy. We decided and executed the following procedure, so that we could do things in a controlled and time-efficient manner.

Procedure:
1) Define the goals for analysis of the project under inspection/review
2) Identify rule sets from the available rules-sets, which support these goals. And for goals those are not supported, we propose to define new rules.
3) Configure the rules sets according to your goal expectations can be done in two ways. Certain rules can be turned and certain can be disabled.
4) Set appropriate priorities for the rules (e.g. informational, error, warning etc.)
5) Run the tool and evaluate the report

In this project, we focus on PMD integration with Eclipse to analysis and report the code checking results.

We picked two programs against which we decided to apply PMD:

1) **Hnefatfl game plug-in** our group developed in assignment 9, whose code is relatively simple;

2) **Process Dashboard**, which is an open source product and whose code is relatively complex.

The object of our experimentation is to capture different aspects of PMD. These aspects include its performance, usability, completeness and soundness. Completeness and soundness are for violations and defects PMD reports.

4. Data Gathered

Based on the code we decided to analyze, we chose the features and then ran PMD on them to get the violations. Following is the list of features we chose.

- Hnefatfl game plugin (boredgames.game.hnafatafl.model Package only)

Feature	Turned On/Off	Rationale
Checking of code size rules. These checks for Cyclomatic and N-Path complexities in the code. This rule set also determines size of classes and methods.	On	This analysis gives a fair idea about the complexity and size of the code. This helps the code to meet the size and complexity guidelines. More the complex code, more vulnerable it's for defects.
Basic rules set tries to address some coding standards as well as tries to	On	We wanted to calculate the complexity of the code. We thought this to be one of the most important features, as it would let the developer know when to refactor the code or redesign. These rules set can be configured and used at an organizational level, which would standardize the code and enhance the understandability of the code.
Rules for J2EE, Sql resultset	Off	The subject code does not use J2EE or sql package. Hence, enabling these rules was unnecessary and removing them would reduce the rules set which in turn would enhance the performance and eliminate the false positives generated by the PMD analysis.

Table 4.1: Rules configuration for testing the game plugin

Table 4.1 shows different rules-sets that we decided to turn off /on during the analysis.

- Process dashboard (net.sourceforge.processdash package only):
  We kept the settings of rules as defined in the table 4.1. We didn't check all subpackages of *net.sourceforge.processdash* package, and we ended up in analyzing approximately 5K lines in process dashboard code. 532 violations were reported with all default rules enabled (i.e., no rules configured or turned on or off from what is set by default)

5. PMD Result Analysis

5.1 Hnefatfl Game Plug-in

We used PMD to check the code of game plug-in, which we developed in assignment 9 and has total **838 LOC**. After running PMD upon the code with the given rule sets, we got total **225** violations in 1 second, and #Violation/LOC equals **268.5**. The analysis of these violations is shown below.

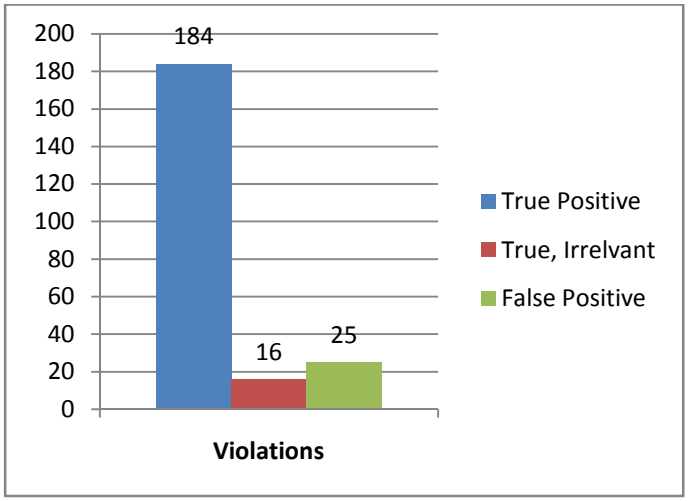


Figure 4.1 Categorized Violations

We evaluated all 225 violations manually. We reviewed every violation that was notified by the tool. Within the total 225 violations, there are 184 true positive violations, 16 true positive but not relevant violations, and 25 false positive violations. For example:

True positive:

- **NPathComplexity:** The method … has an NPath complexity of…

  This violation was observed multiple times. This is the one at line 94 in HnefataflRules class. The violation says that "The method getLegalMovesFor() has an NPath complexity of 1229313". Typically, a threshold of 200 is considered the point where measures should be taken to reduce complexity. From the information provided by PMD, we can see that the NPath complexity of method getLegalMovesFor() is too high which leads us to find a way to reduce the complexity of the code to make it easy to understand and maintain, and avoid being defect-prone.

- **CyclomaticComplexity**: The … has a Cyclomatic Complexity of ...

This violation happens multiple times. This is the one at line 94 in HnefataflBoard class. The violation says that "The class HnefataflBoard has a Cyclomatic Complexity of 41." Generally, 1-4 is low complexity, 5-7 indicates moderate complexity, 8-10 is high complexity, and 11+ is very high complexity. From the information provided by PMD, we get to know that the Cyclomatic Complexity of the class is too high which means we need to reduce the complexity of our code to make it readable, maintainable, not being error-prone, and easy to track when error happens.

- **Cut and Pasted code:**

  Cut and Pasted code detector (CPD) found several instances of cut and pasted code in the process dashboard code base. This indicated that maintainability of such code becomes difficult. And the chances that reflecting a change to one instance of such code to all other instances is not always high. Hence, keeping cut and pasted code lower is a good measure and CPD helps to achieve it.

- **SwitchStmtsShouldHaveDefault**: Switch statements should have a default label.

  This violation happens multiple times. This is the one at line 82 in HnefataflPiece class. Our code is this:

```
switch(getType()){
      case BLACK_WARRIOR: string = "BLACK_WARRIOR";
                                   break;
      case WHITE_WARRIOR: string = "WHITE_WARRIOR";
                                   break;
      case WHITE_KING:  string = "WHITE_KING";
}
```

  The game can run properly with this segment of code. However, we lose the automatic documentation provided by case-statement labels, and we also lose the ability to detect errors with the default clause. Therefore, this is a true positive violation in terms of our code.

- **IfElseStmtsMustUseBraces**: Avoid using if…else statements without using curly braces.

  This violation happens at many places. There is one at line 87 in HnefataflRules class. Our code is this:

```
if(surroundedCount == 4)return opponentPlayer;
      else return null;
```

Even thought the segment of code can executed properly, but better programming practice will make the code more readable. Therefore, we regard it as a true positive violation.

True, but irrelevant

- **NullAssignment**: Assigning an Object to null is a code smell. Consider refactoring.

  This violatioin happens once at line 271 in HnefataflBoard. Our code is this:

```
if(source!=null){
      board[location.getX()-1][location.getY()-1] = null;
}
```

  We use this segment of code to remove a piece from the board, rather than just assign an Object to null. Therefore, this violation is a true positive violation, but not relevant to our code.

- **SystemPrintln**: System.(out|err).print is used, consider using a logger.

  This violation happens once at line 487 in HnefataflBoard class. Our code is this:

```
public synchronized void addObserver(Observer o) {
      System.out.println("addobserver");
      if(HnefataflBoard.observer == null){
            HnefataflBoard.observer = (BoardPanel)o;
            super.addObserver(o);
      }
      else
           super.addObserver(HnefataflBoard.observer;
}
```

We managed the rules at the rules set level and we did not configure the rules at individual rule settings. Hence, we still had certain rule violations that we didn't want to see. This is an important thing that we will keep in mind when select the rules next time.

False Positive

We found the following false positives. Some of the false positives are because of the fact that we did not filter all rules at individual filter level, although we did it at rules set level.

DD Anomaly:

```
public String toString() {
    String string = null;

    switch(getType()){
        case BLACK_WARRIOR: string = "BLACK_WARRIOR";
                            break;
        case WHITE_WARRIOR: string = "WHITE_WARRIOR";
                            break;
        case WHITE_KING:    string = "WHITE_KING";
    }

    return string;
}
```

Fig 4.2 : toString Code: False positive DD

As shown in the code in fig 4.2, string is declared and initialized at the beginning. However, "string" is initialized to null. Then inside, the switch case string is assigned to some value. Dataflow anomaly analyzer reports this as a DD problem. It reports that we are redefining the variable "string" once it is defined. It should take care of "null" and based on that it should note that it's perfectly value to have a redefinition.

Avoid instantiating new objects inside loops:

Under optimization rules set, we have a rule that reports a violation if we instantiate any object inside a loop. This we felt was a false positive, as we thought not having any object instantiated in a look is not practical. And we thought that declaration of an object should not be done in the loop; however, instantiation could be done in a loop.

```
Point oppenent;
HnefataflBoardLocation loc;

for(int i=0;i<4 ;i++){

    oppenent = new Point(location.getX(),location.getY());
    oppenent.translate(dx, dy);
```

Fig 4.3: Instantiation in a loop false positive.

As shown in the fig 4.3, Point is being instantiated inside a loop, which is why PMD is reporting a warning.

AvoidDuplicateLiterals:

The String literal … appears … times in this file; the first occurrence is on line …

Our code of Plural likes this:

@Perm(requires="pure(this!fr) in pieceHasPlr", ensures="pure(this!fr)")

The warnings we got regarding this kind of code like this:

The String literal full(this!fr) in boardUninitialized appears 4 times in this file; the first occurrence is on line 36. It seems that PMD can't recognize the syntax of Plural.

More analysis is below:

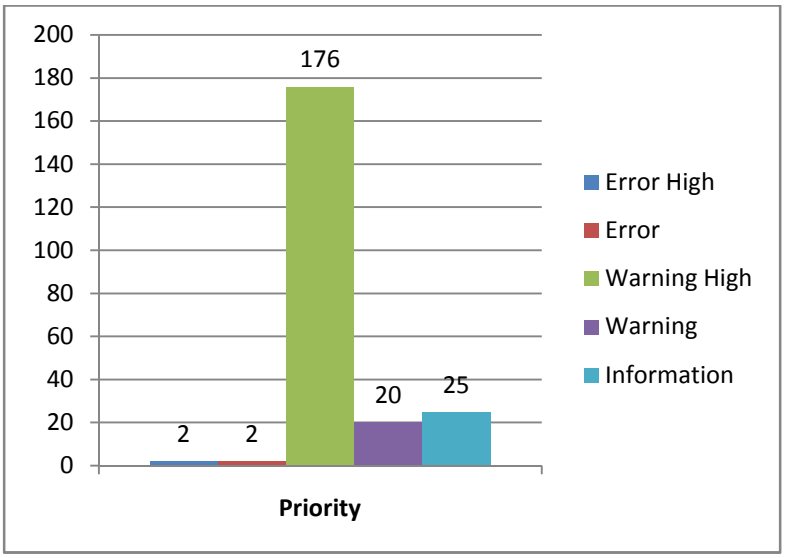


Figure 4.4 Prioritized Violations

Within these violations, most of them (95.7%) are the violations with Warning High priority (Priority 3).

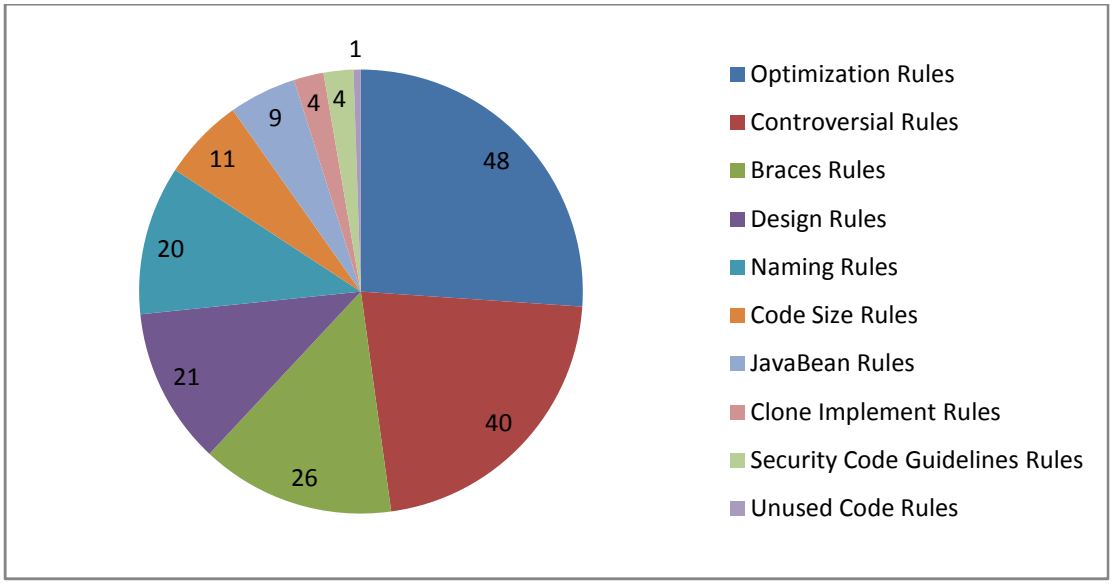


Figure 4.5 Violation Allocation

After excluding the irrelevant and the false positive ones, the true positive violations belong to 10 out of total 22 rule sets, as shown in Figure 4.5. From this chart, we can get the idea about the quality of our code. For example, the biggest portion of all the violations (26.1%) belongs to optimization rules which mean our code may need to be optimized. Meanwhile, 11.4% of our code violates the design rules, which can lead us to look back to our design and figure out if there's any design flaws.

5.2 Process Dashboard

We used PMD to check the code of net.sourceforge.processdash package. After running PMD upon the code with the given rule sets, we got total 532 violations in 6 seconds. However, due to limited amount of time, we could not visit each and every reported violation.

PDataFile.java: "Abstract classes should be named 'AbstractXXX'." This is not a common standard for Java and is definitely not part of Sun's Java coding conventions standard.

6. Lessons Learned

We identified several benefits and drawbacks of using PMD.

6.1 Benefits of PMD

1. **Usability**:

    a. It provides great help documentation with examples and detailed explanation about the warning or error.

    b. It provides flexibility to the developer to assign priorities and severity level to rules, and to define the rules based on their own purpose.

    c. It provides priority filter which can show errors and warnings with certain priority. Developers can look through all the errors and warnings easily for particular level of priority.

    d. It provides resource filter which can be used to view the checking result for a particular project if multiple projects are checked at the same time.

    e. Some useful menus are provided to facilitator the job of the developer, such as the developer can view the checking results with different views, such as show violation to packages, show violation to files, and show package with files.

    f. It provides a Violation Outline window to give a detailed checking result for the chosen Java file. For example, the Violations Overview displays there are four violations in respect of the rule ShortVariable in RulesFactory class. Once you double-click the rule, RulesFactory class will be opened and the Violation Outline window will show up simultaneously with all the violations found along with line numbers and these four violations highlighted. Once you click one of it, the corresponding line in RulesFactory class that violates the rule will be located.

    g. It provides the function to mark a violation as reviewed with reviewer's name and review time and date, which facilitates the review and tracking of the violations.

    h. It provides a flexibility to specify "max" number of reported violation per file. This helps in improving the performance for large files; however, it might give a file impression.

2. It helps code in good programming practice. For example, it can be used to check if there are variables with too short names like x, y, p, or if there's if...else statement without curly braces.

3. It helps find not only syntax problems, but semantic problems in the code. For example, it can verify both the NPath complexity and Cyclomatic complexity of the code, which is hard to find before the code executed.

4. **Rules creation**: It allows the user/developer to create their own rules with the help of inbuilt rule designer. This helps the user to define the rules that based on XPath, and other complex rules need to be defined in a separate class.

5. **Documentation**: For almost every reported violation, PMD provides an example, showing how to solve that violation. It also gives a hyperlink, which navigates to the PMD webpage that describes the violation and gives an example about how to fix it.

6. **Cut and Paste detector**: This is a very useful utility. This allows us to find all instances of cut and pasted code in different source files. Duplicate code is hard to find, especially in large projects. PMD's copy paste detector finds it pretty quickly. It uses Karp Rabin's algorithm for string comparisons. The more cut and pasted code you have, lesser the maintainability of the program.

6.2 Drawbacks of PMD

1. It doesn't offer any "Quick Fix"es, as it common with Java problems found in Eclipse, even though right-clicking on an error in the Violations Outline view has a "Quick Fix..." entry, it is always disabled (grayed-out). This is even for errors where the suggestion is clear. For example, "Consider replacing this Vector with the newer java.util.List".

2. **Mostly checks "Mechanical" bugs:** Most of the times, PMD will identify mechanical defects. It doesn't perform detailed analysis so that it could also give us insight into some of the semantic defects.

3. **Dataflow analyses:** Dataflow analyzer has been recently built in the PMD. And currently it can report on primitive dataflow anomalies in the program. They are expecting other analyses to be built on top of this dataflow analyses. However, it is still in evolving state.

4. Selecting the project in Eclipse (Project Explorer view) and choosing "Generate Reports" doesn't result in anything visible. There was progress showing in the Eclipse progress message area and it seemed to complete to 100%, but then nothing happened. I even had all possible PMD views visible and checked each one of them, but nothing changed in the views.

5. It doesn't seem possible to run PMD on just a single package without it also running the checker on the sub(child) packages. For example, in trying to run it only on code in the "net.sourceforge.processdash" package (selecting that node in the Package Explorer view), it ran it on all code in all subpackages, too. Unfortunately, the majority of the code-base of the Process Dashboard project is a sub-package of net.sourceforge.processdash. And checking the options for PMD there appears to be no option to filter this. However a viable workaround is to simply select all the code in the package of interest and then run PMD on the selected code. In addition, note that the Eclipse Problems view seems to suffer the same limitation - even trying the various options to "show selected element only " and "show selected element and its children" I could not get it to only show the problems for the net.sourceforge.processdash package (the code in that package) itself without also showing that of the subpackages.

6. It requires **understanding of the AST** in order to understand and define the XPath definitions (rules).

7. It doesn’t provide help for Rule Designer, and is not clear what the input is for each part of the designer and how it works with Rules Configuration.

8. Some results are confusing. For example, there two violations appear at line 5 of HnefataflPiece class. Both the violations are identical and there’s no reason why it produces the same result twice.

9. It shows that the file has errors in package explorer if there is any priority 1 (error high) or priority 2 (Error) violation appear. The icon shown for the violations with either priority is a file with a red cross, which is same as the icon for compiling error. It confuses the developers as if these are compile errors in the code.

10. The priority of given rules needs to be refined due to its inaccurate. For example, the priority of SystemPrintln violation (System.(out|err).print is used, consider using a logger.) is defined as Error, but it can be just a warning or a information. Moreover, currently even for a low cyclomatic complexity, it is showing a warning (exact severity is "warning high"). The severity level should at least be lowered for such a case, or the rule should be modified so that this message is not shown unless the complexity is above a given threshold value.

7. Conclusion

PMD can be applied for Java code and would be very beneficial for use in industry. The main drawback is that the developer has to take the time to customize the rules to eliminate the ones that aren't considered relevant and/or reducing the severity of certain rules. But that's typically a one-time setup, followed by possibly some small incremental fine-tuning as it gets used more and more.

After having thought about all its benefits and drawbacks we've decided to use PMD for our studio project in summer.

We will define the QA strategy, which will primarily have what kind of coding, optimization standards we as a team are going to adhere to. Tools like PMD can streamline our process of development of code that adheres to common or predefined standards (at least to certain extent).

We realized that for any analysis tool, it is very important that it can be extended to support more analyses. E.g. PMD supports creation of new rules with quite an ease. This allows the developers to add their rules as and when they need it. This enhances the applicability of the tool.

References

[1] PMD, software policing software, - http://pmd.sourceforge.net/

[2] XML Path Language (XPath), when less is more, http://www.w3.org/TR/xpath

[3] Eclipse, http://www.eclipse.org