ICU-CMU MSE 2008-2009

17-654: Analysis of Software Artifacts

Mini-Project: Tool or Analysis Practicum

April 8, 2009

the CruX

Bokuk Seo
Do-Hoon Kim
Yong Gyu Kim
Sang-hyun Lee

Revision History

Version	Date Updated	Revision Author	Brief Description of Changes
0.1	04-01-2009	Yong Gyu Kim	Initial document created
0.2	04-07-2009	Yong Gyu Kim	Updated for tool experimentation result
0.3	04-08-2009	Yong Gyu Kim	Updated for tool experience detail (FP)
1.0	04-08-2009	Yong Gyu Kim	Updated for reference sites

Instructions: For the suggested topics below, replace <text within brackets> with project specific information. Some of the topics may not apply to all projects. In the topics where a response is optional, a choice, ["N/A for this report"] is included in the instructions.

Table of Contents

< Generated Automatically. To update table based on changes in the document, select the table and hit F9. >

1. Introduction

This document is to describe the Java static analysis tool, PMD which is famous for detecting bugs before running.

1.1 Report general

PMD is the Open Source (BSD licensed) tool of Tom Copeland and has functionality to find bugs as analyzing Java source code. It is similar with FindBugs and Lint4j for general perspectives. But here we tried to catch essential functions and usefulness of PMD as focusing on various viewpoints.

Indeed, PMD checks for a long list of possibly bad programming practices and possible error conditions in source programs. Our feeling about it is mixed -- it is undoubtedly very useful, but the number of warnings about things which actually OK were excessive. Of course, what we should be doing it apply the rules so it diagnoses what we think are important.

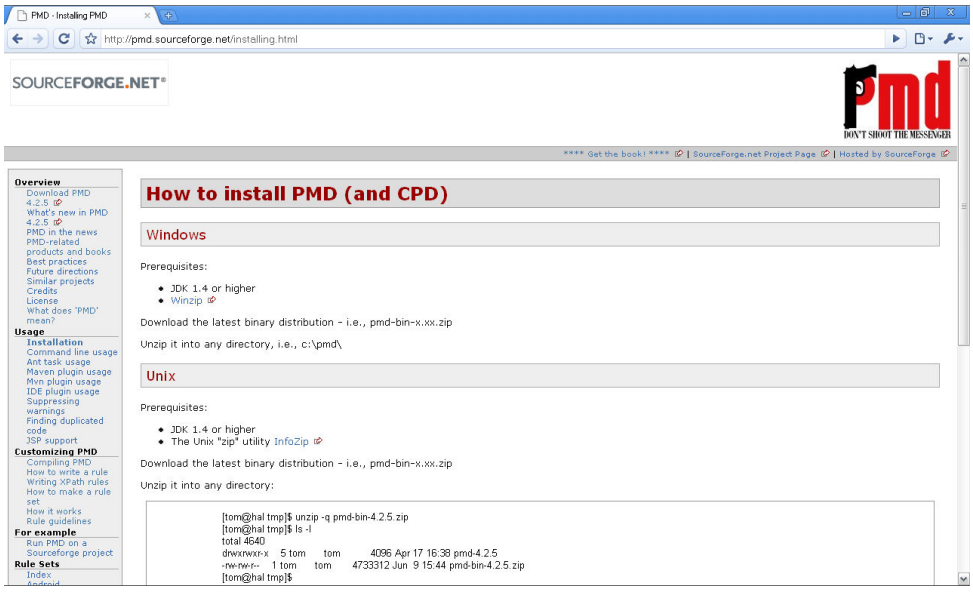
1.2 Running PMD

1.2.1 Installation

PMD is made in Java and it can be used the way of standalone or plug in.

- Standalone: It can be downloaded from SourceForge.net

- Plug-in Eclipse : http://pmd.sourceforge.net/eclipse



[FIGURE1. PDM PROJECT SITE]

1.2.2 How it works

At the heart of PMD is the JavaCC parser generator, which PMD uses in conjunction with an Extended Backus-Naur Formal (EBNF) grammar and JJTree to parse Java source code into an Abstract Syntax Tree (AST). That was a big sentence with a lot of acronyms, so let's break it down into smaller pieces.

Java source code is, at the end of the day, just plain old text. As your compiler will tell you, however, that plain text has to be structured in a certain way in order to be valid Java code. That structure can be expressed in a syntactic metalanguage called EBNF and is usually referred to as a "grammar." JavaCC reads the grammar and generates a parser that can be used to parse programs written in the Java programming language.

[TABLE1. SAMPLE CODE]

- Source Code	`public class Foo {`
	`  public void bar() {`
	`  System.out.println("hello world");`
	`  }`
	`}`

- Abstract Syntax Tree	`CompilationUnit`
	`TypeDeclaration`
	`ClassDeclaration`
	`UnmodifiedClassDeclaration`
	`ClassBody`
	`ClassBodyDeclaration`
	`MethodDeclaration`
	`ResultType`
	`MethodDeclarator`
	`FormalParameters`
	`Block`
	`BlockStatement`
	`Statement`
	`StatementExpression`
	`PrimaryExpression`
	`PrimaryPrefix`
	`Name`
	`PrimarySuffix`
	`Arguments`
	`ArgumentList`
	`Expression`
	`PrimaryExpression`
	`PrimaryPrefix`
	`Literal`

2. Tool Experience

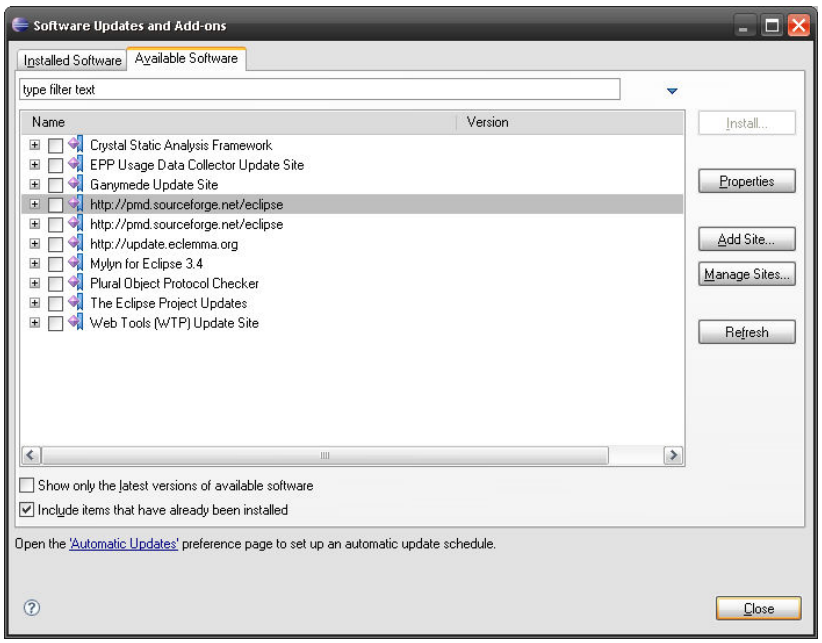
2.1 Basic Use of PMD

2.1.1 Current Rulesets

PMD uses the Rulesets to check and validate the code and they are called as Basic Rules for pre-defined. And list of rulesets and rules contained in each ruleset. [1]

- Android Rules: These rules deal with the Android SDK, mostly related to best practices. To get better results, make sure that the auxclasspath is defined for type resolution to work.
- Basic JSF rules: Rules concerning basic JSF guidelines.
- Basic JSP rules: Rules concerning basic JSP guidelines.
- Basic Rules: The Basic Ruleset contains a collection of good practices which everyone should follow.
- Braces Rules: The Braces Ruleset contains a collection of braces rules.
- Clone Implementation Rules: The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size Rules: The Code Size Ruleset contains a collection of rules that find code size related problems.
- Controversial Rules: The Controversial Ruleset contains rules that, for whatever reason, are considered controversial. They are separated out here to allow people to include as they see fit via custom rulesets. This ruleset was initially created in response to discussions over UnnecessaryConstructorRule which Tom likes but most people really dislike :-)
- Coupling Rules: These are rules which find instances of high or inappropriate coupling between objects and packages.
- Design Rules: The Design Ruleset contains a collection of rules that find questionable designs.
- Finalizer Rules: These rules deal with different problems that can occur with finalizers.
- Import Statement Rules: These rules deal with different problems that can occur with a class' import statements.
- J2EE Rules: These are rules for J2EE
- JavaBean Rules: The JavaBeans Ruleset catches instances of bean rules not being followed.
- JUnit Rules: These rules deal with different problems that can occur with JUnit tests.
- Jakarta Commons Logging Rules: The Jakarta Commons Logging ruleset contains a collection of rules that find questionable usages of that framework.
- Java Logging Rules: The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
- Migration Rules: Contains rules about migrating from one JDK version to another. Don't use these rules directly, rather, use a wrapper ruleset such as migrating_to_13.xml.
- Migration13: Contains rules for migrating to JDK 1.3
- Migration14: Contains rules for migrating to JDK 1.4
- Migration15: Contains rules for migrating to JDK 1.5
- MigratingToJava4: Contains rules for migrating to JDK 1.5

- Naming Rules: The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth.
- Optimization Rules: These rules deal with different optimizations that generally apply to performance best practices.
- Strict Exception Rules: These rules provide some strict guidelines about throwing and catching exceptions.
- String and StringBuffer Rules: These rules deal with different problems that can occur with manipulation of the class String or StringBuffer.
- Security Code Guidelines: These rules check the security guidelines from Sun, published at http://java.sun.com/security/seccodeguide.html#gcg
- Type Resolution Rules: These are rules which resolve java Class files for comparisson, as opposed to a String
- Unused Code Rules: The Unused Code Ruleset contains a collection of rules that find unused code.



[FIGURE2. PDM PLUG-IN WITH ECLIPSE]

2.1.2 Case of basic use

[TABLE2. PMD XML REPORT]

```
<?xml version="1.0"?>
<pmd>
<file name="/Users/elharo/src/ImageGrabber.java">
<violation line="32" rule="ShortVariable" ruleset="Naming Rules" priority="3">
Avoid variables with short names like j
</violation>
<violation line="105" rule="VariableNamingConventionsRule" ruleset="Naming Rules"
priority="1">
Variables that are not final should not contain underscores
(except for underscores in standard prefix/suffix).
```

```
</violation>
</file>
<error filename="/Users/elharo/src/ImageGrabber.java"
  msg="Error while processing /Users/elharo/ImageGrabber.java"/>
</pmd>
```

In table 2, PMD shows the 2 problems in the code list. Line 32 and 105 with underscore violated ImageGrabber.java.

2.2 Custom Use of PMD

2.2.1 Rule Coding

We used JavaCC and JJTree to use an EBNF grammar to turn our source code into an structured object hierarchy. Now we can put those objects to use by writing some rules to look for problems.

Generally, a PMD rule is a visitor that traverses the AST looking for a particular pattern of objects that indicates a problem. This can be as simple as checking for occurrences of new Thread, or as complex as determining whether or not a class is correctly overriding both equals and hashcode.

2.2.2 How to write a PMD rule

Writing PMD rules is cool because you don't have to wait for us to get around to implementing feature requests. Here's a simple PMD rule that checks for empty ***if*** statements

[TABLE3. WRITE PMD RULE: IF]

```
// Extend AbstractRule to enable the Visitor pattern
public class EmptyIfStmtRule extends AbstractRule implements Rule {
   // This method gets called when there's a Block in the source code
   public Object visit(ASTBlock node, Object data){
      // If the parent node is an If statement and there isn't anything
      // inside the block
      if ((node.jjtGetParent().jjtGetParent() instanceof ASTIfStatement)
         && node.jjtGetNumChildren()==0) {
         // then there's a problem, so add a RuleViolation to the Report
         RuleContext ctx = (RuleContext)data;
         ctx.getReport().addRuleViolation(createRuleViolation(ctx,
            node.getBeginLine()));
      }
      // Now move on to the next node in the tree
      return super.visit(node, data);
   }
}
```

3. PMD in practice

3.1 Analysis Overall

3.1.1 Experiment Setup

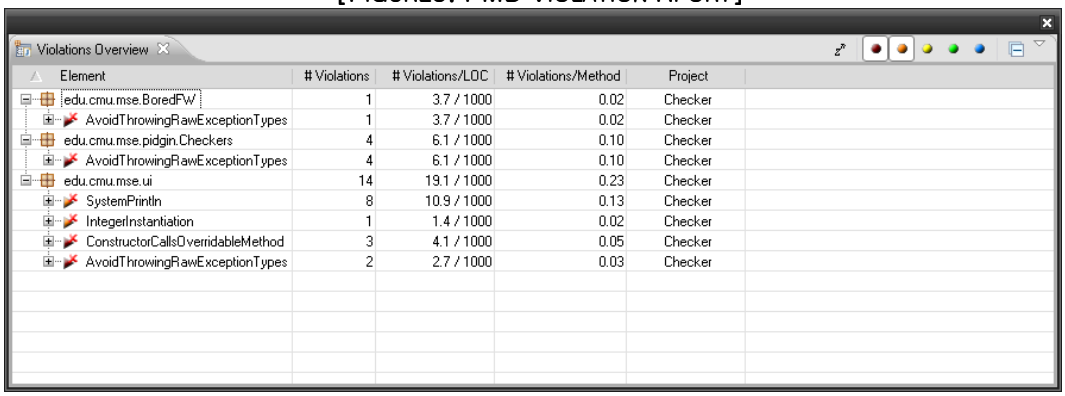
(1) Target Source

CHECKERS: ASSIGNMENT WE DID

- Major Concerns
  - Basic rule verification: Unfamiliar with Java
  - Design Concern: Not much experience for design
  - Others are usual but controversial

3.1.2 Analysis Results from PMD

[FIGURE3. PMD VIOLATION RPORT]



PMD has 5 levels of violation from 1 to 5: High Error, Error, High warning, Warning, and Information. According to 1 and 2 are considered as critical to run the program, we have to solve them firstly. Here are the actual results of violation summary of our target application.

[TABLE4. PMD RESULT]

Level	# of Violation Count	# of Violations / LOC
1	10	5.5
2	9	12.3
3	366	274.5
4	0	0
5	46	34.15

[Note] http://pmd.sourceforge.net/scoreboard.html would show the detail benchmark data using PMD for open sources.

3.2 Findings in detail

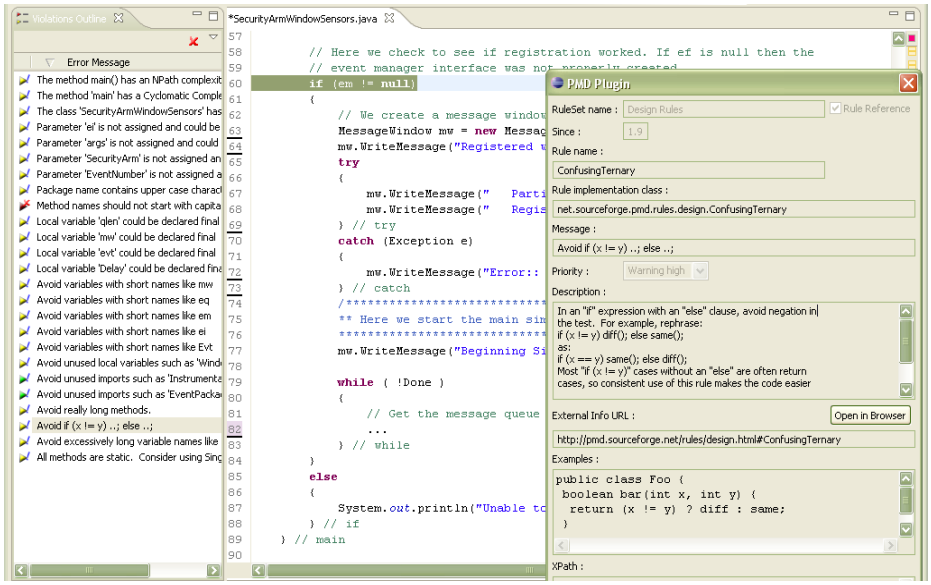
During our practice, we focused on trying to catch exact conformance with tool and our project codes. From this point of approaches, we can get the useful information out projects and additional guides to improve them. Here are the main rulesets we tried as following cases:

- Basic Rules
- Design Rules
- Controversial Rules

3.2.1 Useless (True positives relevant)

- **Basic Rules: ConfusingTernary**

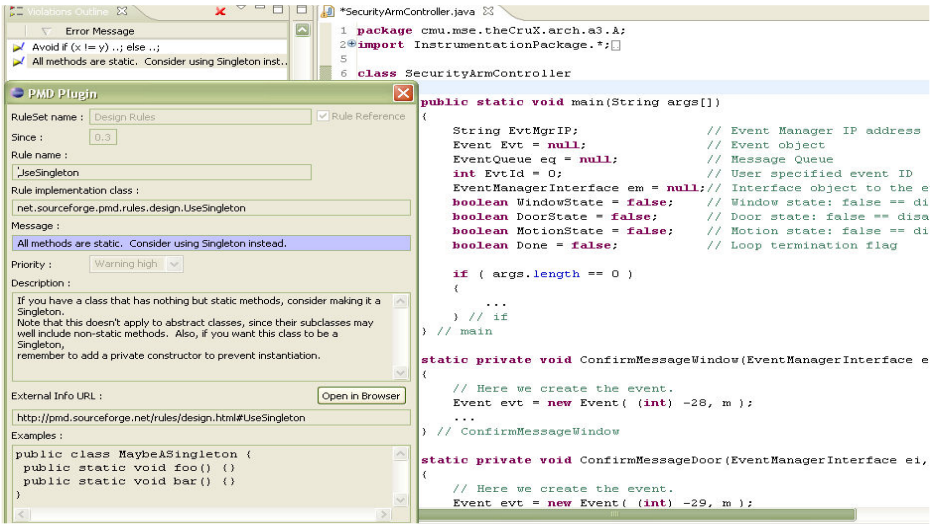
  When we tried to check the code to write the IF statement, we wrote the command prior to else statement. At this case, we must use the Object .equal to follow the rule but we did not. This ruleset clearly shows the misuse of code



[FIGURE3. BASIC RULES: CONFUSINGTERNARY]

- **Design Rules: UseSingleton**

  In case of every static method in the class, the tool recommend to use the Singleton pattern for design.



[FIGURE4. DESIGN RULES: USESINGLETON]

3.2.2 Uselessness (True positive irrelevant)

Here we put two examples we found as the true positive things which are not relevant to goal of project.

- **Design Rule: ConstructorCallsOverridableMethod**

  When we call override-able methods during construction, it poses a risk of invoking methods on an incompletely constructed object and can be difficult to discern.

[FIGURE5. DESIGN RULES: CONSTRUCTORCALLSOVERRIDABLEMETHOD]

```
60          }
61          @Full( value="gameObjectState", requires="noInst
62          public CheckerWindow() {
63              loadConfiguration();
64              game = getGameInstance();
65              chkFrm = new CheckerJFrame();
66              chkFrm.addCaller(this);
67              chkFrm.setVisible(true);
68              chkFrm.initBoard();
69              startGame(true);
70          }
```

- **Controversial Rules: NullAssignment**

  Assigning a "null" to a variable (outside of its declaration) is usually bad form.

[FIGURE6. CONTROVERSIAL RULES: NULLASSIGNMENT]

```
232             //if it is second click of making move
233             else{
234                                 System.out.println("second click")   ;
235                 if( tmpPiece ==null){
236                     if( requestMove(prePoint, new Point(xP,yP), getTurn
237                         onMoveFg =false;
238                         prePoint = null;
239                         preCapPoint=new Point(xP,yP);
240                     }
```

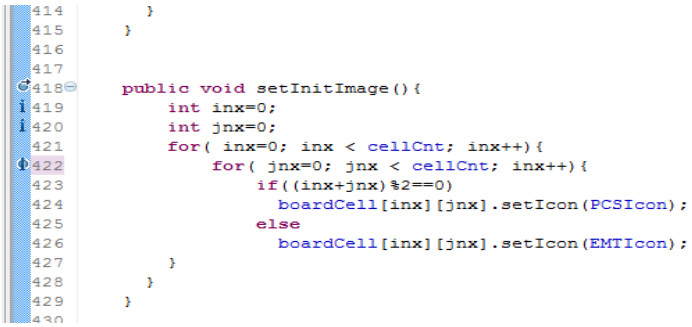
3.2.3 False positives

Here are the results taken to find out "False positives" with PMD. Actually, it might not the big problem but it could be somewhere certain conditions as below:

- **Basic Rule: JumbledIncrementer**

  It inevitably avoids jumbled loop incrementers. Although it is a usual mistake, it's still confusing even if it's what has intended.

[FIGURE7. BASIC RULE: JUMBLEDINCREMENTER]

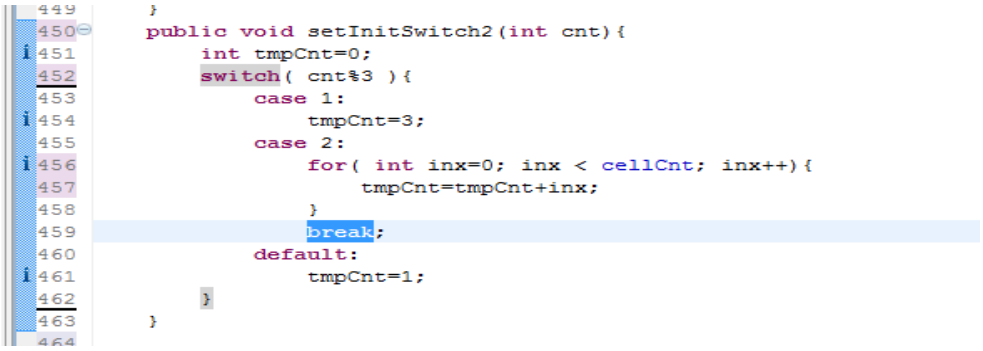


```
414       }
415     }
416
417
418     public void setInitImage(){
419         int inx=0;
420         int jnx=0;
421         for( inx=0; inx < cellCnt; inx++){
422             for( jnx=0; jnx < cellCnt; inx++){
423                 if((inx+jnx)%2==0)
424                   boardCell[inx][jnx].setIcon(PCSIcon);
425                 else
426                   boardCell[inx][jnx].setIcon(EMTIcon);
427         }
428       }
429     }
```

- **Design Rule: MissingBreakInSwitch**

  In case of switch → loop structure, it does not warn about the "break"-missing case of switch statement. If we are correct, it must show the message of MissingBreakInSwitch.

[FIGURE8. BASIC RULE: MISSINGBREAKINSWITCH]



```
449       }
450       public void setInitSwitch2(int cnt){
451           int tmpCnt=0;
452           switch( cnt%3 ){
453               case 1:
454                   tmpCnt=3;
455               case 2:
456                   for( int inx=0; inx < cellCnt; inx++){
457                       tmpCnt=tmpCnt+inx;
458                   }
459                   break;
460               default:
461                   tmpCnt=1;
462           }
463       }
```

4. Conclusion

PMD has typical static analysis tool's characteristics and we can summarize the PMD's own characteristics as below:

- **Strength**
  - Have access to the actual instructions the software will be executing
  - No need to guess or interpret behavior
  - Full access to all of the software's possible behaviors
- **Weakness**
  - Will not find issues related to operational deployment environments

And we can agree with additional but realistic meaning of PMD usage such as

- Cheap for tool but expensive for extension (painful)
  - This tool can be easily learned and operated
  -
- Very Shiny
  - When it finds REALLY hidden errors.
  - But it does not impressive to us because free editor simply covers most of possible error.
- Still needs to test
  - PMD cannot substitutes the unit test and acceptance test

5. Appendix

5.1 References

[1] SourceForge Project Site: http://pmd.sourceforge.net/
[2] OnJava.com, Custom PMD Rules:
http://www.onjava.com/pub/a/onjava/2003/04/09/pmd_rules.html