Analysis of Software Artifacts

Assignment 10: Tool Project

6th April 2009

The Mappers

António Alvim

Filipe Pontes

Paulo Casanova

Pedro Mota

Pedro Saraiva

Introduction

This report is the main delivery for assignment of the 2009 edition of Analysis of Software Artifacts course.

We will present the results of a source code analysis on the following applications:

- **Gaming Framework for the Bored Games** - we analyzed the programming effort for assignments 8 and 9 where we (the Mappers team) participated.
- **Plural Static Analysis Project** – we analyzed the checking tool we used to define protocols for the assignment 7 and 9 of this course.

This report includes the following sections:

Section 2 – We present and briefly describe main objectives of the tool that we used to perform the analysis.

Section 3 – We identify and describe the two projects we analyzed.

Section 4 – We detail how we performed each of the project's individual analysis.

Section 5 – We document the results for each, with particular focus on the true positives, false positives and irrelevant warnings. We provide two examples for each result type – additional examples are included in the appendixes section.

Section 6 – We provide some concluding remarks on the tools usage for the two projects and some general remarks.

Tool

We selected the tool based on the following criteria:

- We plan to use a tool that can be part of our quality assurance strategy for the implementation stage of our studio project.
- Since we are going to implement an information system in Java, the tool should include the analysis of Java source code.

As such, we selected the PMD tool [1]. PMD has no meaning for the tool name but we take the suggested "Programming Mistake Detector", as indicated by the authors[1].

We used the 4.2.5 version, released on February of 2009.

Independently of which tool it is run with, PMD statically checks source code against a set of predefined rules, to identify potential code problems such as empty try/catch/finally/switch statements, overcomplicated expressions and dead or suboptimal code.

The previous list is just the initial summary presented on the product homepage of the categories of more than 250 individual rules possible to analyze with the standard PMD distribution [2].

In this regard, it is possible to use PMD as:

- A command line executable;
- Part of the a projects' integrated build process (Maven [5] or Ant [6]);
- An integrated IDE Plug-in.

We performed our analysis the assignment with the Eclipse plug-in.

The following list highlights the core capabilities provided by PMD:

- **Standard compliance** – J2EE, JSP (and other Java variations) guidelines, coding standards and conventions;
- **Performance/size related issues** – code size, coupling, logging usage;
- **Security** – rules that compare code against the security guidelines published by Sun;
- **Design** – provides some design patterns suggestions and recommendations after analyzing code, like Singleton and improved ways of using interfaces;
- **Correctness** – how object comparisons should be made (`.equals` vs. `==`), or how cloning should be performed.

[1] As seen in http://pmd.sourceforge.net/meaning.html

Because not all of the previous rules types are interesting to the all developers, PMD allows configuring easily which rules we intend to include on our analysis.

It also includes a Rule editor capability, where we can write our own set of rules [3], which serve as an interesting option for developers who need this degree of customization.

The following picture displays eclipses' plug-in configuration menu, where rules can be managed (add, delete or update).

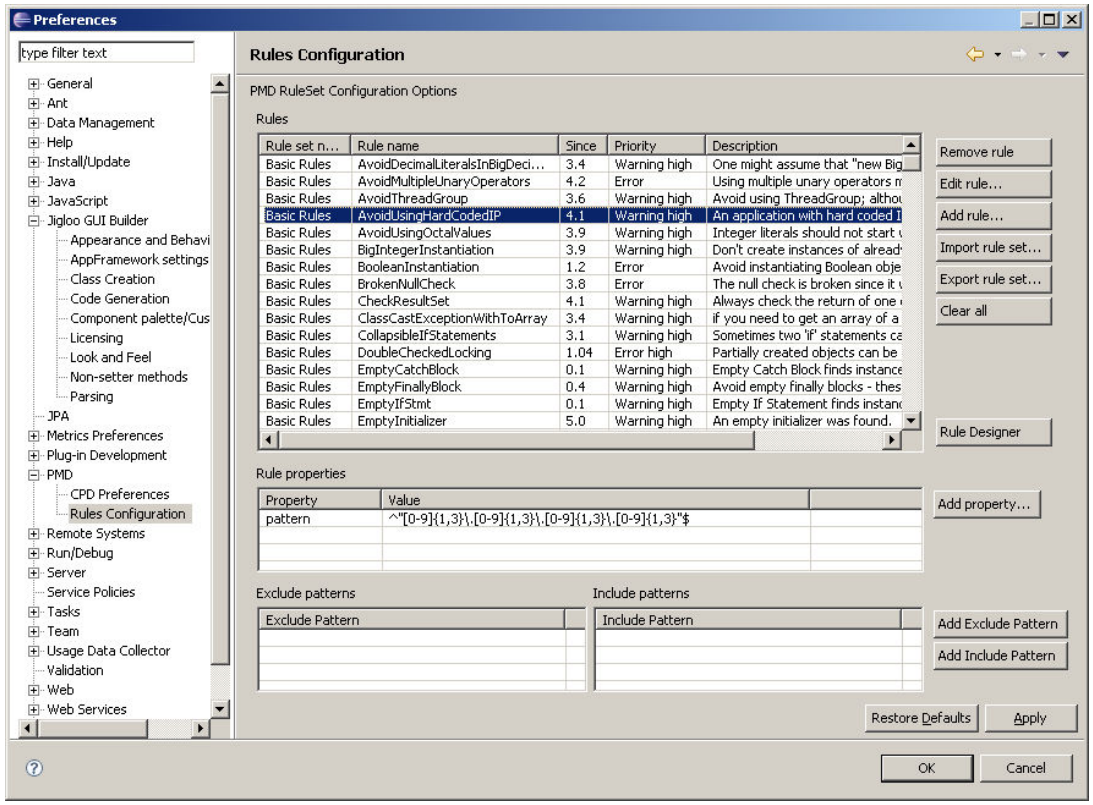

Figure 1 - Rule Configuration Screen of Eclipse's v3.4.1 PMD plug-in.

Note that the authors state the PMD's rules are not set in stone, and that there is configuration effort into achieving the desired results:

"*Generally, pick the ones you like, and ignore or suppress the warnings you don't like. It's just a tool.*"

This means that discovering the best usage for PMD usually comes with some attempts on tuning.

An important feature of PMD is the possibility of marking exceptions. There are certain times where violating a rule makes sense and it is necessary. It is possible to annotate the

source code (with a special notation embedded in a Java comment) to tell PMD to ignore a specific rule on a code section.

Projects

We tested PMD with two different projects:

Project 1 – Analysis Gaming Framework

This was the development effort of three different MSE teams for a standalone Checkers game application. This includes framework code, user interface, a checkers game implementation, artificial intelligence and JUnit tests.

The project includes 167 source code files, which represent around 3400 LOC.

Project 2 – Plural Analysis Tool

This is a project developed by Nels Beckman and Kevin Bierhoff at Carnegie Mellon University, used to perform modular typestate checking tool for Java programs. It is an Eclipse plug-in, built on top of the Crystal static analysis framework and the Antlr parser generator.

The project includes 190 source code files, which represent around 27K lines of code.

We selected these projects because they represent different challenges: the first was a joint effort of 15 elements, with different proficiency levels in the Java language. Its final objective is serving as an academic exercise during a two-week coding effort. Plural, on the other hand, is an active research tool that has been recently suffering significant updates.

Setup

We analyzed both projects source code files with the Eclipse IDE plug-in version of PMD. After the code checking against the rules, we probed the result list in order to discover significant warnings – all of the analysis was performed with Eclipse.

Gaming Framework

We configured PMD to check against all predefined rule sets. The execution time lasted around 10 seconds.

Plural

For Plural we have configured PMD not to check several controversial rules and several naming and coding conventions standards as it would generate too many spurious warnings.

The execution time lasted around 22 seconds.

Analysis

Overall, we did not manage to identify any true bugs with PMD on both plural and framework. Nevertheless, our analysis identified a significant number of best practices and improvements that could promote code maintainability and performance (this is in fact, one of PMD's major strengths).

Violations Overview

Element	# Violations	# Violations/LOC	# Violations/Method	Project
testsFramework	38	N/A	N/A	GrameFramework
tests	98	N/A	N/A	GrameFramework
test	133	N/A	N/A	GrameFramework
com.boredgames.group13.ComponentTests	101	N/A	N/A	GrameFramework
com.boredgames.group13	208	N/A	N/A	GrameFramework
com.boredgames.group10	114	N/A	N/A	GrameFramework
checkers	119	13222.2 / 1000	59.50	GrameFramework
aialg.mmab	102	N/A	N/A	GrameFramework
aialg.checkers	166	217.8 / 1000	3.32	GrameFramework
aialg	15	N/A	N/A	GrameFramework

Figure 2 - Summary violation report on gaming framework (maximum detail enabled).

The previous picture displays an unfiltered result from the PMD analysis on the gaming framework. After browsing some results and reducing the level of detail, it is possible to focus the analysis on the rules more relevant to the team. The user interface is easily customizable and integrated with eclipses' perspectives, which greatly facilitated the plug-in usage.

Violations Overview

Element	# Violations	# Violations/LOC	# Violations/Method	Project
com.boredgames.group13.ComponentTests	98	80.7 / 1000	3.92	GrameFramework
com.boredgames.group13	132	86.8 / 1000	2.64	GrameFramework
AssignmentInOperand	2	1.3 / 1000	0.04	GrameFramework
UIMain.java	1	1.3 / 1000	0.04	GrameFramework
UIMain.java	1	1.3 / 1000	0.04	GrameFramework
TooManyMethods	2	1.3 / 1000	0.04	GrameFramework
UIMain.java	1	1.3 / 1000	0.04	GrameFramework
UIMain.java	1	1.3 / 1000	0.04	GrameFramework
SystemPrintln	10	6.6 / 1000	0.20	GrameFramework
ShortVariable	10	6.6 / 1000	0.20	GrameFramework
CyclomaticComplexity	6	3.9 / 1000	0.12	GrameFramework
ConfusingTernary	2	1.3 / 1000	0.04	GrameFramework
ConstructorCallsOverridableMethod	4	2.6 / 1000	0.08	GrameFramework
LocalVariableCouldBeFinal	10	6.6 / 1000	0.20	GrameFramework
EmptyWhileStmt	2	1.3 / 1000	0.04	GrameFramework
IfElseStmtsMustUseBraces	10	6.6 / 1000	0.20	GrameFramework
IfStmtsMustUseBraces	2	1.3 / 1000	0.04	GrameFramework
UnusedLocalVariable	2	1.3 / 1000	0.04	GrameFramework
SignatureDeclareThrowsException	6	3.9 / 1000	0.12	GrameFramework
DoNotCallSystemExit	10	6.6 / 1000	0.20	GrameFramework
SimplifyBooleanExpressions	2	1.3 / 1000	0.04	GrameFramework
AvoidThrowingRawExceptionTypes	6	3.9 / 1000	0.12	GrameFramework
DefaultPackage	2	1.3 / 1000	0.04	GrameFramework
MethodArgumentCouldBeFinal	10	6.6 / 1000	0.20	GrameFramework

Figure 3 – Violation report expanded (only medium-high warnings enabled).

For example if the team was focusing on reducing methods complexity, PMD detected on Plural several methods with very high cyclomatic complexity: the winner is `edu.cmu.cs.plural.fractions.FractionalPermission.join` with cyclomatic complexity 39. On the gaming framework, the maximum value encountered was 34.

5.1 Relevant True positives

A list of true positives is the following (of course, since these relate to maintainability, the concept of "positive" is arguable):

Gaming Framework

- In Class `aialg.Checkers.CheckersMove`, line 69, PMD complains that the clone method should be implemented using the Java object clone method, `super.clone()`, instead implementing the duplication process.

  The issue is that the Java `Object` method for cloning is the only one that guarantees a correct clone. In this particular project, this isn't a current defect, because there is no subclasses of this class, but if in the future a new subclass is created, then this clone implementation wouldn't be able to clone the subclass object.

  By implementing the PMD suggestions, we would be improving the overall maintainability and robustness of the code.

```
68    @Override
69    public CheckersMove clone() {
70        CheckersMove cl = new CheckersMove();
71        for (CheckersBasicMove cbm : moves) {
72            cl.addMove(cbm);
73        }
74
75        return cl;
76    }
```

Figure 4 - Code Snippet.

Plural

- class `edu.cmu.cs.plural.linear.DisjunvtiveVisitor` (`DisjunctiveVisitor.java:45`) is declared as abstract and has three methods, non abstract, defined with no content. PMD indicates these methods should be abstract instead. In fact, when analyzing all subclasses of `DisjunctiveVisitor`, they all implemented the three methods.

  By leaving the class as it is, it is likely that someone will create a subclass and forgets to override some method. Another comment arises: shouldn't this class actually be an interface?

5.2 Irrelevant True Positives

Although a significant number of the issues identified by PMD are important, some of them can be considered irrelevant, because in reality they do not have any impact on the code.

Gaming Framework

- In different methods PMD states that should have only one exit point and that should be the last statement in the method. Although this is an (arguably) recommendable practice enhancing maintainability, the logic of the application or other type of constraints might enforce that having more than one exit point to be actually clearer than having just one.

  Examples can be found in: `com.boredgames.group13.UIMain`, lines 209, 433, 440, 494 and 501.

```
202    private boolean handleLoad() throws IOException {
203        System.out.println("Enter file name of the saved game to load: ");
204        String filename = readLine();
205
206        try {
207            currentGame = GameFactory.getGameFactories().get(0).loadGame(filename);
208            System.out.println("Load succesfully!");
209            return true;
210        } catch (NotSupportedException e) {
211            System.out.println("DEBUG - handleLoad: Game loading not supported");
212            System.exit(1);
213        } catch (IOException e) {
214            System.out.println("Problem loading from file " + filename);
215        } catch (NoGamePluginsException e) {
216            System.out.println("DEBUG - handleLoad: No plugins detected");
217            System.exit(1);
218        }
219        return false;
220    }
```

Figure 5 - code snippet

Plural

- Class `edu.cmu.cs.plural.perm.parser.ParsedParameterSummary`, line 171, PMD complains on the following code:

  ```
  Integer index = new Integer(paramIndex);
  ```

  Stating that it would be more efficient to use `Integer.valueOf(paramIndex)` instead of instantiating a new integer object. In fact, since Plural is using Java 5, just using the `paramIndex` would lead to fewer code, and a more readable one. However, this is a mere detail, which is globally irrelevant.

5.3 False positives

Gaming Framework

- Class `com.boredgames.group10.Game`, line 523. PMD marks the method `switchToInitializedState` indicating it should be abstract but it is actually documented as a dummy method specifically to be used for Plural purposes.

```
522     @Full(requires = "preInit", ensures = "initialized", fieldAccess = true)
523     protected void switchToInitializedState() {
524         // dummy method for plural
525     }
```

Figure 6 - Code Snippet.

Plural

- Class `edu.cmu.cs.plural.perm.parser.AbstractParamVisitor`, line 424. PMD marks the method indicating it should be abstract but it is actually documented as returning true.

Conclusion

PMD is a static analysis tool focusing primarily on enforcing good programming practices. It doesn't focus much on bug detection such as detecting null pointer dereferencing. Therefore, instead of promoting "functional quality" it promotes maintainability and some low-level performance optimization.

Within these framing characteristics, PMD does an excellent job being extremely fast, deeply integrated with most current development environments and build tools, and containing many predefined rules making it useful out-of-the-box.

One of the most important (unexpected) benefits of this tool is to help newcomers in the Java language by providing guidance to the best practices (PMD not only signals the issues but also explains why they are relevant).

Usage of PMD with Eclipse is straightforward: the plug-in is easy to install (follows Eclipse's standards for auto-installing and updating plug-ins) and results are easy to browse. It does take some time to identify whether the defects are in fact true positives or not.

PMD is fully customizable both in what rules to check, what is the severity associated with each rule. PMD can be extended with new rules and, being fully open source, existing rules may be modified, if desired. PMD also allows specific exceptions to the rules to be marked avoiding permanently raising warnings known to be false positives.

We find that PMD has some overlapping with some of Eclipse's built-in static analysis but we don't see that as a real disadvantage: the checks can be turned off and PMD may still be run to check them during a continuous integration build (with Apache maven, for instance).

By itself, PMD will not ensure quality of a program not will it prove any characteristic of a program like Plural does. It is, however, non-intrusive and can (should) be combined with other static analysis tool.

We would recommend using PMD on every Java project, with a configuration which has to adapter to the level of expertise of its users.

References

[1] PMD homepage:

http://pmd.sourceforge.net

[2] PMD complete list of current rules with descriptions:

http://pmd.sourceforge.net/rules/index.html

[3] PMD instructions for writing rules:

http://pmd.sourceforge.net/howtowritearule.html

[4] Plural Analysis Tool:

http://code.google.com/p/pluralism

[5] Maven homepage:

http://maven.apache.org

[6] Ant homepage:

http://ant.apache.org

Appendices

Additional true positives discovered

Gaming framework

- In Class `aialg.Checkers.CheckersBasicMove.`, line 345, PMD complains on the following code:

  ```
  if (o == null || !(o instanceof CheckersBasicMove)));
  ```

  Stating that there is no need to check for null before an `instanceof`, because this keyword already returns false when given a `null` argument. By removing the first comparison, we would be improving the overall performance of the code, and reducing its complexity.

Plural

- In class `edu.cmu.cs.plural.perm.parser.FieldFPVisitorConj`, method visit, line 98, throws `RuntimeException("Unimplemented")`. PMD complains that raw exception types should not be thrown. In fact, this would probably be a `java.util.UnsupportedOperationException`.
- In class `edu.cmu.cs.plural.states.StateSpaceImpl`, line 117, an overridable method is calling in the constructor. This may lead to a bug if the class is extended.

Additional irrelevant true positives

Plural

- Class `edu.cmu.cs.plural.perm.parser.BoolLiteral`: this class contains only private constructors and PMD says it should be declared as final. In fact that is true since it allows some optimization, but this is not a relevant issue for Plural.