

# Software Engineering IIB

COMP2501

Assignment 2

Semester 2, 2002

Igor Kromin 40125374

Sean Ness 40120025

Raja Pantelic 40089177

## Table Of Contents

1.0 Introduction.....	3
2.0 PMD Description .....	4
3.0 Packaging.....	5
4.0 Classes.....	7
4.1 TypeSet Class.....	7
4.2 Renderers .....	8
4.3 PMD Class .....	8
4.4 Rules .....	8
4.5 RuleViolation Class .....	9
4.6 Report Class .....	10
4.7 Symbol .....	10
4.8 SymbolTable .....	10
4.9 Namespace .....	11
4.10 ExternalRuleID .....	11
4.11 RuleSetNotFoundException .....	11
4.12 RuleSet.....	11
4.13 RuleSetFactory.....	12
4.14 RuleContext .....	13
5.0 Relationships.....	14
6.0 Design Patterns .....	17
6.1 Factory Pattern (RuleSetFactory) .....	17
6.2 Adapter Pattern (Rule) .....	17
6.4 Proxy Pattern (AbstractRule).....	18
7.0 Future Research .....	<b>Error! Bookmark not defined.</b>
7.1 Proposed Extensions .....	<b>Error! Bookmark not defined.</b>
7.2 Further Analysis of PMD.....	<b>Error! Bookmark not defined.</b>
7.3 Test Plan.....	<b>Error! Bookmark not defined.</b>
8.0 Reflection.....	<b>Error! Bookmark not defined.</b>
8.1 Igor Kromin 40125374 .....	<b>Error! Bookmark not defined.</b>
8.2 Sean Ness 40120025 .....	<b>Error! Bookmark not defined.</b>
8.3 Raja Pantelic 40089177 .....	<b>Error! Bookmark not defined.</b>
9.0 Journal.....	<b>Error! Bookmark not defined.</b>
10.0 CVS Use Evidence.....	<b>Error! Bookmark not defined.</b>
11.0 Build Evidence.....	<b>Error! Bookmark not defined.</b>

## 1.0 Introduction

This report will describe the design of a java source code analyser tool, PMD. The main focus will be on the central classes in the project. Hence, for the purposes of this report, some packages will not be covered.

The description will start with looking at the packaging within the PMD project. After this has been examined, the focus will shift to the more important classes of PMD. Each of the major classes will be individually discussed.

Subsequent to this, the relationships amongst the discussed classes will be presented with the aid of UML class diagrams that will depict the dependencies. Any existing design patterns shall be discussed to familiarise the reader with the internal structure of PMD.

As a convention for this report, we will use the following symbols to denote classes in our UML diagrams. The public, private and protected functions/attributes will be shown with their corresponding symbols as below.



Interfaces will be denoted with the symbol below and similarly, for their attributes, the symbols used are shown.



## 2.0 PMD Description

PMD is a tool that is designed to analyse Java source code and detect any potential slip-ups that a programmer might have made. These will include things like:

- Unused local variables
- Empty catch blocks
- Unused parameters
- Empty 'if' statements
- Duplicate import statements
- Unused private methods
- Classes which could be Singletons
- Short/long variable and method names

Several issues must be considered in this design. First of all, since PMD tries to identify various patterns within the source code, these patterns must be detectable in some way. Second, it must be easy to add additional patterns to be checked. And third, the user must be able to check for an arbitrary number of patterns during any given run of the program.

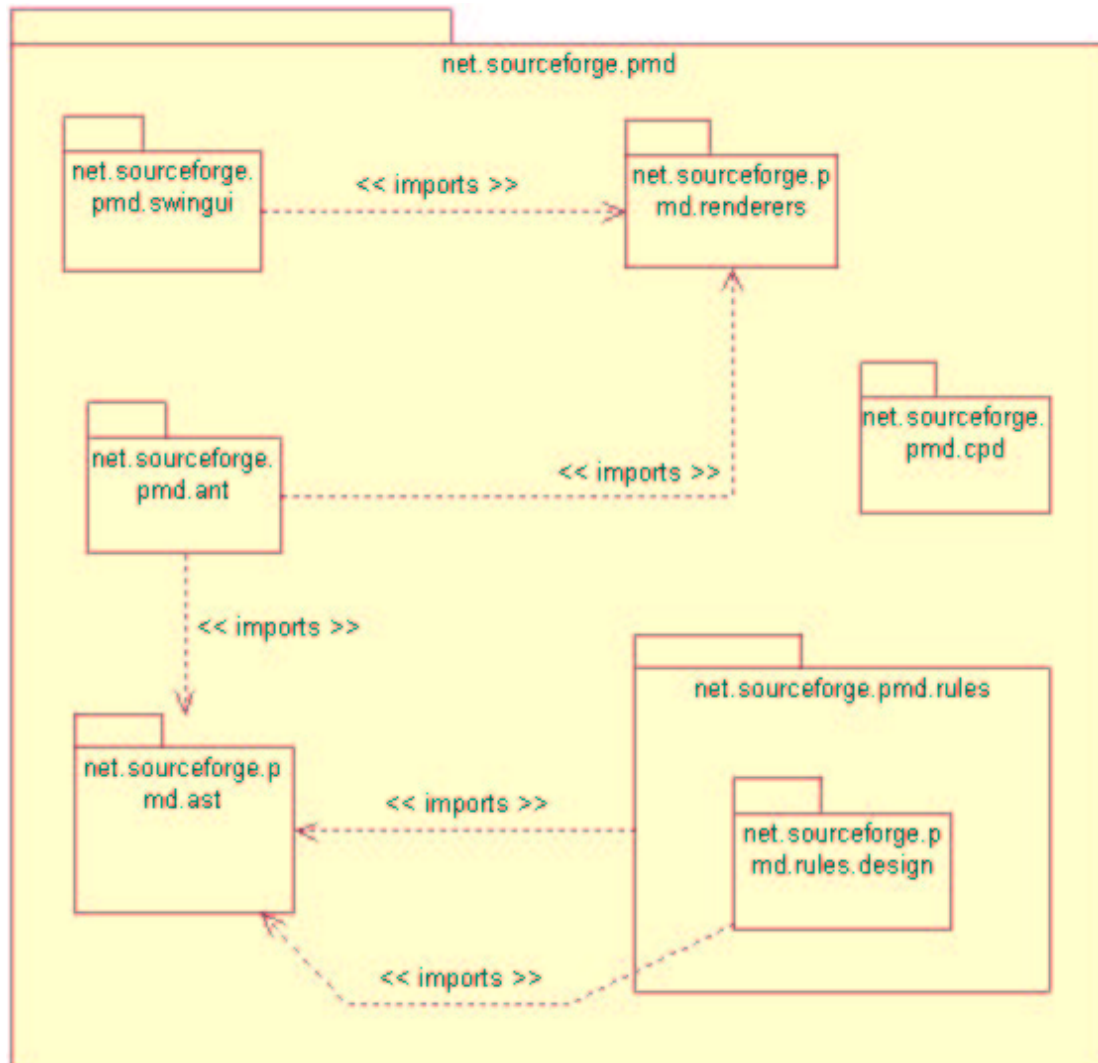
The first requirement is met by using Abstract Syntax Trees to generalise the Java source code into a more hierarchical structure. This structure can then be easily traversed to detect any inconsistencies in the generated tree.

The second requirement is met by making a factory class that would construct concrete classes that will check for different patterns. Each of these classes extend an abstract class and therefore make it easy to be used by other classes that can handle working with the abstract class.

The last requirement is met by using files that tell which rules should be loaded for checking. This is done with the use of XML files.

### 3.0 Packaging

The classes within PMD are split into different packages that distinguish their roles and the tasks they perform. The general structure of packages is detailed below. Note, the rules package separates rules into different categories too.



The focus of this report will primarily be on the contents of the net.sourceforge.pmd package. The only classes that will be largely examined will be the ones located directly in this package. Details of the inner packages and their classes will be omitted, although their purpose will be explained.

net.sourceforge.pmd – This package contains the main classes that deal with loading the rules and parsing the configuration files as well as classes that actually make use of other packages within PMD to make the application work.

net.sourceforge.pmd.ant – This package provides an interface between PMD and Ant.

net.sourceforge.pmd.ast – This package provides the necessary classes to parse a java source file and built an Abstract Syntax Tree from it.

net.sourceforge.pmd.cpd – This package takes care of tokenising files.

net.sourceforge.pmd.renderers – This package contains classes that take care of outputting the results of running the tests in an appropriate format (ie XML or HTML).

net.sourceforge.pmd.rules – This package provides the default rules that come with PMD.

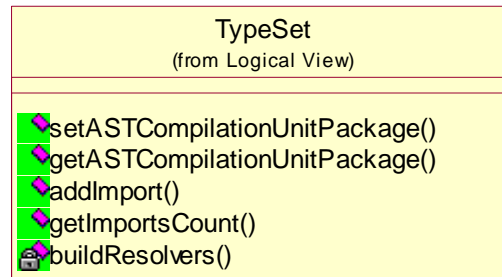
net.sourceforge.pmd.rules.design – This package provides the default design rules that come with PMD.

net.sourceforge.pmd.swingui – This package is intended to provide a graphical interface for PMD, however, at this stage, it is not fully functional.

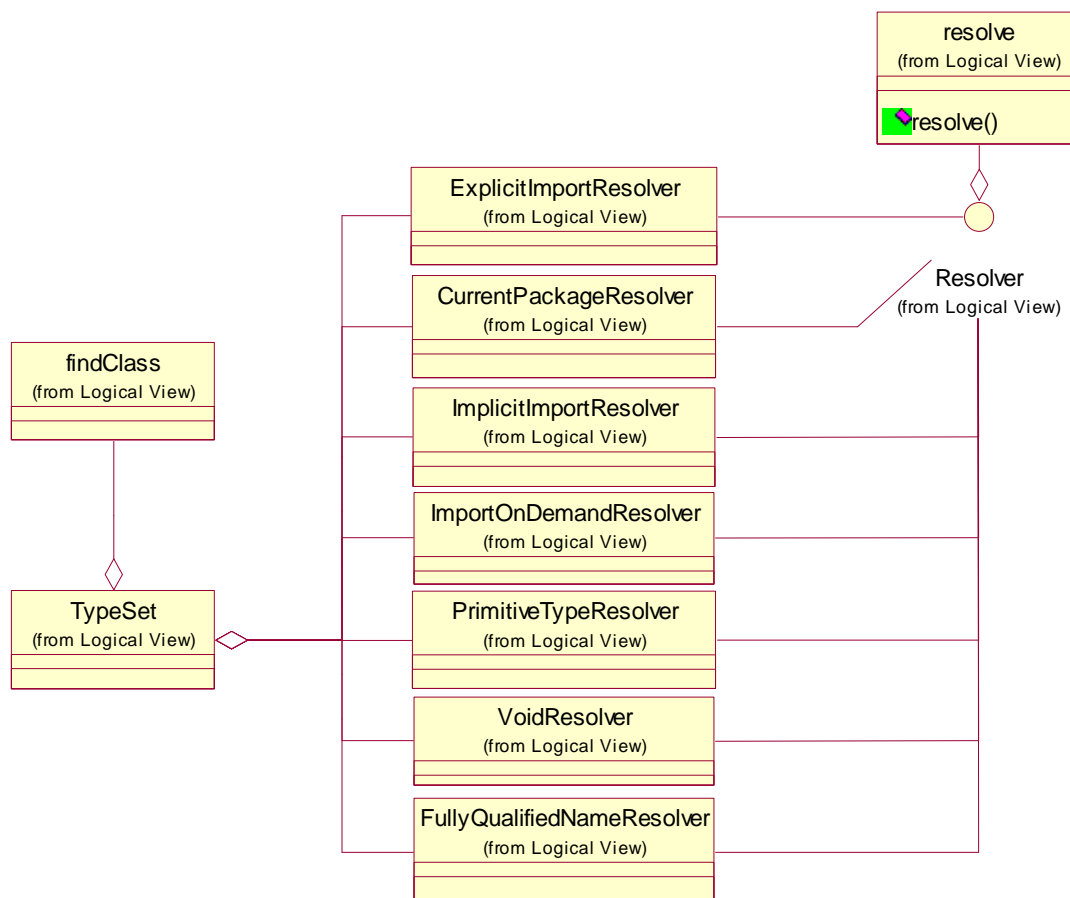
## 4.0 Classes

### 4.1 TypeSet Class

This class is used by one of the rules that are supplied with PMD and deals with keeping track of the types that were encountered within the source code. It has inner classes that deal with specific types.

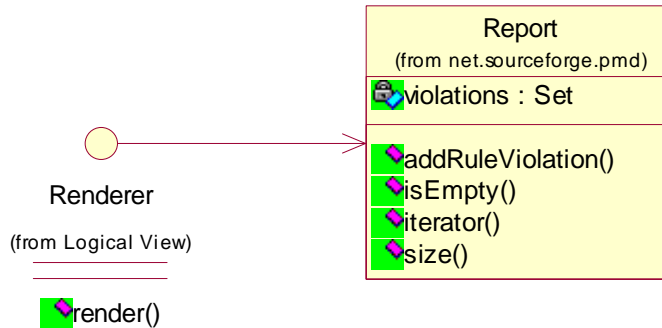


The way that the inner classes are used, they are parts of the larger class and are not used by other classes individually. A simple builder pattern is used when a resolver table is built. The TypeSet class is responsible for instantiating each of the resolvers and creating an array out of them.

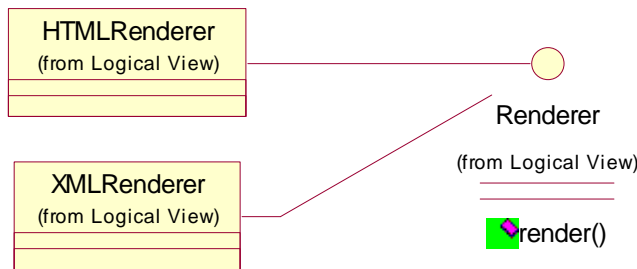


### 4.2 Renderers

The Renderer interface simply defines a single method for the two concrete classes that implement it. This method deals with formatting output in the correct manner. The input to this method is the Report class.

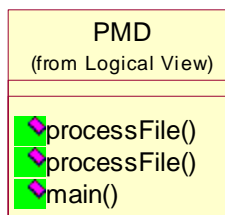


The two possible renderers are HTML and XML renderers. These simply take care of making the output readable by either web browsers or XML browser tools. They simply output the text to a String buffer.



### 4.3 PMD Class

This is the main class within the PMD application. It deals with reading in rule set files and the java source file that it is analysing. It is largely dependent on other classes within the *net.sourceforge.net.pmd* packages.

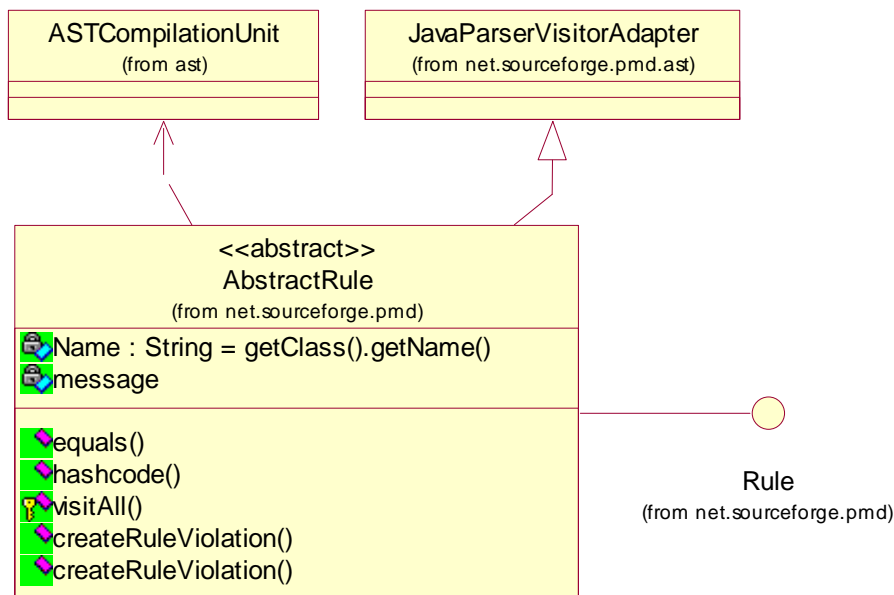


### 4.4 Rules

The rule interface defines the functions that each of the rules that PMD will understand must implement. It itself is not dependent on anything else in the PMD packages, however, its abstraction is.

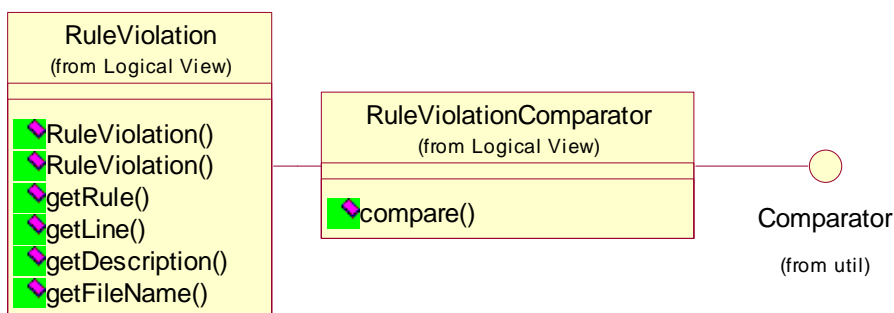


Adding a level of indirection, the AbstractRule class implements the Rule interface and also defines a few additional things. It extends the JavaParserVisitorAdapter class, allowing it to be able to communicate with all of the possible ASTCompilationUnit classes i.e. visiting each node in the generated AST tree.



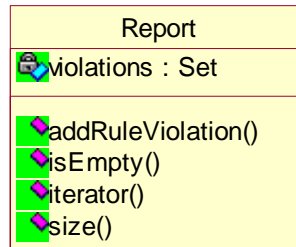
### 4.5 RuleViolation Class

This class encapsulates the information needed for each violation of the PMD rule being checked. It has a Comparator class that takes care of comparing two distinct RuleViolation classes.



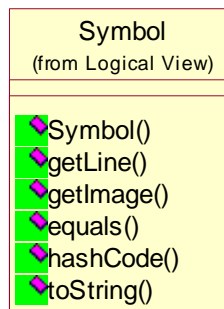
#### 4.6 Report Class

This class simply keeps a tree of all of the violations in the java source file. There will only be one instance of each violation recorded even if multiple violations of the same kind occurred.



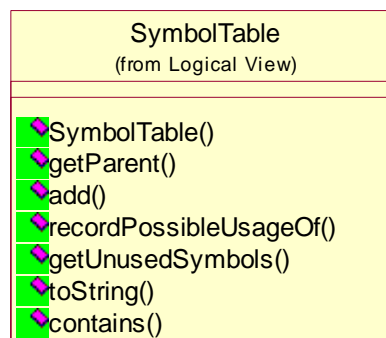
#### 4.7 Symbol

A symbol is a string of code from the body of the program, as well as the line number. These are stored as one object, a symbol. The symbol can be compared to another Object to see whether it is identical (in the equals() method).



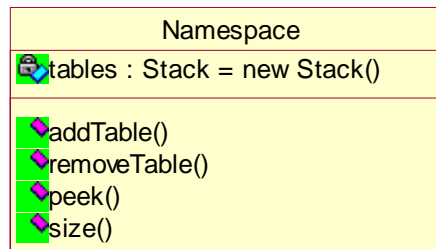
#### 4.8 SymbolTable

Stores a collection of symbols into a HashMap. Each symbol can only be stored once. It can then convert the symbols to strings and find symbols in the file that have not been used.



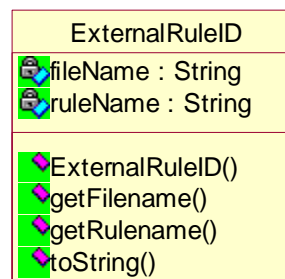
## 4.9 Namespace

Namespace keeps a stack of SymbolTables for the convenience of certain rules.



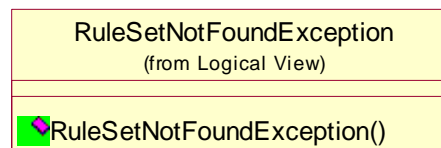
## 4.10 ExternalRuleID

ExternalRuleID keeps a filename (from which the rule is taken) and a rulename for each individual rule. This data is kept external of any other data structures in the program.



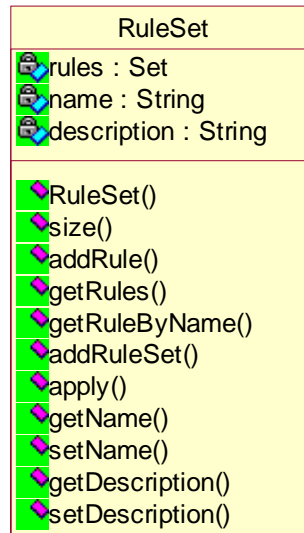
## 4.11 RuleSetNotFoundException

An Exception called upon when a particular RuleSet cannot be found.



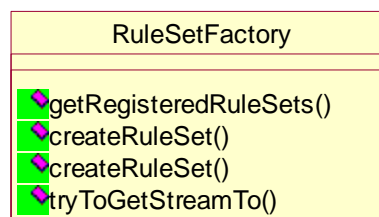
## 4.12 RuleSet

A RuleSet is a collection of individual rules stored in a HashSet data structure. Each rule is an instance of the Rule class, and when stored in RuleSet, each rule has an associated name and description - both of which are type string. RuleSet is fully searchable with setter and getter methods.

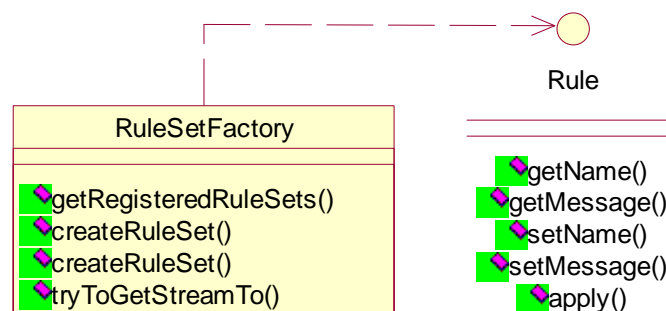


### 4.13 RuleSetFactory

RuleSetFactory is used to generate various instances of the RuleSet class. It takes an input stream or a string and builds it into a type of RuleSet, the specific nature is determined by the input itself.









RuleSetFactory uses objects that implement the Rule interface.



#### 4.14 RuleContext

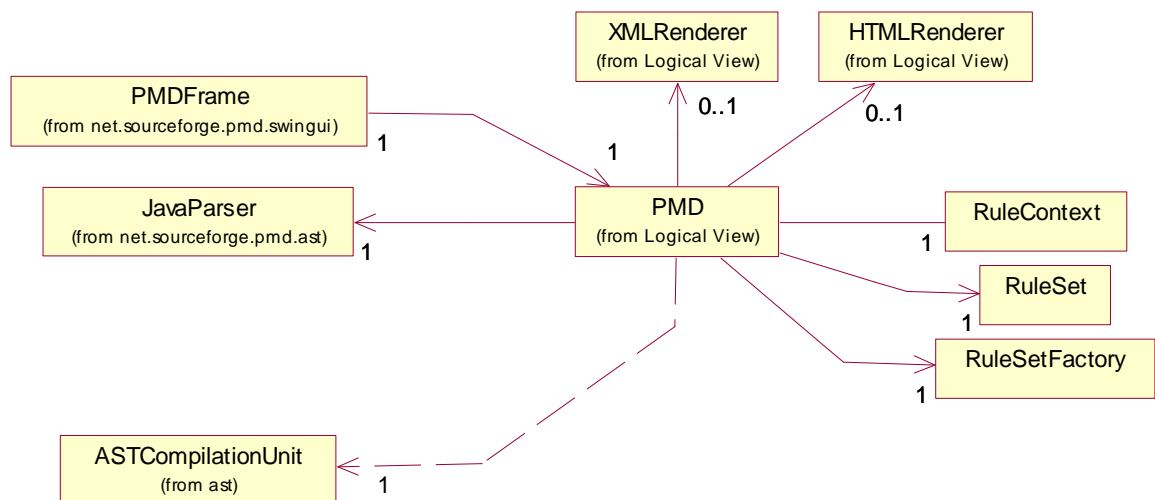
Contains a report and string, the string indicating the filename of the source code of origin. Places source code, indicated by the string, in the context of a report.

RuleContext	
	report : Report
	sourceCodeFilename : String
	getReport()
	setReport()
	getSourceCodeFilename()
	setSourceCodeFilename()

## 5.0 Relationships

PMD is related to numerous other classes in various packages. PMD calls upon so many packages because it is arguably the most important class in the program. It uses JavaParser to parse the java source code. HTMLRenderer and XMLRenderer are used to output the results of running PMD in an appropriate format. RuleContext, RuleSet and RuleSetFactory are used to construct appropriate sets of rules to apply upon the source code being tested.

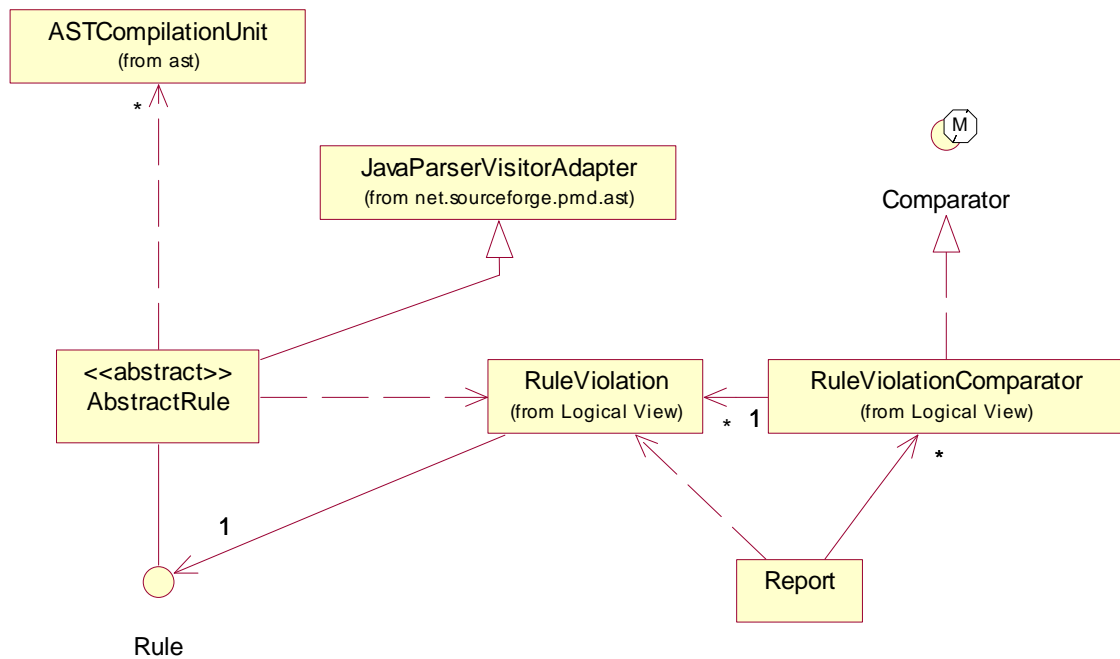
PMDFrame encapsulates the PMD class by creating an instance of it and applying the instance to a specified file containing Java source code. When applied, the PMD instance will perform the required rule checks.



AbstractRule is an abstract class that implements the Rule interface; any rules that are to be applied must extend AbstractRule. The AbstractRule class also extends the JavaParserVisitorAdapter so that it may visit all nodes contained within the AST tree. For example, it must call upon ASTCompilationUnit to search the source code for patterns which do not match the specifications of the specific rule, which implements it.

When a specific rule has been violated by the source code, a RuleViolation object is created. RuleViolationComparator – which implements the Comparator interface and is a nested class of RuleViolation - can be used to compare two violations.

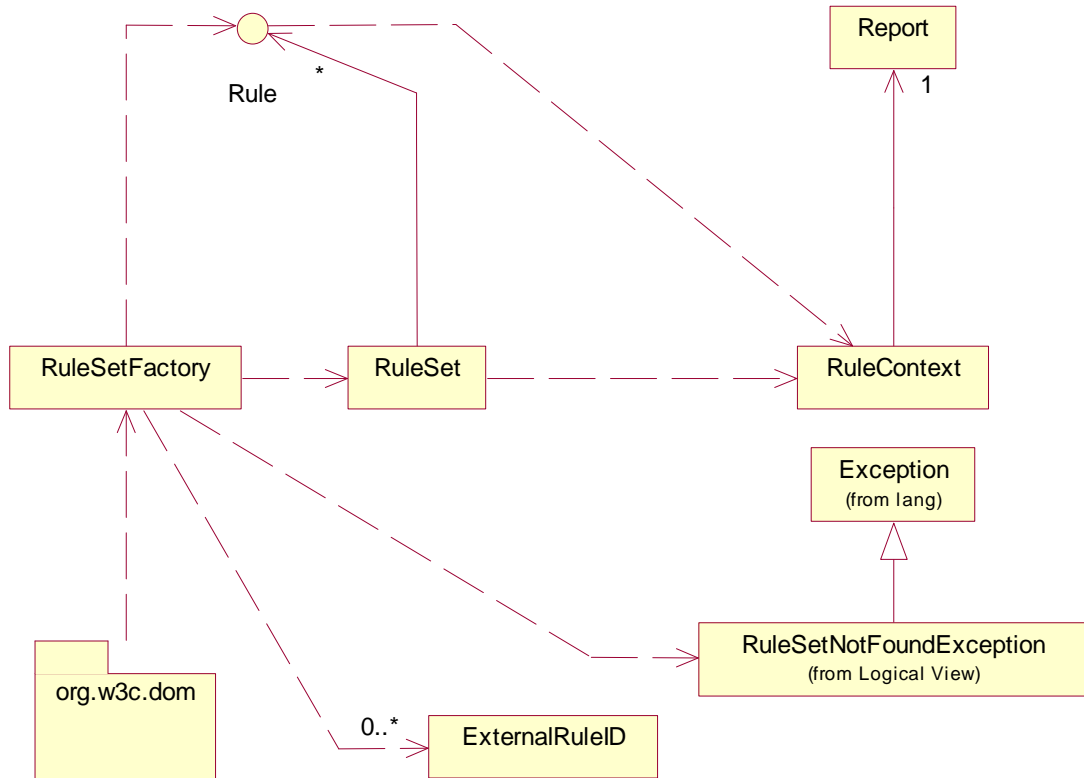
Finally, Report stores a set of RuleViolation objects for display by the renderer classes.



RuleSetFactory generates new RuleSet objects; RuleSet objects, in turn, are collections of Rule objects. All three are therefore related to one another in that order. Additionally, RuleSetFactory calls upon Rule for various clerical uses.

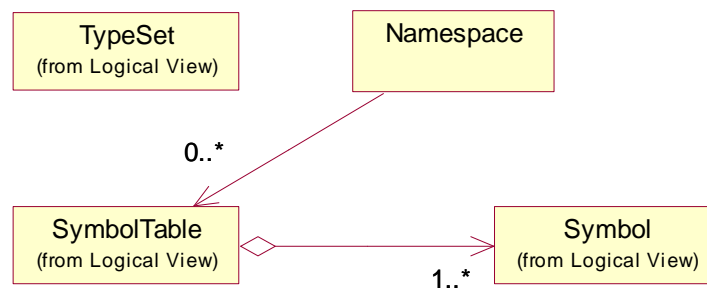
ExternalRuleID is called upon by RuleSetFactory to provide external documentation for each rule. It also contains rule setter and getter methods, among other functions; if there are any conflicts, a RuleSetNotFoundException may be thrown.

Because each RuleSet must have a description, it also calls upon RuleContext. RuleContext also stores its data as a Report object. Finally, RuleSetFactory makes use of the W3C DOM packages to parse the rule set XML files.



Although not detailed in this report, the following classes are used by some of the rules that PMD provides. For example, *TypeSet* class is used by the *DuplicateImportsRule* class to keep track of the reference variables within the source code.

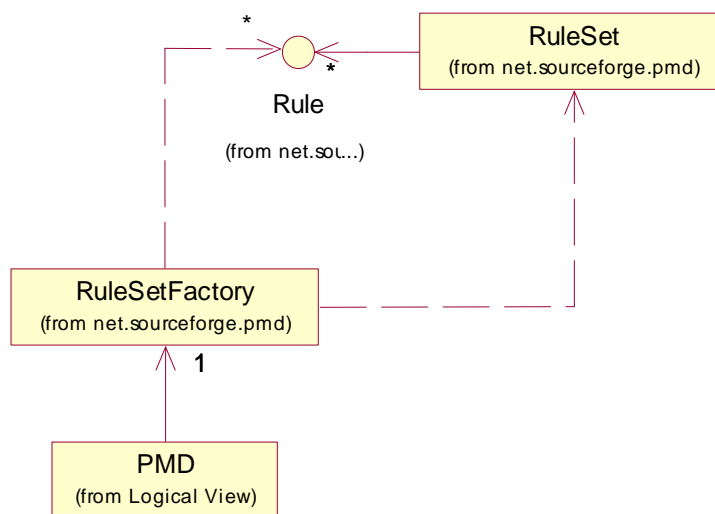
Also, the *UnusedLocalVariablesRule* class uses *Symbol* instances to check the source code for unused variables. *SymbolTable* is a container of *Symbol* objects, and *Namespace* is a stack of *SymbolTable* objects.



## 6.0 Design Patterns

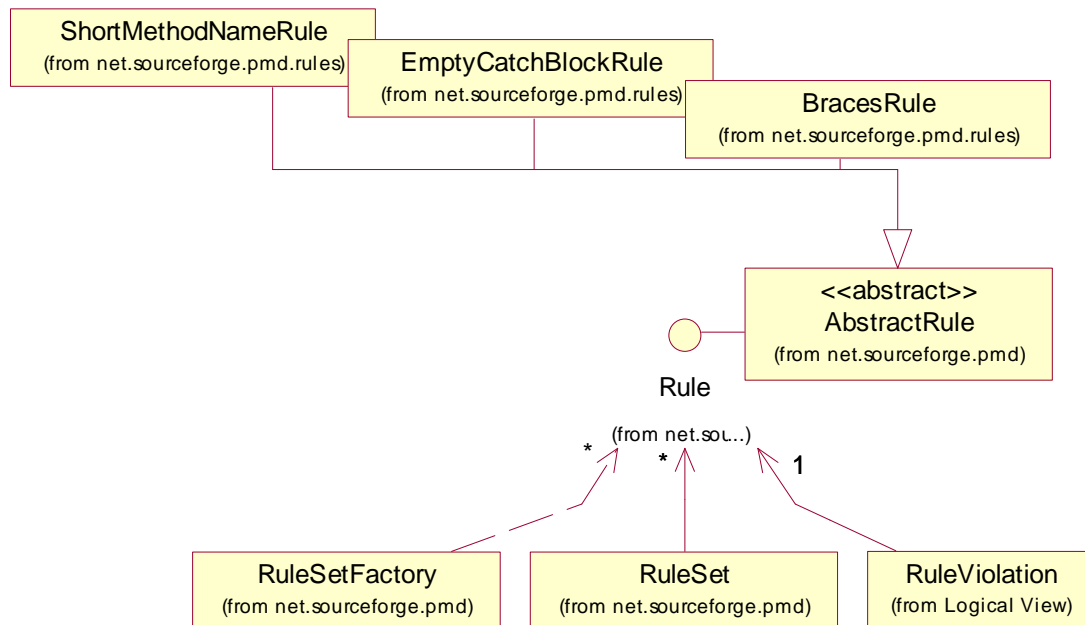
### 6.1 Factory Pattern (RuleSetFactory)

The factory method is used to create new instantiations of classes, and it is used primarily when a class cannot anticipate the class of objects it must create. This is true of the PMD factory class RuleSetFactory. This class creates an Iterator containing RuleSet objects. A factory class is simpler because the exact type of RuleSet objects –e.g. the sets of rules that are being implemented – are not important to the running of the class; it will create the iterator and its objects no matter what. It would be far too logically complex to devise a class that created classes differently when different rule sets are needed. It is much more efficient to put all that detail in a single factory class. Thus, by implementing the factory design pattern, the RuleSetFactory class serves to greatly simplify PMD's logical complexity.



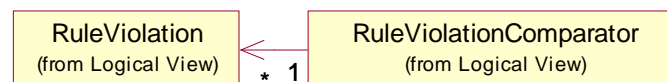
### 6.2 Adapter Pattern (Rule)

The Rule interface is an example of the adapter design pattern. Adapter patterns are used on interfaces when the goal is to create a single, usable type that is recognised by all the classes that call upon it. Many other classes extend AbstractRule, so it in turn implements Rule, and in doing so, can be used in all classes that call upon the Rule interface, such as PMD and RuleSetFactory.



### 6.3 Singleton Pattern (RuleViolationComparator)

A singleton is a design pattern that guarantees only one instantiation of a class. The RuleViolationComparator class can be classified as a singleton, because it is declared to be static. RuleViolationComparator is a nested class, inside the RuleViolation class; however, it is clear from the structure of RuleViolation that RuleViolationComparator can only be instantiated once inside every instance of the former. Hence, RuleViolationComparator can be considered a singleton design pattern.



### 6.4 Proxy Pattern (AbstractRule)

The Proxy pattern is responsible for facilitating message passing amongst several different types of classes through a common interface. The AbstractRule in conjunction with the JavaParserVisitorAdaptor implement this design pattern to allow the AbstractRule class to access everything in the AST package directly.

