CODE QUALITY ASSURANCE WITH PMD

An extensible static code analyser for Java and other languages

BY: Andreas Dangel
Software Engineer
MicroDoc GmbH

Developing new software is great. But, writing software that is maintainable and sustainable is not so easy. Luckily, there are tools available that can help you achieving better code quality. One of these tools is PMD.

```java
public class Example {
    void run() {
        try {
            // do something
        } catch (Exception e) {
            e.printStackTrace(); // Not good - this exception should be logged via a logger.
        }
    }
}
```

Figure 1: Example for AvoidPrintStackTrace

WHAT IS PMD?

PMD is a static source code analyser. It scans your source code and searches for patterns, that indicate problematic or flawed code. Sometimes it is just a too complex solution which might increase maintenance costs in the long term or it might be a indication of a real bug. In that sense, PMD can be seen as another pair of eyes, that reviews your code.

For example, PMD can be used to find usages of the printStackTrace() method, which is often generated by the IDEs when surrounding a statement with a try-catch-block. Just printing the stacktrace might result in swallowing the original exception, since the output might end up somewhere. Ususally such output should be logged with the appropriate logging framework. PMD provides the rule AvoidPrintStackTrace, which detects such cases. See figure 1.

The abbreviation "PMD" is not exactly defined, it is actually a backronym. But "Programming Mistake Detector" or "Project Mess Detector" are the most logical meanings. However, the tool is usually known and referred to simply as "PMD", sometimes with the tagline "Don't shoot the messenger". See Figure 2 for the official logo.

Figure 2: The PMD logo

The patterns, that PMD is searching for, are defined by rules. PMD is shipped with more than 250 built-in rules, that can be used immediately.

When the rules detect a problematic piece of code, a rule violation is reported. Furthermore, own rules can be developed in order to adapt PMD to specific project requirements. With so many possible rules, it is clear, that one cannot simply enable all rules. Some rules even contradict each other. And some rules just have different coding conventions in mind, that might not be suitable for the concrete project at hand.

In the field of code analysers and so called linters, there are other products available. For Java projects, often checkstyle is used in order to enforce a common (project- or company-wide) code style. Having a common code style helps a lot if multiple developers working together on the same project, since each part of the project is then be read and skimmed as easy as any other part - regardless of the author. Checkstyle concentrates on the source code directly including whitespace checks like correct indentation and also documentation via JavaDoc comments.

PMD doesn't support whitespace checks, but it has basic support for comments, like enforcing the existence of JavaDoc comments for classes or fields. Other tools like FindBugs and its successor SpotBugs are analysing the compiled bytecode of Java projects instead of the source code. They have therefore access to the compiler optimised code and might see slightly different code. Moreover, SpotBugs can rely on the structure of a classfile and does not need to deal with syntax errors. SpotBugs can only be used after the project has been compiled, while Checkstyle could be run before.

PMD can be seen in between these two tools: While the starting point for PMD is also the source code, PMD takes advantage of the compiled classes. This feature in PMD is called "type resolution" and it helps PMD to understand the analysed source code better in order to avoid false alarms. E.g., if PMD knows the return type of a method call, rules can be written that only apply to a specific type. Otherwise, the rule would need to "guess" and assume the type by looking at the type name only and do a simple string comparison. If the project has an own class with the same name, then we might mix up the classes. A concrete example can be seen in unit tests: PMD provides several rules for JUnit. But if the project uses a different test framework with the same class names (but obviously different packages), then these rules would find issues, which are maybe irrelevant for the other test framework.
There are other big players for code quality tools on the market like SonarQube that support a more integrated solution to also monitor quality improvements or regressions over time.

When PMD is integrated into the build pipeline, it can act as a quality gate. For example, if rule violations are detected, the build can be failed or the commit can be rejected. This can be used to enforce a specific quality goal. The build pipeline could also be configured to only make sure, that no new rule violations are introduced, so that the code quality doesn't degrade and hopefully improves over time.

There is one other component in PMD, that is often overseen: CPD - the Copy-Paste-Detector. This is a separate component, that searches for code duplications in order to follow the DRY principle (Don't Repeat Yourself).

OVERVIEW / HOW DOES IT WORK?

PMD analyses the source code by first parsing it. The parsing process consists of the two steps:

- lexing, which produces a stream of tokens
- and parsing, which produces an abstract syntax tree (AST).

This tree is the equivalent representation of the source code and has the root node "Compilation Unit". In Java, you can define multiple types in one source file (as long as there is only one public) and

>>

```
public class Example {
    public void method1() { }

    public void anotherMethod() { }

    public static class NestedClass { }
}

class OtherClass {
}
```

Figure 3: Source code of the AST example

classes can be nested. Classes itself can have methods, which in turn have zero or more statements. Figures 3 and 4 show a simple java class and the corresponding AST.

When the source code could be parsed to an AST, then the syntax is correct. Nowadays, it is recommended to use PMD after the project has been compiled in order to take advantage from type resolution. This means that PMD can concentrate on valid syntax, e.g. if the parsing fails, the analysis of this source file is simply skipped. Technically an own grammar for JavaCC is used to implement the parser for the Java language. Therefore, failing to parse a specific source file might actually indicate a bug in PMD's own Java grammar and does not necessarily mean, that the source code is not valid.

After that, the AST is enriched by a couple of visitors: First, the qualified names for the types, that are defined in the source code, are determined. This is later helpful when referencing this class (and its nested classes and lambdas) itself. Second, the symbol facade visits the AST. It searches for the fields, methods and local variables and looks up their usages within the scope of this source file. The information collected in this step is made available to the rules, e.g. they can easily figure out, if a (private) field or method is used or not. The found variables are organised in different scopes, that are nested. The third visitor is the "Data Flow" facade. It's goal is to follow variable definitions, assignments and reassignments and their accesses throughout the program flow. It allows to detect anomalies such as assigning a new value to a variable after it has been accessed. It's currently limited to a single method. The last visitor is the "Type Resolution" facade. It traverses the AST and resolves the concrete Java types of variable declaration, method parameters, and classes whenever a referenced type is used. It uses the compile-time classpath (also known as the auxiliary classpath) of the project that is being analysed.

Now, after the AST has been created

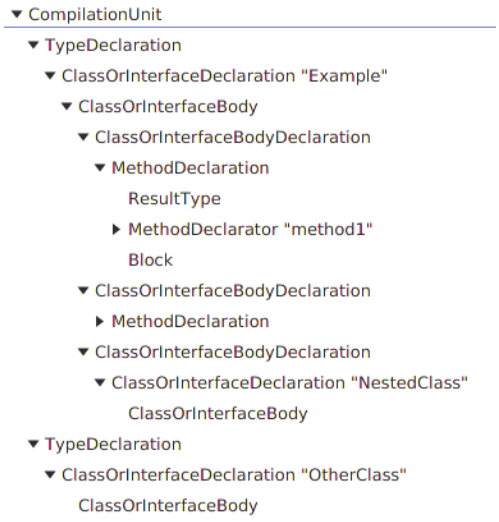

Figure 4: AST example

and filled with additional information, the rules are executed. While all rules for one file are executed one after another, the analysis of multiple files (and ASTs) is executed multi-threaded. Each rule has the possibility of reporting rule violations, which are collected in reports. The violation contains the information about the rule, the location (like line and column in the source file) and a message. In the end, the reports are transformed into the desired output format, such as XML or HTML.

When utilising PMD for a project, there are a few different approaches possible. For greenfield projects, it's a no-brainer: PMD is active with a basic set of rules from the very beginning. So, every code, that is added, will be checked by PMD. For projects with an existing code base, the situation is most likely different. It can be overwhelming, if a whole bunch of rules are activated at once. You might be drowning in violations and it is not clear, which one to fix first. For this situation, an incremental approach is recommended: Prioritising and enabling one rule at a time.

Alternatively, all the selected rules can be enabled at once and the current number of violations are monitored. The goal is then, to reduce the violations with every commit and not introduce new violations. This however requires support from the build environment and is not possible with PMD alone. But it can

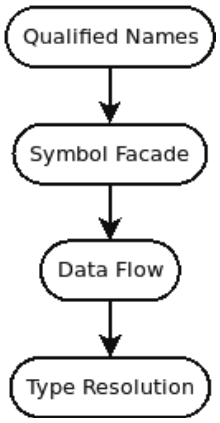

Figure 5: Visitors

be implemented using a quality gate in SonarQube.

PMD should be integrated into the development process as early as possible. The earlier PMD is used, the less issues need to be fixed later on. Therefore there are also IDE plugins that execute PMD while developing code. For Eclipse, there are today 3 different plugin implementations:

- The official pmd-eclipse plugin
- eclipse-pmd
- and the qa-eclipse-plugin.

For other IDEs and editors, there are plugins, too. For the full list, see the Tools

/ Integrations documentation page. Especially if your project is using Apache Maven as the build tool and you are using Eclipse, you should have a look at m2e-code-quality plugins, which transform the configuration from your Maven project files and make them available for the PMD, Checkstyle and Findbugs plugins in Eclipse. This means, you can configure your code quality tooling within your build tool and it is automatically working in Eclipse.

To compile, build and package software projects, usually build tools are used, such as Apache Maven, Gradle or Ant. For Ant, PMD provides an own task, that can be used. For the other build tools, plugins are existing, that can execute PMD. And most importantly: these plugins can fail the build, acting as a simple gate keeper. The Maven PMD Plugin can create a report for the project site and also contains a check goal, to fail the build, if PMD rules are violated. It also supports CPD, the copy paste detector.

All the previous tools are good, if you are building the project locally. But if a whole team is working on the project together, there is usually a central continuous integration server. Basically, such CI servers could just execute the build tool with its configuration for PMD, but they often provide a little bit more support for code quality tools like PMD: Since they regularly build the project and can keep a history, they allow to compare the reports generated by PMD from build to build. This allows you to see the development of the code quality over time like new introduced violations or violations that are resolved. For Jenkins, there is a PMD Plugin available, which produces a simple graph of violations.

Nowadays, such CI servers are available as a service, too. Especially for open source projects they are often free to use. PMD itself uses e.g. Travis CI. GitHub as a code hosting platform provides integrations with various 3rd party services, that can be enabled. Two such services already use PMD to offer their service: Code Climate and Codacy. These services can also be integrated for verifying pull requests to get early feedback. Since these service also create a history, you can see the results over time.

PMD provides many different built-in rules. Since PMD 6, these rules are organised into 8 categories: Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance, and Security. The recommended approach is, to create an own ruleset, which references the rules that should be used for the specific project. This ruleset should be part of the project, so that it can be easily shared between developers and build tools. For Maven projects, often an extra module with the name “build-tools” is created, which can be used as a dependency. This is described in the Multimodule Configuration for the maven-pmd-plugin.

You might also find yourself in a situation, that you need a very specific rule, which is not available in PMD itself. Since it is very specific to your project, it might not be even useful outside of your project. Therefore you can define own rules, and the code for these custom rules naturally goes into the “build-tools” module as well.

The ruleset can also contain project wide file exclusion patterns, e.g. if you don’t want to analyse generated code.

While referencing the existing rules in your ruleset, you can configure them exactly to your needs. Many rules can be easily customised via properties. The rules also define the message, that appears in the report, if a violation is detected. This message can also be overridden and customised. A typical customisation is the priority. You can give each rule a specific priority and during the build, you can decide to fail the build because of an important rule violation but ignore other rules. You can also add own rules. See Figure 6 for an example of a custom ruleset.

FEATURES

It’s now time to look at a few selected

```
<?xml version="1.0"?>

<ruleset name="Custom RuleSet"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

    <description>
Custom RuleSet
    </description>

    <!-- Just use the rule DuplicateImports as is -->
    <rule ref="category/java/codestyle.xml/DuplicateImports"/>

    <!-- Use the rule EmptyCatchBlock, but raise the priority -->
    <rule ref="category/java/errorprone.xml/EmptyCatchBlock">
        <priority>1</priority> <!-- this is high priority -->
    </rule>

    <!-- Use the rule CyclomaticComplexity and configure the thresholds -->
    <rule ref="category/java/design.xml/CyclomaticComplexity">
        <properties>
            <property name="classReportLevel" value="40"/>
            <property name="methodReportLevel" value="5"/>
        </properties>
    </rule>

     <rule name="AvoidPublicField"
           language="java"
           message="Fields, that are not constants, should be private"
           class="net.sourceforge.pmd.lang.rule.XPathRule">
         <description>
     Data encapsulation would be broken, if fields are accessible from outside. Declare the field as private and use getters and setters to allow access if needed.
         </description>
         <priority>2</priority>
         <properties>
             <property name="xpath">
                 <value>
 <![CDATA[
 //FieldDeclaration[@Public='true'][@Static='false'][@Final='false']
 ]]>
                 </value>
             </property>
         </properties>
     </rule>
 </ruleset>
```

Figure 6: Example of a custom ruleset

features, that PMD provides. The first feature is the support for XPath based rules. Since the AST is a tree structure, it can be dealt with like a XML document. The document can then be queried using XPath expressions, to find nodes within the AST, that match certain criteria. This provides an alternative API to develop rules, if you don't want to implement a rule using the visitor pattern to traverse the AST. This is a very convenient way to create ad-hoc rules. There is even a graphical rule designer to make it easier to develop XPath rules. The designer shows the parsed AST and executes a given XPath query. You can see the matched nodes directly. In the end, the developed XPath expression can be exported as a custom PMD rule in XML format, that you can add to your own ruleset. Since the rule designer displays the AST, it is also a valueable tool for developing rules in Java using the visitor pattern. See Figure 7 for a screenshot of the designer. This way of providing access to the AST and reuse XPath to write custom rules is a unique feature of PMD, that does not exist in other static code analysers.

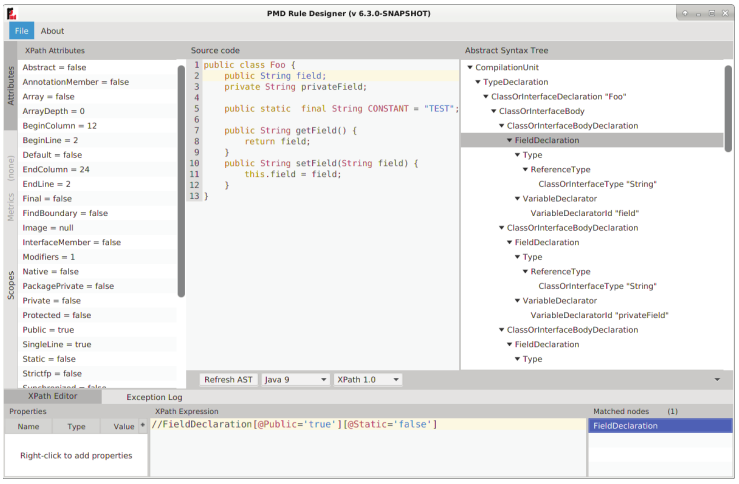

Figure 7: PMD Designer

Another feature of PMD is the so called type resolution. As explained above, type resolution happens as an extra step after parsing the source code. The goal is, that the AST is enriched with concrete type information whenever possible. Consider the following source code:

```
import org.slf4j.Logger;

public class Example {
    private static final Logger LOG = Logger.getLogger(Example.class);

    public void someMethod(String arg) {
        LOG.debug("This is a debug logging message: " + arg);
    }
}
```

Via type resolution, the field declaration for `LOG` is assigned the type `Logger`, which (through the import) is identified as `org.slf4j.Logger`. If the library "slf4j-api" is on the auxiliary classpath, then PMD can attach a concrete instance of `Class<org.slf4j.Logger>` to that node in the AST and the rule can access it. The rule can now first verify, that this field really is a logger, instead of simply relying on naming conventions of the field name or the simple class name. This helps greatly to reduce false positives for rule violation detection. In the example code snippet, PMD is correct to suggest to use the slf4j placeholder syntax (`"... message: {}", arg`), but PMD would be wrong, if the logger would be of a different type. Since the rule has access to the concrete class instance, it can even use reflection to gather more information as needed. This type resolution does not only work for 3rd party libraries, but in the same way it works within the same project, that is being analysed by PMD. That's why it is necessary, that the project is compiled first before PMD is executed. This means that references to other classes within the same project are resolved exactly the same way and the concrete class instances are made available.

There are a couple of rules, that make use of type resolution. And more rules will make use in the future, since type resolution is enabled by default for new Java rules. For example, the rule "LooseCoupling" finds usages of concrete collection implementations which should be replaced by the collection interface (e.g. use `List<>` instead of `ArrayList<>`). The fairly new rule "MissingOverride" actually uses type resolution and reflection to figure out, which methods are overriding methods from the super class and are missing a `@Override` annotation.

Type resolution has been available for a long time now in PMD. However, it is still under development. There are currently limitations for determining the types of method parameters, especially when overloading is in use and generics come into play.

The next feature is quite new: Metrics. It was added in 2017 during a Google Summer of Code project and provides a clean access to metrics of the analysed source code.

The metrics are e.g. access to foreign data (ATFD) or weighted method count (WMC). There are more metrics available already and the whole framework is usable by other languages, too. The metrics can be accessed by Java rules as well as by XPath rules. In the easiest case, these metrics can be used to detect overly complex or big classes, such as in the rule "CyclomaticComplexity". Multiple metrics can be combined to implement various code smell detectors such as "GodClass".

The next step in this area is to support multi file analysis. Currently, PMD looks only at one file, but for metrics it would be interesting to relate certain numbers of one class against, e.g., the total number of classes within the project. There are also benefits for the symbol table, if it has a view of the whole project. This will then allow to do full type resolution. Each rule has then access to all information which makes the rules more robust to false positives and also allows to find otherwise ignored special cases. Implementing this involves sharing data between the different file analysers - possibly involving an additional processing stage. The challenge is of course, to provide this functionality and not affecting the performance of the analysis negatively.

BEYOND JAVA

PMD started as a static code analyser just for the Java programming language only. This was the status for PMD version up to and including 4.3 (except for a little support for JSP). With PMD 5, a big refactoring took place, in order to support multiple languages. And with the initial release of PMD 5, three new languages were included: JSP, JavaScript (aka. ecmascript) and XML. Later on, support for PLSQL and the templating language Apache Velocity has been added while keeping the Java support up to date. The last big addition was support for Salesforce.com Apex.

Now, PMD supports in total 10 different languages including rules. Most rules are for Java, of course. Adding a new language takes quite some effort, but it is described in the step-by-step guide "Adding a new language". It involves in-

tegrating the language specific parser, mapping the language AST to the generic PMD interface types and last, but not least, writing new rules. Most of the PMD framework can be reused, so you'll immediately benefit from the possibility, to write XPath based rules for your language. The Copy-Paste-Detector (CPD) on the other hand supports many more languages. This is, because you only need to support a language specific tokeniser, which is much simpler than a full language grammar with productions. PMD provides even a "AnyLanguage" for CPD, which basically tokenises the source code at whitespaces. Language specific support is needed to improve the results of CPD, e.g. correctly identifying keywords and statement separators. With more effort, there is also the possibility to ignore identifier names during copy-paste-detection. This allows then to find duplicated code, which only differs in variable names, but is otherwise structurally the same. This feature however is only available for Java at the moment.

THE PROJECT

The following is a summary of the history of PMD that Tom Copeland wrote in the book "PMD Applied. An Easy-To-Use Guide for Developers". It covers the years 2002 till 2005.

The project PMD was started in Summer 2002. The original founders are David Dixon-Peugh, David Craine and Tom Copeland. The goal was to replace a commercial code checker, which these three guys were using in a government project in the US. They decided to write their own code checker and got approval to open source it. Now PMD was living on SourceForge. In November 2002, PMD version 1.0 was released with already 39 rules and a copy/paste detector. In March 2003, thanks to Dan Sheppard, XPath rules were introduced with PMD 1.04. Since PMD 1.3 (October 2003), the BSD license is used, which helped a lot to adopt it. Since then it has been integrated into many products.

The copy/paste detector has been rewritten a couple of times and improved in performance. With every release of PMD, new rules or report formats have been added and existing rules fixed. With PMD 2.0 (October 2004) the data flow analysis component has been added. With PMD 3.0 (March 2005) support for Java 1.5 was added.

Java 1.6 was added with PMD 3.9 (December 2006), Java 1.7 with PMD 4.3 (November 2011), Java 8 with PMD 5.1.0 (February 2014), Java 9 with PMD 6.0.0 (December 2017), Java 10 with PMD 6.4.0 (May 2018), Java 11 with PMD 6.6.0 (July 2018), and Java 12 with PMD 6.13.0 (March 2019).

A big step happened between PMD 4 and 5: A major refactoring took place in order to properly support rules for multiple languages. This introduced many breaking API changes and was released in 2012. Also with PMD 5, Apache Maven is being used as the primary build tool instead of Ant. Support for PLSQL was added in February 2014 with PMD 5.1.0. With PMD 5.2.0 (October 2014) the code was completely modularised into a core module and several language modules. This made it easier to add new languages. With PMD 5.5.0 (June 2016) Salesforce.com Apex has been added. With PMD 6.0.0 another small, but important refactoring took place. It has unfortunately a bigger impact on end users: All the rules have been categorised, so that they are easier to find. They have been moved into different rulesets. However, we are keeping the old rulesets for backwards compatibility, so that the existing custom rulesets still continue to work.

Over the last years, the project gradually moved more and more infrastructure from SourceForge towards GitHub. The complete subversion repository has been converted to git. It contains the full history back to the year 2002. While at the beginning every sub-project was in the same repository, we have now several separate repositories, e.g. for the eclipse plugin or other extensions.

The move to GitHub was a step forward in terms of presence and attracting new contributors. The GitHub web interface is more user friendly, easier to use and feels faster than SourceForge. GitHub especially encourages contributions through the concept of pull requests. GitHub is now the primary location for the source code and the issue tracker. On SourceForge, we still have the mailing list running and a webspace and the archive of old releases. There are other services PMD uses, e.g. travis-ci as a build server. It builds every push and deploys the snapshot via the OSS Repository Hosting service by Sonatype. For releases, this build server is even able to deploy the final artifacts directly to Maven Central.

Also, every pull request is built automatically. Other services are e.g. coveralls for test coverage and BinTray for hosting the eclipse plugin update site.

In 2017, PMD participated the first time in Google Summer of Code. This is a student stipend program offered by Google. Students all around the world have the opportunity to work during semester break on various open source projects. Open source organisations provide projects and mentors and the students apply for a project with a proposal. In 2017 two students worked on type resolution and metrics. In 2018 PMD is participating again.
As of today, the project has 3 active maintainers, about 100 different contributors, 500 merged pull requests. According to cloc it contains about 100k Java lines of code, surprisingly 88k XML LOC (which probably are the test cases) and many other types.

THE FUTURE

What's left to do for PMD? Aside from keeping the support for Java and other languages up to date and fixing bugs, adding new rules, adjusting additional rules, there are a few topics, that sound promising. In order to lower the barrier of using PMD, specialised rulesets might be useful.

There could be a "Getting Started" ruleset, that has just enough generic rules, that are useful for any project. This might be the default ruleset and could be a template for creating an own customised tailored ruleset for the project. There could also be use-case based rulesets, the group the rules not by category but by another topic, e.g. Unit testing, Logging, Migration of library usages, Android specific patterns.

Another interesting feature is autofixes. Since PMD has the knowledge, where a violation exactly is in the source code, it is for some rules trivial to provide a fix. The goal is, that PMD provides directly the fixed source code, that can be confirmed in a IDE plugin and applied automatically. Then, besides type resolution, which is still not completely finished, there is also the data flow analysis (DFA) part. PMD has a good start for the DFA, but it's still very limited. A related feature is control flow analysis. With that available, rules could be written which can detect unused code.

Or rules, that verify that a specific guarding method must be called before another method. Having the call stack available, would make this possible to verify. This requires, similar to the mentioned multi file analysis, an overview of the complete project that is being analysed.

And last, but not least, a possible future feature could be cross language support. Since PMD already supports multiple languages, this would put multi-language support onto the next level: Some languages allow to embed other languages, e.g. JavaScript inside HTML, or PHP+HTML+JavaScript. Or there is Salesforce.com VisualForce with Lightning.

When and if these features are implemented is unknown. The project is driven by volunteers and contributors and all this depends on the available time. New contributors are always welcome to work together and make PMD

WANT TO FIND OUT MORE?

Go visit https://pmd.github.io