

PMDでバグを退治する

この手軽な静的解析ツールで、噛まれる前にバグを見つける

Elliotte Rusty Harold
Adjunct Professor
Polytechnic University

2005年 1月 07日

オープン・ソースの静的解析ツールであるPMDは、バグ退治のツールとして手元に置くべき価値のあるものです。この記事ではPMDに組み込まれているルールの使い方と、Javaコードの品質を改善するためにカスタムのルール・セットをどのように作るかを、Elliotte Rusty Haroldが説明します。

Tom CopelandによるPMDはオープン・ソース（BSDライセンス）のツールとして、Javaのソースコードを解析し、潜在的なバグを見つけます。PMDはFindBugsやLint4jのような汎用ツール（[参考文献](#)）と似ています。ただし、こうしたツールはそれぞれ異なる種類のバグを見つけるので、どのコード・ベースに対しても、それぞれのツールを走らせた方が得策です。この記事ではPMDの使い方と、PMDでどんなことが期待できるかを説明します。この記事ではコマンドライン・インターフェースを使いますが、PMDはAntと統合することもでき、また主なIDEやプログラム・エディター用のプラグインも存在しています。

PMDのインストールと実行

PMDはJava言語を使って書かれており、JDK 1.3またはそれ以上が必要です。コマンドラインに違和感のない人であれば、PMDのインストールと実行は単純です。zipファイル（[参考文献](#)）をダウンロードし、/usrやホーム・ディレクトリーなど、適当な場所でunzipします。この記事では、/usrにunzipしていると想定しています。

PMDを実行する一番簡単な方法は、pmd.shスクリプト（Unix/Linux）またはpmd.batスクリプト（Windows）を呼び出すことです。普通とは異なり、これらのスクリプトはbinディレクトリーではなく、pmd-2.1/etcディレクトリーにあります。このスクリプトは、次の3つのコマンドライン引き数を取ります。

- チェックすべき .javaファイルへのパス
- 出力フォーマットを示す、html、またはxmlキーワード
- 実行すべきルール・セットの名前

例えば、次のコマンドは名前付けに対するルール・セットを使ってImageGrabber.javaファイルをチェックし、XML出力を生成します。

```
$ /usr/pmd-2.1/etc/pmd.sh ImageGrabber.java xml rulesets/naming.xml
```

結果の解析

上記のコマンドの出力はリスト1のレポートのようになりますが、デフォルトで`System.out`に送られます。

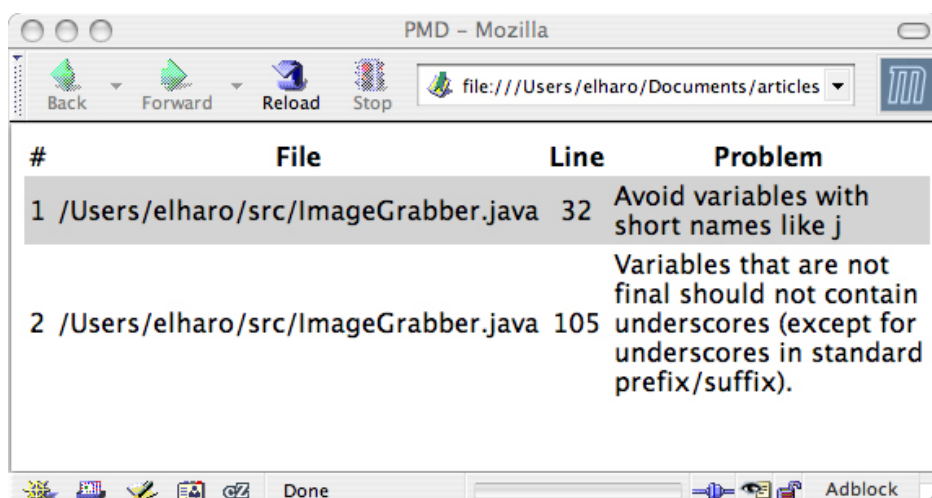
リスト1. PMD XMLレポート

```
<?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>
```

リスト1を見ると、PMDが2つの問題を見つけたことが分かります。ImageGrabber.javaの32行目にある短い変数名と、105行目にある、下線（underscore）を含む名前です。これらは些細な問題に見えるかも知れませんが、驚くような結果を引き起こします。この場合では、105行目の下線は簡単に修正できる、昔からある間違いです。ところが最初の問題を調べてみると、別途増加されている別の変数の機能と重複しているので、`j`変数を完全に削除してしまう可能性があることが分かります。プログラムは動作しましたが、将来何か変更されると、非常に壊れやすくなっていることとなります。一行削減すれば、それだけバグが入り込む余地が少なくなるのです。

PMD出力をファイルにリダイレクトしたり、エディターにパイプしたりすることも普通と同じようにできます。私はよく、出力をHTMLで生成して、Webブラウザーにロードします。これを図1に示します。

図1. HTMLで描画されたPMD出力



#	File	Line	Problem
1	/Users/elharo/src/ImageGrabber.java	32	Avoid variables with short names like j
2	/Users/elharo/src/ImageGrabber.java	105	Variables that are not final should not contain underscores (except for underscores in standard prefix/suffix).

出力をファイルに送ることができる、ソース・ツリーをチェックする時に特に便利です。最初の引き数としてディレクトリー名やzipファイル、またはJARアーカイブ・ファイルを渡すと、PMDはそのディレクトリーまたはアーカイブにある全ての .java ファイルを再帰的にチェックします。ただし、特にPMDが大量のfalse positiveを生成する時には、膨大な出力に少し尻込みしたくなります。例えばXOMコード・ベース ([参考文献](#)) に対してPMDを実行すると、常に「inのような短い名前を持つ変数を避ける」というレポートが出されることになります。私はInputStreamを指す変数の名前として、「in」は全く問題ないと思っています。いずれにせよ、適切なテキスト・エディターで出力を調べれば、最も頻繁に現れるfalse positiveは非常によく似ているので、それを見つけて削除することは簡単にできます。その後で、残りの問題を修正すればよいわけです。

PMDに決定的に欠けているのは、明らかに危険な操作を意図的に行おうとする場合に、ソースコードに「lint comment」を追加してその意図を表すことができないということです。もっともこれはバグではなく、特長なのかも知れません。実際私は、何を本当のfalse positiveとするかに関して何度か自分の考えを変え、結局PMDが正しい、としたことが一度ならずあります。例えば、下記のようなtry-catchブロックが、長年XOMの様々な場所で起きています。

```
try {
    this.data = data.getBytes("UTF8");
}
catch (UnsupportedEncodingException ex) {
    // All VMs support UTF-8
}
```

PMDはこれを空のcatchブロックだとしてフラグを上げます。私は実際に一部のVMではUTF-8エンコードを認識しないことに気がつくまで、これを問題ではないと思っていました (UTF-8エンコードを認識しないとJava準拠ではなくなってしまうはずです)。そこで私はこのブロックを次のように変更しました。そうするとPMDも文句を言うのを止めたのです。

```
try {
    this.data = data.getBytes("UTF8");
}
catch (UnsupportedEncodingException ex) {
    throw new RuntimeException("Broken VM: Does not support UTF-8");
}
```

どんなルールが使えるか

PMDにはJavaコードで一般的な問題を網羅する、16のルール・セットがあります。その内の一部は、やや議論を呼ぶものとなっています。

どんな名前を渡すか

コマンドライン上でどんな名前のルールを渡すべきかについては、あまりよく文書化されていないため、少し試行錯誤が必要な場合があります。この記事でカッコに入っている名前は動作します。

- 基本ルール・セット (rulesets/basic.xml) -- 恐らく大部分の開発者が納得する、多種多様なルールです。catchブロックは空であってはならない、equals()をオーバーライドする時には必ずhashCode()をオーバーライドする、等をチェックします。

- 名前付けに対するルール・セット (rulesets/naming.xml) -- Javaにおける、標準的な名前付けに関するテストです。変数名を短くしすぎない、メソッド名を長くしすぎない、クラス名の先頭は大文字にする、メソッド名やフィールド名の先頭は小文字にする、等のルールをチェックします。
- 未使用コードに対するルール・セット (rulesets/unusedcode.xml) -- 読まれることのないプライベート・フィールドとローカル変数、到達不能ステートメント、呼ばれることのないプライベート・メソッド等を探します。
- 設計に対するルール・セット (rulesets/design.xml) -- 良い設計とするための、様々な原則をチェックします。switchステートメントはdefaultブロックを持つべきである、深くネストしたifブロックは避けるべきである、パラメーターは再割り当てすべきではない、doubleが等しいかどうかの比較はすべきではない、などの原則に合致しているかどうかをチェックします。
- インポート・ステートメントに対するルール・セット (rulesets/imports.xml) -- 同じクラスを2度インポートしたり、java.langからクラスをインポートしたりといった、インポート・ステートメントにおけるマイナーな問題をチェックします。
- JUnitテストに対するルール・セット (rulesets/junit.xml) -- メソッド名の綴りが正しいかどうか、suite()メソッドが静的でパブリックかどうかなど、テスト・ケースやテスト・メソッドに固有の問題を探します。
- スtringに対するルール・セット (rulesets/string.xml) -- 文字列リテラルが重複したり、Stringコンストラクターを呼んだり、Stringオブジェクトに対してtoString()を呼んだり、といった、文字列を扱う時に起きてくる一般的な問題を特定します。
- カッコに対するルール・セット (rulesets/braces.xml) -- forやif、それにwhileやelseなどのステートメントがカッコを使っているかどうかをチェックします。
- コード・サイズに対するルール・セット (rulesets/codesize.xml) -- 異常に長いメソッドや、メソッドが多すぎるクラス、またこれらに類似した、リファクターの元になりそうなものをテストします。
- Javabeansに対するルール・セット (rulesets/javabeans.xml) -- シリアル化できないbeanクラスなど、JavaBeansのコーディング規約に違反していないかどうか、Javabeansコンポーネントを検査します。
- ファイナライザーに対するルール・セット -- Javaではfinalize() メソッドはあまりにも稀にしか使われないので（私も書こうと思ったことはありません）、詳細に説明されているにも関わらず、その使い方のルールは知られていません。このチェック・グループは、空のファイナライザーや他のメソッドを呼ぶfinalize() メソッド、finalize() への明示的なコールなど、finalize() メソッドにまつわる様々な問題を探し出します。
- クローンに対するルール・セット (rulesets/clone.xml) -- clone() メソッドに関する幾つかのルールをテストします。つまりclone() をオーバーライドするクラスはCloneableを実装する必要があり、clone() メソッドはsuper.clone() を呼ぶべきであり、clone() メソッドは（実際に投げなくても）CloneNotSupportedExceptionを投げるとして宣言すべきである、などのルールをテストします。
- カップリングに対するルール・セット (rulesets/coupling.xml) -- クラス間の結合が強すぎることを示す兆候がないかを探します。例えばインポートが多すぎないかとか、スーパータイプやインターフェースで充分なところにサブクラス・タイプを使っていないか、あるいは一つのクラス内にフィールドや変数、戻り型の数が多すぎないかどうか、などを調べます。
- 厳密な例外に対するルール・セット (rulesets/strictexception.xml) -- 例外を、さらに詳しくテストします。メソッドはjava.lang.Exceptionを投げるように宣言すべきではない、とか、例

外はフロー制御に使うべきではない、`Throwable`を捉えるべきではない、などのルールをテストします。

- 論争の余地のあるルール・セット (rulesets/controversial.xml) -- PMDのルールのうち、一部はどんな開発者でも受け入れるものですが、残りの幾つかには、少なくとも議論の余地を残しています。このルール・セットには、変数にヌルを割り当てていないか、メソッドからの戻り点が複数ないか、`sun`パッケージからインポートしていないかなど、議論の余地のあるテストを幾つか含んでいます。
- ロギングに対するルール・セット (rulesets/logging-java.xml) -- おかしな`java.util.logging.Logger`の使い方をしている部分を探します。例えば `final` になっていないとか、`static` になっていないとか、クラス内に一つ以上のロガーがある、などを探します。

コマンドラインで名前の間をカンマで区切ることで、幾つかのルール・セットを同時に使ってチェックを行うことができます。

```
$ /usr/pmd-2.1/etc/pmd.sh ~/Projects/XOM/src html
rulesets/design.xml, rulesets/naming.xml, rulesets/basic.xml
```

独自のルール・セットを作る

ある特定なルール・セットを頻繁に使う場合には、自分で作ったルール・セット・ファイルの中に、そうしたルール・セットを組み込んでしまうことができます。この例をリスト2に示します。このルール・セットでは、基本ルールと名前付けルール、それに設計ルールをインポートしています。

リスト2. 基本ルールと名前付けルール、それに設計ルールをインポートするルール・セット

```
<?xml version="1.0"?>
<ruleset name="customruleset">

  <description>
    Sample ruleset for developerWorks article
  </description>

  <rule ref="rulesets/design.xml"/>
  <rule ref="rulesets/naming.xml"/>
  <rule ref="rulesets/basic.xml"/>

</ruleset>
```

よりきめ細かくしたい場合には、含めたい個々のルールのみを、各ルール・セットから選び出すことができます。例えばリスト3は、PMDに組み込まれている3つのルール・セットから特定な11のルールを選択する、カスタムのルール・セットを示しています。大きなコード・ベースをチェックするには非常に時間がかかるので、例え高速なハードウェアを使っている場合であっても、この方法は特定な問題をより速く見つけるための役に立つはずです。

リスト3. 特定な11のルールをインポートするルール・セット

```
<?xml version="1.0"?>
<ruleset name="specific rules">

  <description>
```

```
Sample ruleset for developerWorks article
</description>

<rule ref="rulesets/design.xml/AvoidReassigningParametersRule"/>
<rule ref=
  "rulesets/design.xml/ConstructorCallsOverridableMethod"/>
<rule ref="rulesets/design.xml/FinalFieldCouldBeStatic"/>
<rule ref="rulesets/design.xml/DefaultLabelNotLastInSwitchStmt"/>
<rule ref="rulesets/naming.xml/LongVariable"/>
<rule ref="rulesets/naming.xml/ShortMethodName"/>
<rule ref="rulesets/naming.xml/VariableNamingConventions"/>
<rule ref="rulesets/naming.xml/MethodNamingConventions"/>
<rule ref="rulesets/naming.xml/ClassNamingConventions"/>
<rule ref="rulesets/basic.xml/EmptyCatchBlock"/>
<rule ref="rulesets/basic.xml/EmptyFinallyBlock"/>

</ruleset>
```

また、ルール・セットにある大部分のルールは含め、自分が同意できないようなルールやfalse positiveを大量に発生しそうなルールは除外してしまうこともできます。例えばXOMでは、テーブルの参照を行う時に、デフォルト・ブロック無しにswitchステートメントを使うことがよくあります。そうした場合には、設計ルールの大部分を保持した上で、設計ルールをインポートするルール要素に対して<exclude name="SwitchStmtsShouldHaveDefault"/>子要素を追加し、欠けているdefaultブロックをチェックする機能をオフすることができます。これをリスト4に示します。

リスト4.switchステートメントはデフォルトを持つべきだとする設計ルールを除外するルール・セット

```
<?xml version="1.0"?>
<ruleset name="dw rules">

  <description>
    Sample ruleset for developerWorks article
  </description>

  <rule ref="rulesets/design.xml">
    <exclude name="SwitchStmtsShouldHaveDefault"/>
  </rule>

</ruleset>
```

(しかし、よく考えてください。PMDが正しいのかも知れません。こうしたルールを考えるよりも、デフォルト・ブロックを追加すべきでしょう。)

PMDに組み込みのルールに制約される必要はありません。Javaコードを書いてPMDを再コンパイルするか、あるいはもっと簡単に、各Javaクラスの抽象構文ツリーに対して解決されるXPath表現を書くことによって、新しいルールを追加することもできるのです。

まとめ

PMDを使えば、組み込みで用意されているルールだけでも、コードに潜んでいる真の問題を確実に幾つか見つけることができます(組み込みで用意されているルールは非常に広範に渡っています)。問題の一部はマイナーなものかも知れませんが、一部は大きな問題かも知れませんが、ただし、PMDで全ての問題を見つけられるわけではありません。ユニット・テストや受け入れテストは相変わらず必要です。またPMDは、既知のバグを追跡するための良質なデバッガーの代用に

はなりません。しかし、自分で気が付かなかったようなバグを見つける場合には、その真価を発揮します。私は、PMDで何の問題も見つからなかったようなコード・ベースは未だかつて見たことがありません。PMDを使えば安価で容易に、そして楽しくプログラムを改善することができます。まだPMDを試したことがなければ、ぜひ試すべきですし、また皆さんの顧客にも試してもらうべきでしょう。

著者について

Elliote Rusty Harold



Elliote Rusty Harold はニューオーリンズ出身であり、時々、おいしい gumbo (オクラ入りのスープ) を食べに帰っています。ただし現在はニューヨークのブルックリン近郊の Prospect Heights に、妻の Beth と猫の Charm (charmed quark からとりました) と Marjorie (義理の母の名前からとりました) と一緒に住んでいます。彼は Polytechnic University のコンピューター・サイエンスの非常勤教授として、Java 技術とオブジェクト指向プログラミングを教えています。彼の [Cafe au Lait](#) Web サイトは、インターネット上で最も人気のある独立系 Java サイトの一つです。また、そこから派生した [Cafe con Leche](#) は、最も人気のある XML サイトの一つです。彼の最近の著作には『[Java I/O, 2nd edition](#)』があります。現在は XML 処理用の [XOM](#) API や [Jaxen](#) XPath エンジン、Jester テスト・カバレッジ・ツールなどに取り組んでいます。

© Copyright IBM Corporation 2005

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)