# PMD:

## Seu código Java à prova de balas

Em projetos de sistemas de software, é comum encontrarmos equipes com diversos desenvolvedores, muitas vezes até trabalhando em regiões geograficamente distintas. Com tantas pessoas "mexendo" no código, como podemos garantir a qualidade do que está sendo produzido? Como garantir a uniformidade e a integridade conceitual[1] das classes, objetos e métodos que compõem um sistema orientado a objetos? Essa discussão se torna ainda mais crítica em empresas que empregam o modelo de "fábrica de software" (o que, diga-se de passagem, é um péssimo nome). Neste modelo, a empresa contratante entrega sua pilha de especificações para uma consultoria (a "fábrica"), que é então incumbida de implementar e entregar o software de acordo. Sem entrar nos méritos desse modelo de trabalho, como podemos garantir um mínimo de qualidade para o código recebido? É bastante difícil... Porém, existem algumas práticas de métodos ágeis e ferramentas que podem ajudar. Entre as práticas, podemos destacar: o pair programming (ou, pelo menos, revisão de código), a propriedade coletiva de código, uso de testes automatizados (preferencialmente TDD) e integração contínua. Essas práticas estimulam a uniformidade e coesão do código por meio de padrões, demonstram a corretude do código (por meio dos testes), além de prover uma suíte de testes de regressão para o sistema.

Juntamente com essas práticas, existem algumas ferramentas que podemos usar para ajudar a melhorar a qualidade numa base de código Java. Entre elas, podemos citar o PMD, o FindBugs e o CheckStyle. Estas são ferramentas de análise estática de código que aplicam diversos tipos de regras de validação e emitem relatórios informando a respeito da violação dessas regras. Todas elas podem ser executadas por scripts Ant e também por plugin do Eclipse. Neste artigo, falaremos sobre o PMD.

### Instalação e execução da ferramenta

Vamos demonstrar o uso do PMD como plugin para o Eclipse (neste artigo, estamos usando o Eclipse Helios). Para instalar o plugin, vá em "Help" -> "Install New Software". Adicione o PMD usando a seguinte URL: http://pmd.sourceforge.net/eclipse. Após o Eclipse carregar os itens, selecione "PMD for Eclipse 3" e clique em "Next". Aceite os termos da licença e conclua a instalação. Ao final, o Eclipse deverá ser reiniciado.

Para usar o PMD, pode-se clicar com o botão direito sobre o projeto (ou sobre um pacote ou classe) e selecionar a opção "PMD" -> "Check code with PMD". Neste caso, o Eclipse abrirá a perspectiva do PMD, onde é possível monitorar todas as violações de regras detectadas no código. A figura 1 exibe a perspectiva do PMD mostrando as violações encontradas no código da Listagem 1 (não tente entender o que o código faz).

Conforme pode ser visto na figura, entre os tipos de violações exibidas para esse código, encontramos:

- Evite variáveis com nomes curtos, como "x";
- System.out.print está sendo usado;
- Usar escopo específico, em vez do pacote default.

[1] *Integridade conceitual é um conceito central do clássico livro "The mythical man-month", de Fred Brooks. O conceito, discutido fundamentalmente no ponto de vista do usuário, está relacionado ao entendimento e facilidade de uso do software. É um desafio manter essa integridade num ambiente com grande divisão de trabalho.*

**Alexandre Gazola**
(alexandregazola@gmail.com; twitter: @alexandregazola): é bacharel em Ciência da Computação pela Universidade Federal de Viçosa (UFV) e mestre em Informática pela PUC-Rio. Trabalha como Analista de Sistemas no BNDES, desenvolve em Java desde 2003 e possui as certificações SCJP e SCWCD. Mantém um blog em http://alexandregazola.wordpress.com.

*Olá, caro leitor! Neste primeiro artigo da coluna Cinto de Utilidades, falaremos um pouco sobre uma ferramenta já bastante conhecida pelos desenvolvedores Java: o PMD. Esta ferramenta funciona por meio da aplicação automatizada de regras de validação sobre bases de código e seu uso pode nos ajudar a garantir mais qualidade em nossos projetos. Apesar de ser uma ferramenta antiga, muitos desenvolvedores iniciantes não a conhecem e muitos desenvolvedores mais experientes não a utilizam regularmente em seus projetos. Neste artigo, faremos um breve apanhado da ferramenta, fornecendo as principais dicas para que o leitor possa aplicá-la imediatamente a seus projetos.*

Listagem 1. Código non-sense para demonstração de algumas violações do PMD.

```
public class MinhaClasse {

   synchronized void teste() {
      int z = 5;
      boolean x = true;

      if (z == 5) {
         System.out.println("OK!");
         return;
      } else if (x) {
         if (z % 3 > 4) {
            if (z == 8 && z > 2) {
               z = 25;
               return;
            } else {
               String teste = "";
               while (z < 20)
                  teste = teste + "teste";
               System.out.println("erro");
            }
            return;
         }
      }

      if (z > 45) {
         while (z < 100) {
            z++;
            if (z == 90) {
               System.out.println("90");
            }
         }
      }
   }
}
```

É interessante manter o PMD continuamente monitorando o código-fonte produzido. Para isso, clique com o botão direito sobre a pasta do projeto, vá em "Properties" -> "PMD" e marque a opção "Enable PMD". É aconselhável desmarcar também a opção "Handle high priority violations as Eclipse errors" para que o PMD não considere como erros violações das regras mais prioritárias (mais adiante falaremos sobre as propriedades das regras), caso contrário podemos indevidamente pensar que existe algum erro de compilação no código.

## Descrição geral

Como já explicado, o PMD realiza uma varredura numa base de código, procurando por possíveis problemas, tais como código duplicado, trechos de código de elevada complexidade, problemas de sincronização de threads etc. Os principais tipos de regras do PMD aparecem resumidos na tabela 1. Ainda existem regras relacionadas ao uso de tecnologias mais específicas, como Java EE, JSP, JSF e outros.

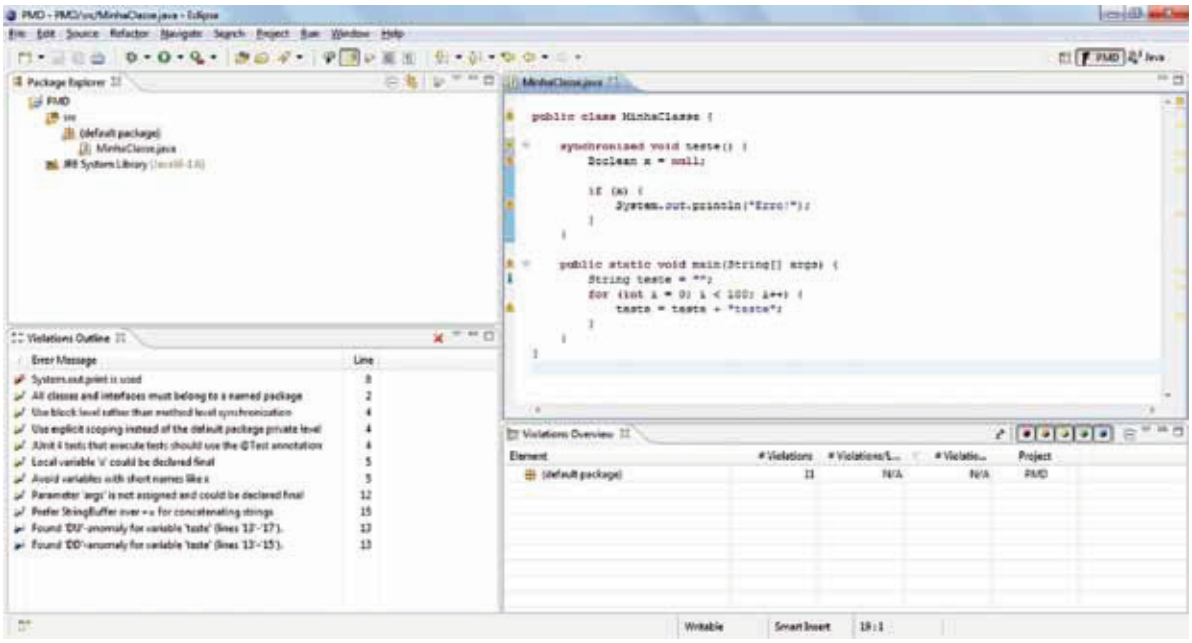

*Figura 1. Perspectiva do PMD mostrando violações de regras.*

| Tipos de regras | Nome no ruleset | Descrição | Exemplos |
|---|---|---|---|
| Basic rules | basic.xml | Regras básicas gerais. | Blocos try ou catch vazios, usar return desnecessariamente, etc. |
| Braces rules | braces.xml | Regras relacionadas ao uso de chaves. | Ifs, whiles e loops devem possuir chaves. |
| Code size rules | codesize.xml | Regras que avaliam questões relacionadas ao tamanho do código. | Detecção de complexidade ciclomática, grande número de métodos públicos ou campos em uma classe etc. |
| Controversial rules | controversial.xml | Regras de aplicação geral, mas de aplicação controversa. | Presença de construtor desnecessário, atribuição de null a uma variável etc. |
| Coupling rules | coupling.xml | Regras relacionadas ao acoplamento entre objetos e pacotes. | Número excessivo de imports, uso de classe concreta em declaração ao invés da interface (ex.: "ArrayList l" em vez de "List l" etc.). |
| Design rules | design.xml | Regras que avaliam o design do código. | Chamada a métodos que podem ser sobrecarregados no construtor, preservar o stacktrace em exceções, colocar o literal primeiro em comparações etc. |
| Import statement rules | imports.xml | Regras relacionadas ao uso de import. | Abuso do recurso de static import. |
| Naming rules | naming.xml | Regras que avaliam nomes de variáveis e métodos. | Nomes de métodos ou variáveis curtos demais ou longos demais, nomes de classes devem começar com maiúscula etc. |
| Optimization rules | optimizations.xml | Regras relacionadas à otimização. | Conversão de argumentos de métodos para final, evitar instanciação de objetos em loops etc. |
| Strict Exception rules | strictexception.xml | Regras relacionadas ao lançamento e captura de exceções. | Captura de Throwable, exceções usadas para controle de fluxo etc. |
| String and StringBuffer rules | strings.xml | Regras que verificam o bom uso das classes String e StringBuffer. | Evitar a duplicação de literais, instanciação de Strings etc. |
| Unused code rules | unusedcode.xml | Regras que detectam código não utilizado. | Campos privados, variáveis locais e parâmetros de métodos não usados. |

*Tabela 1. Alguns dos principais tipos de regras implementados pelo PMD.*

É um pouco enfadonho e ineficaz utilizar o PMD com todas as regras que ele possui. Muitas das regras são desnecessárias, outras podem fazer sentido em alguns contextos, e assim por diante. Devido a esses problemas, o PMD permite que o conjunto de regras aplicadas (os rulesets) possa ser customizado por meio de um arquivo de configuração XML. Isso é extremamente recomendável e simples de fazer. A Listagem 2 traz um exemplo de arquivo de regras do PMD customizado.

Vamos desconsiderar o cabeçalho básico e passar para o mais importante, que é a especificação de quais regras executar. Dizemos qual tipo de regra desejamos por meio da tag <rule ref=”rulesets/xxxxxxx” />, em que “xxxxxxx” é o nome do ruleset do tipo de regra desejada (ex.: “controversial.xml”, “design.xml” etc.), conforme mostrado na coluna “Nome do ruleset” da tabela 1 (outros tipos de regras e suas referências podem ser consultados no site do PMD). Especificando a tag <rule> com o nome do ruleset faz com que todas as regras daquele ruleset sejam utilizadas na validação. Como nem todas as regras são úteis, pode-se excluir as regras não desejadas por meio da tag <exclude name=yyyyyy>, em que ”yyyyyy” é o nome da regra específica do ruleset (os nomes das regras, bem como suas descrições, presentes em cada ruleset podem ser encontrados no site do PMD, na seção Rule Sets). Alternativamente à configuração por exclusão (que é a usada na Listagem 2), podemos criar o arquivo customizado incluindo regra a regra por meio do nome completo na tag <rule>, por exemplo: <rule ref="rulesets/naming.xml/ShortMethodName ">.

Outro conceito presente nas regras do PMD é a sua prioridade. Existem regras de maior prioridade e regras de menor prioridade. A definição da prioridade é feita por meio da aplicação da tag <priority>valor_da_prioridade</priority> como tag filha de <rule>. Quanto menor o valor da prioridade, mais crítica é a regra.

Em meus projetos, costumo deixar todas as regras na prioridade default e, para as regras menos prioritárias, atribuo o valor 5. Isso deixa um efeito interessante no Eclipse com o PMD habilitado: violações das regras de prioridade default são exibidas na aba de problemas como “warnings” (avisos) padrão do Eclipse, ao passo que violações menos prioritárias (aqueles que possuem explicitamente prioridade 5) são exibidas separadamente como “info” (informativo) na aba de problemas do Eclipse. Com isso, podemos definir para uma equipe de desenvolvimento a diretriz de “zero warnings” no projeto, deixando o tratamento das violações “informativas” a cargo do desenvolvedor. Em resumo, podemos ter o seguinte processo de escolha das regras que irão compor o arquivo de configuração:

- Se a regra é essencial e nunca deve ser violada, então ela deve ser incluída no arquivo de configuração (violações irão para as mensagens “warning”). Os desenvolvedores cuidarão para que nunca exista nenhuma mensagem de warning no projeto.
- Se a regra é importante, mas sua aplicação depende do contexto, então ela é colocada com prioridade menor no arquivo de configuração (violações irão para as mensagens “info”). As mensagens deverão ser analisadas caso a caso.
- Se a regra não é importante para o projeto, ela simplesmente é excluída do arquivo de configuração e, logo, não será aplicada na base de código.

Note que no arquivo da Listagem 2, nesse esquema de configuração por exclusão, para reduzir a prioridade de uma regra é necessário primeiramente excluí-la da importação de todas as regras e depois incluí-la individualmente com a prioridade desejada (ver o caso da regra ImmutableField).

Apenas para comentar, é possível suprimir uma mensagem de violação do PMD colocando um comentário //NOPMD. Por exemplo, o PMD possui uma mensagem de violação quando ele avalia que uma conexão com o banco de dados corre o risco de não ser fechada. Neste caso, poderíamos suprimir a mensagem dessa forma:

```
Connection connection = null; //NOPMD a conexão será fechada
….. // restante do código ….
```

Esse recurso pode ser útil para uma equipe que vai começar a implantação da ferramenta numa base de código legado (sem testes automatizados) e, por isso, talvez não tenha a confiança necessária para remover todos os avisos. Dito isso, aconselho o leitor a se esquecer desta possibilidade e disciplinar-se para que a prática seja a eliminação completa das mensagens de warning (se acontecer de este tipo de recurso começar a ser usado com frequência, muito provavelmente será um indício de que alguma regra deva ser eliminada ou ter sua prioridade reduzida).

Listagem 2. Exemplo de arquivo de configuração.

```
<?xml version="1.0"?>
<ruleset xmlns="http://pmd.sf.net/ruleset/1.0.0" name="MUNDOJ_
RULESET" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xs
i:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.
xsd" xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0 http://pmd.sf.net/
ruleset_xml_schema.xsd">
  <description>
    Conjunto de regras de exemplo
  </description>


  <rule ref="rulesets/clone.xml" />

  <rule ref="rulesets/controversial.xml">
    <exclude name="OnlyOneReturn" />
    <exclude name="AtLeastOneConstructor" />
    <exclude name="CallSuperInConstructor" />
    <exclude name="AvoidFinalLocalVariable" />
    <exclude name="AvoidAccessibilityAlteration" />
    <exclude name="AtLeastOneConstructor" />
    <exclude name="AssignmentInOperand" />
    <exclude name="DataflowAnomalyAnalysis" />
    <exclude name="SingularField" />
    <exclude name="UnnecessaryParentheses"/>
    <exclude name="NullAssignment" />
  </rule>

  <rule ref="rulesets/design.xml">
    <exclude name="ImmutableField" />
    <exclude name="AbstractClassWithoutAbstractMethod" />
  </rule>
  <rule ref="rulesets/design.xml/ImmutableField">
    <priority>5</priority>
  </rule>

  <rule ref="rulesets/imports.xml">
    <exclude name="TooManyStaticImports" />
  </rule>
  <rule ref="rulesets/imports.xml/TooManyStaticImports">
    <properties>
      <property name="maximumStaticImports" value="15"/>
    </properties>
  </rule>
</ruleset>
```

Na definição da regra, pode-se passar uma mensagem para a tag rule usando o atributo message, por exemplo:

```
<rule ref="rulesets/optimizations.xml/UseStringBufferForStringAppends"

    message="Prefer StringBuilder over += for concatenating strings">

</rule>
```

Este exemplo de configuração de mensagem em particular é útil, porque a mensagem padrão aconselha o uso de StringBuffer, que é uma classe mais antiga do JDK e é sincronizada. A partir do 5, foi introduzida a classe StringBuilder, que possui o mesmo comportamento, porém não possui o overhead de sincronização, sendo, portanto, a mais recomendada para a maioria dos casos.

Algumas regras possuem propriedades. Esse é o caso, por exemplo, da regra para validar o número máximo de imports estáticos (TooManyStaticImports) permitido em um arquivo Java. Quando o recurso de import estático é abusado, a classe que faz uso do recurso pode ficar de difícil compreensão devido à ausência dos namespaces (fica meio confuso de onde as coisas estão vindo). No exemplo da Listagem 2, estamos limitando o número de imports estáticos a 15. Outras regras também possuem propriedades similares, como as que determinam o número máximo de métodos ou campos permitidos em uma classe.

Aproveitando que entramos um pouco no assunto de regras relacionadas a métricas e tamanho de código, vamos ressaltar a importância dessas regras. Os desenvolvedores, especialmente os mais iniciantes, muitas vezes não se importam com o tamanho de um método ou com o número de métodos que uma classe possui; sendo que estes são fatores importantes que podem indicar "code smells" ("cheiro ruim", ou problemas, no código). O uso de regras que cuidam dessas questões permite que haja uma disciplina no projeto, a qual favorece o uso contínuo de refatoração visando uma melhor separação de responsabilidades, maior coesão e menor acoplamento. Uma regra, que muitas vezes é negligenciada, é a que trata da complexidade ciclomática. Grosso modo, no PMD, a complexidade ciclomática de um método é determinada pelo número de pontos de decisão presentes no método (ifs, whiles, for, switch). O objetivo é minimizar ao máximo essa comple-

xidade. Uma refatoração bastante útil neste caso é o "Compose method", cuja ideia é extrair cada trecho lógico de um método (muitas vezes, cada ponto de decisão) em um método privado da classe, de forma que o método original possa ser lido em alto nível, com todos os passos no mesmo nível de abstração (alguns desenvolvedores trabalham com a premissa que cada método deve possuir, no máximo, um nível de identação).

Bom, uma vez definido o arquivo de configuração, para utilizá-lo, vá Preferences -> PMD -> Rules Configuration. Limpe todas as regras existentes (botão "Clear all"). Em seguida, clique em "Import rule set" e selecione o arquivo de configuração das regras. É importante marcar a opção "Import by copy" (por algum bug do plugin, só funciona nessa opção). Confirmando a inclusão do arquivo, o Eclipse solicitará um rebuild do projeto e aplicará as regras conforme especificado no arquivo (nota: por algum problema do plugin, depois de algum tempo de uso, às vezes acontece de ele voltar a aplicar novamente todas as regras do PMD, como se nenhum arquivo de configuração tivesse sido especificado. Quando isso acontece, é necessário repetir este processo de seleção do arquivo).

Além da utilização do PMD para feedback instantâneo via plugin do Eclipse, é bastante recomendável incluir as suas verificações de forma automática no build do projeto. Uma forma de se fazer isso é usando algumas tasks Ant do PMD (por exemplo: PMDTask). Incluindo o script num servidor de integração contínua, torna-se possível monitorar continuamente o projeto e alertar os desenvolvedores de possíveis problemas e desvios dos padrões estabelecidos.

Com essas informações, já é possível utilizar o PMD de maneira eficaz em projetos de desenvolvimento que fazem uso da linguagem Java. No site da revista, estará disponível um arquivo de configuração que poderá ser usado como base e ajustado conforme as necessidades do projeto. Ainda é possível criar regras customizadas, mas esse assunto está fora do escopo deste texto (o site do PMD possui informações sobre isso).

## Para Saber Mais

"Herança e Composição – os princípios por trás dos padrões", ed. 39 da MundoJ – Eduardo Guerra

"Refatoração: Melhoria Contínua do Código", ed. 33 da MundoJ – Eduardo Guerra

Série "Design patterns para um Mundo Real", eds. 21, 22 e 23 da MundoJ – Rodrigo Yoshima

## Considerações finais

Neste primeiro artigo, falamos sobre uma ferramenta bastante útil para projetos de software que utilizam a linguagem Java: o PMD. A ferramenta realiza varreduras automáticas em bases de código, gerando relatórios de possíveis problemas encontrados. Seu uso permite que possamos garantir mais qualidade para os códigos produzidos. É interessante que cada equipe customize seu próprio arquivo de regras e que todos no projeto se esforcem para que as regras definidas não sejam violadas.

Certamente, não é possível assegurar a qualidade de um código apenas pelo uso de uma ferramenta como o PMD. Por isso, é importantíssimo complementar o uso da ferramenta com testes automatizados, pair programming (revisão de código), integração contínua e todas as boas práticas da nossa indústria.

Cuide bem do seu código! E fique atento às violações!

"Ouve o conselho e recebe a instrução, para que sejas sábio nos teus dias por vir." (Pv 19:20)•

## Referências


- *Site do PMD: http://pmd.sourceforge.net/*
- *Refactoring to patterns – Joshua Kerievsky*
- *Refactoring: Improving the Design of Existing Code*


**GUJ – Discussões sobre o tema do artigo e assuntos relacionados**

**Discuta este artigo com 100 mil outros desenvolvedores em www.guj.com.br/MundoJ**