FERRAMENTAS PARA AVALIAÇÃO DE QUALIDADE DE CÓDIGO EM PROJETOS DE DESENVOLVIMENTO DE SOFTWARE EM JAVA

Wander Antunes Gaspar Valente*

RESUMO

Este artigo visa apresentar ferramentas de *software* livre disponíveis para a avaliação da qualidade de código em projetos de sistemas com a linguagem de programação Java. São discutidos os escopos de detecção de violações de código dessas ferramentas e testados os *plug-ins* para a integração com os ambientes de desenvolvimento mais utilizados.

Palavras-chave: Engenharia de software. Ferramentas de qualidade de código. Linguagem de programação Java.

ABSTRACT

This article aims to provide code-quality free *software* tools available for systems projects with the Java programming language. Code violations detection scope of these tools are discussed and *plug-ins* tested for integration with more used developing environments.

Keywords: *Software* engineering. Code-quality tools. Java programming language.

1 INTRODUÇÃO

Desde os primórdios da computação, os programadores têm se esforçado para escrever código de boa qualidade. Entretanto, uma definição clara do que seja código-fonte de qualidade pode variar bastante e depende de fatores diversos relacionados ao seu desenvolvimento como ser seguro, rápido, enxuto, de fácil manutenção, possível de entender e de ser entendido por outros desenvolvedores. Além disso, a linguagem de programação utilizada é fator importante em discussões sobre o tema. Portanto, para avaliar a qualidade de código em uma linguagem específica torna-se necessário especificar os critérios.

*Professor assistente do CES/JF e Mestrando em Modelagem Computacional (UFJF).

Este artigo tem por objetivo apresentar ferramentas de *software* livre para avaliação da qualidade de código em projetos de desenvolvimento em Java e discutir o escopo de atuação. A ênfase do trabalho concentra-se nas ferramentas **CheckStyle** (BURN, 2008) e **PMD** (COPELAND et al., 2008). A escolha de Java para o estudo em questão justifica-se pela ampla utilização no ambiente acadêmico e pela disponibilidade como *software* de código aberto. É ainda objetivo do presente trabalho testar a integração das ferramentas com os ambientes de desenvolvimento *Eclipse*[1] e *Netbeans*[2].

2 PARÂMETROS PARA MESURAR A QUALIDADE DE CÓDIGO

Entre os aspectos avaliados para mesurar a qualidade de código incluem-se (DARWIN, 2007): formatação consistente e facilidade de entendimento (indentação e espacejamento); regras de nomeação consistentes; ausência de erros de compilação; capacidade adequada e consistente de tratamento de erros de execução; aderência a boas práticas de programação e de projeto; documentação abrangente e de fácil entendimento em todo o código.

Algumas das características apresentadas podem ser consideradas fáceis de implementar, mas não todas. Muitas empresas e organizações possuem padrões documentados que os desenvolvedores devem seguir. Na prática, provou-se ser muito difícil para essas organizações obter sucesso nessa padronização (MYATT, 2008). Para se adequar, os programadores precisam assimilar os padrões adotados e rever, freqüentemente, o código produzido para garantir a conformidade do código aos padrões estabelecidos. Esse processo gasta tempo e é difícil de ser conduzido manualmente.

Uma alternativa mais adequada é o emprego de ferramentas capazes de automatizar o processo de avaliação da qualidade de código. Será ainda mais produtivo se tais ferramentas puderem ser utilizadas diretamente em ambientes de desenvolvimento integrado (IDEs – *Integrated Development Environment*).

3 CHECKSTYLE

CheckStyle (BURN, 2008) é uma ferramenta de qualidade de código, projetada para auxiliar os programadores Java a detectar violações de estilo de codificação.

---

[1]Disponível em: <http://www.eclipse.org>. Acesso em: 08 ago. 2008.
[2]Disponível em: <http://www.netbeans.org>. Acesso em: 08 ago. 2008.

O *software* encontra-se na versão 4.4 disponível sob licença LGPL[3] e pode ser usado como um aplicativo, como parte de um *script Ant*[4] ou como *plug-in* para IDEs como *NetBeans* e *Eclipse*.

Checkstyle contém diversos controles (“*checks*”), um para cada padrão de codificação ou estilo capaz de identificar. O desenvolvedor pode configurar o *software* habilitando os controles que desejar para o código a ser avaliado. A partir da avaliação, é gerado um relatório das violações detectadas. Entretanto, **Checkstyle** não é capaz de fazer a refatoração automática do código.

Entre os padrões de codificação e estilo que podem ser identicados incluem-se:

– *imports* duplicados;

– ausência de comentários *Javadoc*;

– falta de aderência à convenção adotada para nomeação – a verificação baseia-a na avaliação de expressões regulares para identificadores válidos. Por exemplo, a ilustração 1 aponta um *check* para garantir que identificadores comecem com um *m* seguido por uma letra maiúscula e o restante dos caracteres letras e números;

– ausência de arquivos de cabeçalho;

– ausência de espaços em branco à esquerda e à direita de identificadores;

– blocos de código duplicados;

– outras boas práticas de codificação, tais como ordem dos modificadores – em conformidade com as especificações da linguagem –, restrição de operadores lógicos em expressões relacionais, e linhas de código extensas. A versão 4.4 do **Checkstyle** possui mais de 100 controles (*checks*) para verificação da qualidade do código fonte.

```
<module name="LocalVariableName">
  <property name="format" value="^m[A-Z][a-zA-Z0-9]*$"/>
</module>
```

ILUSTRAÇÃO 1- Exemplo de expressão regular para identificadores válidos

3.1 CONTROLE STRICTDUPLICATECODE

Checkstyle pode verificar o código e ajudar a identificar linhas duplicadas, incluindo partes em que o desenvolvedor empregou o mecanismo de copiar e colar

[3]A GNU Lesser General Public License (anteriomente, GNU Library General Public License) é uma licença de *software* livre escrita com o intuito de ser um meio-termo entre a GPL e licenças mais permissivas como a licença BSD e a licença MIT.

[4]Apache Ant é uma ferramenta utilizada para automatizar a construção de *software*. Ant foi escrito em Java e projetada inicialmente para ser utilizada em projetos de *software* da própria linguagem.

para produzir blocos de código semelhantes. Esses blocos podem gerar mais pontos de manutenção, duplicação de bugs e ainda dificultar o entendimento do código.

Esse controle contém uma propriedade **min** que especifica o número de linhas para que um bloco de código seja considerado repetido. No arquivo de configuração, apresentado na ilustração 2, se dois blocos de código com três linhas iguais forem detectados em qualquer parte do código fonte, serão considerados duplicados, mesmo que a indentação esteja diferente. Porém, a duplicidade não será detectada se os blocos de código forem funcionalmente idênticos, mas usarem nomes de variáveis diferentes.

Segundo (BURN, 2008), **CheckStyle** procura manter uma relação de compromisso entre rapidez, baixo uso de memória, baixa ocorrência de alarmes falsos e implementação de recursos avançados de pesquisa. O autor observa, porém, que se encontram disponíveis ferramentas comerciais de detecção de código duplicado capazes de obter resultados superiores.

Uma vez identificado uma duplicação de código, o desenvolvedor deve examiná-lo e determinar se a sua funcionalidade é genérica o suficiente para substituí-lo por um método que possa ser reutilizado quantas vezes for necessário.

3.2 CONTROLE UNUSEDIMPORTS

Esse controle é um dos mais simples e úteis disponíveis no **CheckStyle**. Ele verifica todo o código e identifica comandos import que não estão sendo utilizados ou estão duplicados. Esse controle é importante por uma razão: se classes de uma biblioteca de terceiros são importadas para o projeto, torna-se necessário manter a referência aos *links*. O problema é que esses *imports* continuam sendo referenciados mesmo que as classes da biblioteca não estejam mais sendo utilizadas no código.

O *NetBeans*, por exemplo, pode facilmente detectar *imports* não utilizados em um código fonte aberto a partir do comando *Corrigir importações*. Mas esse processo é difícil de ser colocado em prática em grandes projetos de *software*, compostos de muitos arquivos de código fonte. Nesse caso, **Checkstyle** pode ser bastante útil, ao identificar onde existem *imports* não utilizados em todo o projeto. O desenvolvedor terá apenas que abrir e corrigir os fontes detectados.

Ressalta-se também que *imports* não utilizados devem ser removidos antes de gerar *builds* de aplicações Java. Se as referências a bibliotecas estiverem incorretas ou ausentes ocorrerá erro no *build*, o que pode consumir tempo na correção do problema.

3.3 CONTROLE MULTIPLEVARIABLEDECLARATIONS

Trata-se de outro controle útil do **Checkstyle**, capaz de identificar ocorrências

de múltiplas variáveis declaradas e inicializadas em uma única linha de código. Códigos escritos dessa forma são de difícil leitura e documentação. O arquivo de configuração do **Checkstyle** apresentado na ilustração 2 inclui um controle **MultipleVariableDeclarations**.

3.4 ARQUIVO DE CONFIGURAÇÃO DO CHECKSTYLE

Todos os controles do **Checkstyle** são declarados em um arquivo XML (*eXtensible Markup Language*), em que uma *tag module* é definida para cada *check* habilitado (ILUSTRAÇÃO 2). Esse arquivo de configuração pode ser executado a partir de uma linha de comando, como parte de um *script Ant* ou a partir de um *plug-in* para os principais IDEs disponíveis, como *Eclipse* e *Netbeans*.

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
  "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
  "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<module name="Checker">
  <module name="StrictDuplicateCode">
    <property name="min" value="3"/>
  </module>
  <module name="TreeWalker">
    <module name="LocalVariableName">
      <property name="format" value="^m[A-Z][a-zA-Z0-9]*$"/>
    </module>
    <module name="UnusedImports" />
    <module name="MultipleVariableDeclarations"/>
  </module>
</module>
```

ILUSTRAÇÃO 2 - Exemplo de arquivo XML de configuração do Checkstyle

O módulo **Checker** é o topo da hierarquia de *checks* e encapsula todos os demais controles do **Checkstyle**. **TreeWalker** é o pai de todos os controles que avaliam classes individualmente. Assim, todos os *checks*, nesse exemplo, estão em um nível inferior a **TreeWalker**, exceto **Strict DuplicateCode**. Isso significa que o controle de código duplicado verifica todos os fontes incluídos no projeto e é capaz de fazer a detecção mesmo que a duplicidade ocorra em arquivos diferentes. Cabe ao desenvolvedor determinar se a funcionalidade é genérica o suficiente para substituir o trecho de código duplicado por um método que possa ser reutilizado.

O arquivo de configuração exemplificado pode ser executado via linha de

comando, através de um *build Ant* ou a partir de um *plug-in* para IDEs. A ilustração 3 apresenta uma linha de comando para execução do **Checkstyle**, utilizando-se o arquivo de configuração **sun_checks.xml**, conforme mostrado na ilustração 2.

```
java com.puppycrawl.tools.checkstyle.Main –c c:\checkstyle\sun_checks.xml
-r c:\checkstyle\src -o c:\checkstyle\error.txt
```

ILUSTRAÇÃO 3 - Exemplo de execução do Checkstyle a partir da linha de comando

Nesse exemplo, o parâmetro **–r** especifica que todos os fontes Java do projeto contidos na pasta **...\src** serão verificados a partir do arquivo de configuração sun_checks.xml especificado. O parâmetro **–o** especifica que as violações de código detectadas serão listadas no arquivo de texto **error.txt**.

Para efeito de teste, consideremos o arquivo de código fonte Java **Teste.java**, presente na pasta especificada na linha de execução apresentada na ilustração 3.

```
import java.util.Arrays;
import java.util.Scanner;
public class Teste {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String[] matr = new String[20];
        char[][] resp = new char[20][10];
        int n, i, t, c, j;
        n = in.nextInt();
        String gabarito = in.next();
        char[] g = gabarito.toCharArray();
        c = in.nextInt();
        for(i = 0; i < c; i++) {
            matr[i] = in.next();
            String aux = in.next();
            resp[i] = aux.toCharArray();
        }
        System.out.println(“Candidatos:”);
        for(i = 0; i < c; i++) {
            System.out.print(matr[i]);
            t = 0;
            for(j = 0; j < n; j++) {
                if(g[j]==resp[i][j])
                    t++;
            }
            System.out.println(“ “ + t);
        }
```

```
            System.out.println("Questoes:");
            for(i = 0; i < n; i++) {
                System.out.print(i+1);
                t = 0;
                for(j = 0; j < c; j++) {
                    if(g[i] == resp[j][i])
                        t++;
                }
                System.out.println(" " + t);
            }
        }
    }
```

ILUSTRAÇÃO 4 - Código fonte Java Teste.java

A execução do **Checkstyle** para o código fonte da ilustração 4 irá gerar o arquivo **error.txt**, contendo as violações de código encontradas (ILUSTRAÇÃO 5).

```
Starting audit...
c:\checkstyle\src\Teste.java:24: Found duplicate of 4 lines in c:\checkstyle\src\Teste.java, starting from line 34
c:\checkstyle\src\Teste.java:1:8: Importação não utilizada - java.util.Arrays.
c:\checkstyle\src\Teste.java:5:17: Nome 'in' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:6:18: Nome 'matr' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:7:18: Nome 'resp' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:8:9: Each variable declaration must be in its own statement.
c:\checkstyle\src\Teste.java:8:13: Nome 'n' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:8:16: Nome 'i' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:8:19: Nome 't' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:8:22: Nome 'c' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:8:25: Nome 'j' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:10:16: Nome 'gabarito' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:11:16: Nome 'g' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
c:\checkstyle\src\Teste.java:15:20: Nome 'aux' deve condizer com o padrão '^m[A-Z][a-zA-Z0-9]*$'.
Audit done.
```

ILUSTRAÇÃO 5 - Arquivo error.txt contendo violações de código detectadas

3.5 *PLUGIN* DO **CHECKSTYLE** PARA O *ECLIPSE*

O processo de automação da verificação de qualidade de código torna-se mais produtivo quando incorporado a um ambiente de desenvolvimento de *software*. Existem diversos *plug-ins* para integrar os recursos do **Checkstyle** a IDEs amplamente utilizados pela comunidade de engenharia de *software*, como o *Eclipse* (DAVID, 2005).

Um dos trabalhos mais sólidos nesse sentido denomina-se **Eclipse Checkstyle**

(SCHENEIDER e KÖDDERITZSCH, 2008). Desenvolvido por Schneider e Ködderitzsch, encontra-se atualmente na versão 4.4.2. O *software* recebeu o prêmio *Eclipse Community Award 2007* na categoria *Best Open Source Eclipse-based Developer tool*. O *download* da versão corrente e a instalação do *plug-in* podem ser feitos diretamente no **Eclipse** a partir de *Help -> Software Updates -> Find and Install*... Na janela seguinte, *Search for new features to install -> New remote site*. No passo seguinte, cria-se um nome para o *link*, por exemplo, *Checkstyle plug-in*, e informa-se o site do recurso: http://eclipse-cs.sourceforge. net/update.

Uma vez instalado corretamente o *plug-in*, o arquivo de configuração do **Checkstyle**, conforme apresentado na ilustração 6, pode ser incorporado ao *Eclipse* a partir de *Window -> Preferences -> Checkstyle*.

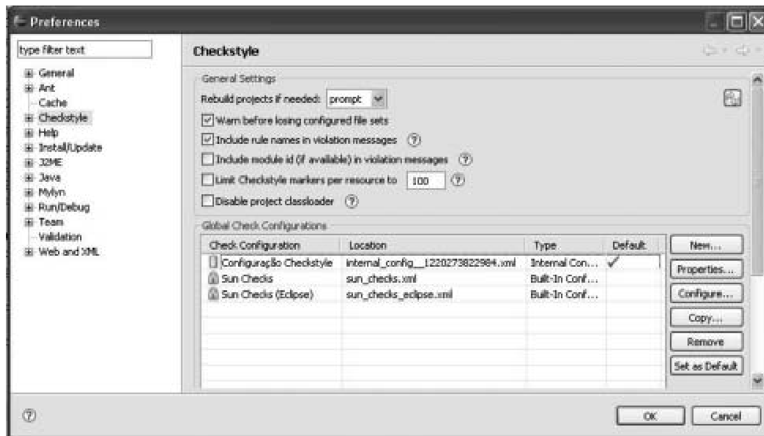

ILUSTRAÇÃO 6 - Janela de configuração do Checkstyle no IDE Eclipse

Em *Global Check Configurations* são listadas as configurações de controle disponíveis. Uma nova configuração, chamada Configuração **Checkstyle**, foi acrescentada, referente ao arquivo **sun_checks.xml**, definido anteriormente e habilitado como configuração padrão. Observe que a instalação do *plug-in* do **Checkstyle** já inclui duas configurações, **Sun Checks** e **Sun Checks (Eclipse)**. A primeira verifica a aderência às convenções de codificação Java sugeridos pela Sun e a segunda é uma adaptação para o padrão de codificação do *IDE Eclipse*.

As mensagens de verificação do **Checkstyle** são apresentadas na janela *Problems* (ILUSTRAÇÃO 7). As opções de visualização podem ser controladas na opção **Checkstyle**, clicando-se com o botão direito do mouse sobre o projeto na aba *Package Explorer* ou sobre o fonte na janela de edição.

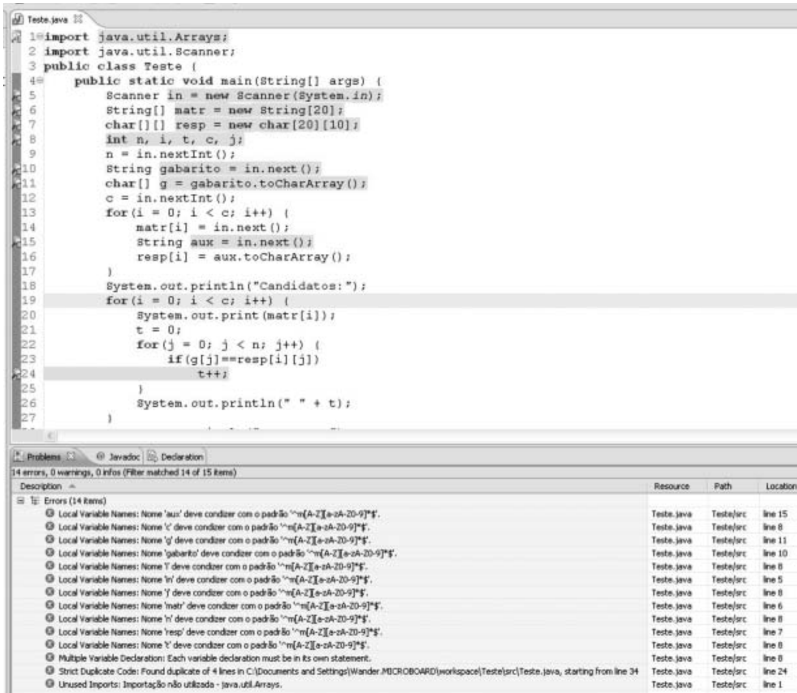

ILUSTRAÇÃO 7 - Janela de execução do *Checkstyle* no *IDE Eclipse*

Outra alternativa de integração do **Checkstyle** ao *IDE Eclipse* é o *plug-in* **Checkclipse** (MEEGEN, 2008), atualmente na versão 2.01. A integração do **Checkstyle** com o *Netbeans* também pode ser obtida a partir de algumas opções de *plug-ins* disponíveis: **Checkstyle Beans** (JAMES, 2008) encontra-se na versão 1.2.0; **NbCheckstyle** (GOULBOURN, 2008), cuja versão mais recente (4.0) mantém compatibilidade com **Netbeans 4.1**; e **Checkstyle Task List Integration** (SAUBRECHET, 2008), que se encontra na versão 1.0 e mantém compatibilidade com **Netbeans** 6.0 M10 ou superior.

4 PMD

PMD (COPELAND et al., 2008) é uma ferramenta de análise estática projetada para a identificação de violações de código e bugs. Enquanto **Checkstyle** é mais focado em estilos e padrões de codificação, **PMD** está mais relacionado a bugs e àotimização do código fonte, além de uma série de outros problemas detectáveis em programação.

O desenvolvimento da ferramenta **PMD** teve início a partir dos trabalhos de Dixon-Peugh e Copeland e se encontra atualmente na versão 4.2.3, disponível para uso sob licença *BSD-style*. Segundo os autores, **PMD** não se trata de nenhum acrônimo e, portanto, não há um nome por extenso aplicável ao *software* (COPELAND, 2005).

PMD pode ser usado a partir de linha de comando do sistema operacional, como um *build Ant* ou integrado aos principais IDES disponíveis, como *Eclipse* e *Netbeans* e mesmo de ambientes voltados ao ambiente acadêmico, como o *BlueJ*.

4.1 VISÃO GERAL DO PMD

Assim como **Checkstyle**, **PMD** também possui uma série de controles, cada um relacionado a um tópico específico que a ferramenta é capaz de avaliar. É possível configurar o **PMD** usando um ou mais controles e avaliar a partir daí o código fonte. **PMD** irá gerar um relatório contendo as violações detectadas. O desenvolvedor Java poderá então reavaliar o código e fazer os aprimoramentos que julgar pertinentes.

Entre os principais problemas de codificação avaliados pelo **PMD** destacam-se (COPELAND et al., 2008):

– possíveis *bugs*, por exemplo, *try/catch* vazios ou *switch* sem *breaks*;

– *dead code*, por exemplo, variáveis locais ou métodos não referenciados;

– *suboptimal code*, por exemplo, no uso inadequado de classes *String* e *StringBuffer*;

– código ou expressões complexas, por exemplo, em *if´s* desnecessários, laços *for* em lugar de laços *while* mais simples;

– código duplicado.

A versão 4.2.3 do **PMD** possui centenas de controles (*checks*) para verificação da qualidade do código fonte, classificados em diversas categorias, como básicos, *design, strings* e *Junit tests*. Todos os controles do **PMD** devem ser declarados em um arquivo xml, em que uma *tag rule ref* é definida para cada *check* habilitado (ILUSTRAÇÃO 8). Esse arquivo de configuração pode ser executado a partir de uma linha de comando, como parte de um *script Ant* ou a partir de um *plug-in* para os principais IDEs disponíveis, como *Eclipse* e *Netbeans*.

```
<?xml version="1.0"?>
<ruleset name="cesjf"
   xmlns="http://pmd.sf.net/ruleset/1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0
      http://pmd.sf.net/ruleset_xml_schema.xsd"
   xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_
```

```
    schema.xsd">
  <rule ref="rulesets/design.xml/MissingBreakInSwitch"/>
 <rule ref="rulesets/optimizations.xml/UseStringBufferForStringAppends"/>
</ruleset>
```

ILUSTRAÇÃO 8 - Arquivo xml de configuração do PMD

4.2 CONTROLE MISSINGBREAKINSWITCH

PMD pode identificar comandos *switch* sem cláusulas *break*. O arquivo de configuração apresentado na ilustração 8 inclui um controle **MissingBreakInSwitch**. O exemplo de código fonte da ilustração 9 contém um comando *switch* sem as cláusulas *break* correspontentes a cada case e o controle específico incluído no arquivo cesjf. xml permitirá ao **PMD** detectar a violação de código.

```
import java.util.Scanner;
class Teste2 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int val = in.nextInt();
      String string = teste(val);
      System.out.println(string);
   }
   public static String teste(int i) {
      String string = "";
      switch (i) {
         case 0:  string = inteiros(2);
         case 1:  string = inteiros(3);
         default: string = inteiros(1);
      }
      return string;
   }
   public static String inteiros(int num) {
      String valor = "";
      for (int i = 0; i < 100; i += num)
         valor += i;
      return valor;
   }
}
```

ILUSTRAÇÃO 9 - Código fonte Java Teste2.java

4.3 CONTROLE USESTRINGBUFFERFORSTRINGAPPENDS

Esse controle do **PMD** é capaz de identificar inserções inadequadas em objetos da classe *String*. Muitas vezes os programadores escrevem trechos de programas com concatenações repetidas em *strings*, principalmente em comandos de laço, o que gera um código ineficiente, sujeito à degradação na performance e inadequação no uso da memória. A concatenação a partir de *StringBuffer* é mais eficiente porque os objetos dessa classe não são imutáveis, ao contrário dos objetos da classe *String*. O problema reside no fato de que, a cada nova concatenação com *strings*, a JVM gera um novo objeto na memória para executar a operação (HUTCHERSON, 2000).

O método "inteiros" presente no código fonte (ILUSTRAÇÃO 9) concatena diversos números inteiros em um objeto *String*. O arquivo xml de configuração **PMD** (ILUSTRAÇÃO 8) inclui um controle para a identificação de uso inadequado da classe *String* em comandos aritméticos de atribuição +=.

4.4 ARQUIVO DE CONFIGURAÇÃO DO PMD

Após a determinação pela equipe de desenvolvimento de todos os controles **PMD** a serem utilizados no projeto, será criado um arquivo xml de configuração conforme apresentado na ilustração 8.

O objetivo do trabalho de verificação do código fonte com a ferramenta **PMD** a partir do arquivo de configuração produzido é compelir a equipe de desenvolvimento a seguir boas práticas de codificação em Java definidas para o projeto (DOEDERLEIN, 2006).

O arquivo de configuração pode ser executado via linha de comando, através de um *build Ant* ou a partir de um *plug-in* para algum IDE. A ilustração 10 apresenta uma linha de comando para execução do **PMD**, utilizando-se a configuração cesjf.xml, conforme mostrado na ilustração 8.

```
pmd c:\pmd\src\Teste2.java text c:\pmd\xml\cesjf.xml
```

ILUSTRAÇÃO 10 - Exemplo de execução do PMD a partir da linha de comando

A linha de comando acima informa o fonte, o tipo de saída do relatório de verificação e o arquivo de configuração utilizado. A ilustração 11 apresenta o relatório de verificação do **PMD**.

```
c:\pmd\src\Teste2.java:14  A switch statement does not contain a break
c:\pmd\src\Teste2.java:28  Prefer StringBuffer over += for concatenating strings
```

ILUSTRAÇÃO 11 - Exemplo de execução do PMD a partir da linha de comando

4.5 INTEGRAÇÃO DO *PMD* COM O *NETBEANS*

Existem atualmente *plug-ins* disponíveis para integração do **PMD** com diversos IDEs utilizados no desenvolvimento de *software* em Java. O *plug-in* para *Netbeans*, desenvolvido por Kubacki e colaboradores (COPELAND et al., 2008), encontra-se na versão 2.2.1, compatível com **PMD** versão 4.2.1.

O *plug-in*, uma vez obtido a partir do repositório de projetos *open source* **SourceForge.net**, deve ser descompactado e instalado no *Netbeans*, a partir de *Tools -> Plug-ins*. A seguir, em *Downloaded*, clicar em *Add plug-ins* e selecionar o arquivo *pmd.nbm* extraído da descompactação. A instalação deve ser confirmada e a licença de uso aceita.

Após a instalação, é necessário configurar os controles do **PMB** que serão habilitados no ambiente de desenvolvimento (ILUSTRAÇÃO 12). Isso é feito a partir de *Tools -> Options -> Miscellaneous*.

ILUSTRAÇÃO 12 - Janela de configuração do PMD no IDE Netbeans

A opção *Enable scan* – desabilitada por default – deve ser habilitada para a verificação automática do código fonte Java (ILUSTRAÇÃO 12). Qualquer violação das regras estabelecidas no arquivo de configuração **PMD** é listada na área de marcadores (glyphs) da interface do *Netbeans*. Se o desenvolvedor optar por manter desabilitada

essa opção, será necessário executar o **PMB** manualmente sempre que desejado.

A janela *Rule editor*, selecionada a partir de *Manage rules*, permite configurar os controles **PMD** que deverão ser habitados. Para cada controle é apresentada breve descrição do objetivo e um exemplo de aplicação (ILUSTRAÇÃO 13). As regras selecionadas neste exemplo correspondem aos mesmos controles definidos manualmente (ILUSTRAÇÃO 8) através do arquivo xml do **PMD**. A configuração torna-se mais simples e intuitiva, utilizando-se o *plug-in* para o **Netbeans**.

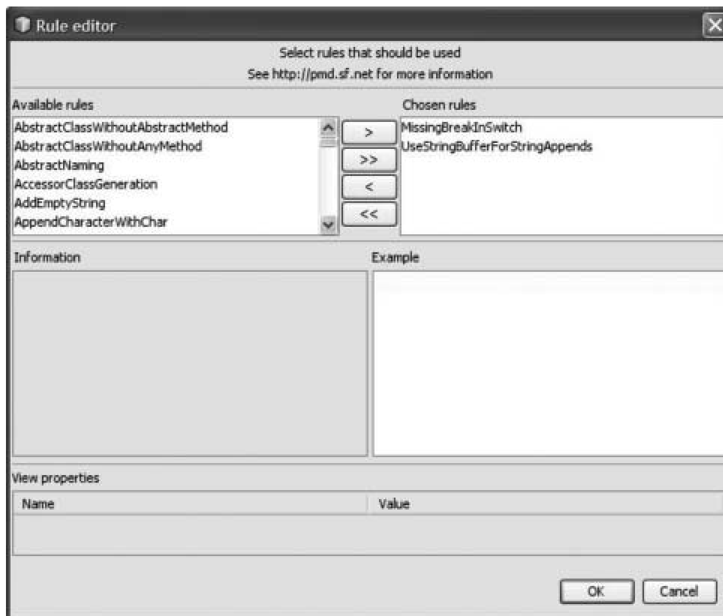

ILUSTRAÇÃO 13 - Janela Rule editor do PMD integrado ao Netbeans

As opções de configuração do **PMD** integrado ao Netbeans também permitem ao desenvolvedor selecionar um arquivo xml de controles previamente escrito. Isso pode ser feito a partir da janela *Rulesets editor*, selecionada em *Tools -> Options -> Miscellaneous*, aba **PMD**, botão *Manage rulesets*. Assim, um arquivo de configuração, como o da ilustração 8, também pode ser definido como o conjunto padrão de controles para o **PMD**.

Com a opção *Enable scan* habilitada, as violações detectadas no código pelo **PMD** são indicadas através de marcadores na margem esquerda da janela de edição do **Netbeans** (ILUSTRAÇÃO 14).

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int val = in.nextInt();
        String string = teste(val);
        System.out.println(string);
    }
    public static String teste(int i) {
        String string = "";
        switch (i) {
            case 0:  string = inteiros(2);
            case 1:  string = inteiros(3);
            default: string = inteiros(1);
        }
        return string;
    }
    public static String inteiros(int num) {
        String valor = "";
        for (int i = 0; i < 100; i += num)
            valor += i;
        return valor;
```

ILUSTRAÇÃO 14 - Marcadores PMD no Netbeans

É possível exibir uma janela **PMD** output contendo a lista de violações detectadas nos arquivos de código fonte de um projeto. Essa opção é selecionada em T*ools* -> *Run PMD* (ILUSTRAÇÃO 15).

5 OUTRAS FERRAMENTAS DE QUALIDADE DE CÓDIGO

Existem ainda outras opções disponíveis de *softwares* livres para a análise estática da qualidade de código-fonte em Java. Destaca-se o **FindBugs** (PUGH et al., 2008), projeto desenvolvido pela Universidade de Maryland, EUA, e distribuído sob licença LGPL, cuja versão corrente é a 1.3.5.

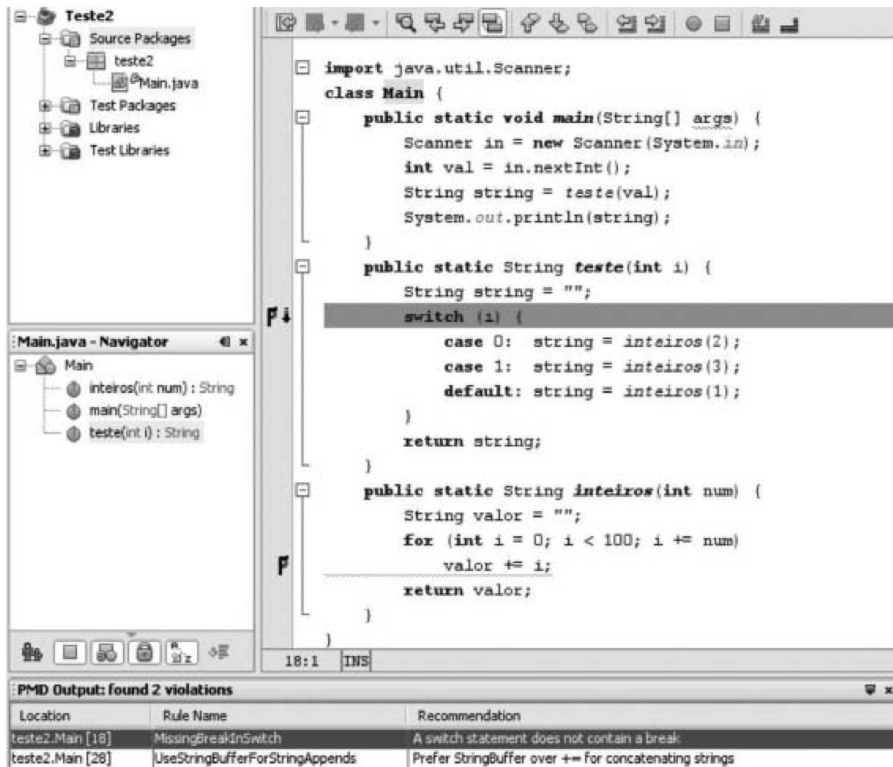

ILUSTRAÇÃO 15 - Janela PMD Output com violações detectadas no projeto

FindBugs pode ser usado como uma aplicação Java, seja a partir da linha de comando do sistema operacional ou de uma interface gráfica baseada em componentes *Swing*, como parte de um *script Ant* ou integrado a ambientes de desenvolvimento como *Netbeans* e *Eclipse* (PUGH et al., 2008).

Software Quality Environment (SQE) (REIMERS, 2009) é um projeto colaborativo associado ao portal de tecnologia *Kenai* patrocinado pela *Sun Microsystems*. O objetivo do projeto é prover a integração unificada ao *Netbeans* de diversas ferramentas de qualidade de código, incluindo **Checkstyle, PMD** e **FindBugs** além de outros *softwares* livres de análise estática de código Java. A versão corrente é compatível com *Netbeans* 6.1. Com o **SQE** integrado ao **Netbeans** é possível executar cada uma das ferramentas individualmente ou em grupo e analisar o resultado da avaliação do código.

6 CONCLUSÃO

As equipes de desenvolvimento em Java dispõem de diversas ferramentas livres destinadas à melhoria da qualidade de código em seus projetos de *software*. Esses utilitários são capazes de verificar o código-fonte produzido e detectar as violações encontradas, incluindo-se códigos duplicados, *bugs*, aderência a boas práticas de programação e projeto, considerações sobre desempenho, entre outros. Esses recursos trazem benefícios concretos ao processo de desenvolvimento. Fazer manualmente a avaliação de qualidade do código fonte é um trabalho difícil, sujeito a erros e omissões, além de consumir um tempo precioso.

A integração com ambientes de desenvolvimento encoraja e agiliza a adoção sistematizada das ferramentas de qualidade de código. Os ganhos no desenvolvimento de *software* podem ser observados em códigos mais seguros e mais rápidos, coesos, de fácil manutenção, entre outros aspectos relevantes.

Artigo recebido em: 05/09/2008
Aceito para publicação: 02/10/2008

REFERÊNCIAS

BURN, O. **Checkstyle 4.4**. Disponível em <http://checkstyle.sourceforge.net/>. Acesso em: 09 ago. 2008.

COPELAND, T. **PMD applied.** Alexandria: Centennial Books, 2005.

COPELAND, T.; DIXON-PEUGH, D.; GRIFFA, M.; CAPLAN, A. **PMD**. Disponível em: <http://pmd.sourceforge.net/>. Acesso em: 02 set. 2008.

DARWIN, I. **Checking Java Programs.** [S.l.]: O´Reilly, 2007.

DAVID, C. **Eclipse Distilled.** Upper Saddle River: Pearson Education. 2005.

DOEDERLEIN, O. **Writing Quality Code with Netbeans.** Netbeans Magazine. Issue One. May, 2006.

GOULBOURN, P. **NbCheckstyle.** Disponível em <http://nbcheckstyle.sourceforge.net/>. Acesso em: 05 ago. 2008.

HUTCHERSON, R. StringBuffer versus String: what is the performance impact of the String Buffer and String classes? **JavaWorld.com.** 24 mar. 2000. Disponível em: <http://www.javaworld.com/ javaworld /jw-03-2000/jw-0324-javaperf.html>. Acesso em: 01 set. 2008.

JAMES, M. **Checkstyle Beans.** Disponível em: <http://www.sickboy.cz/checkstyle/>. Acesso em: 09 ago. 2008.

MEEGEN, M. **Checkclipse.** Disponível em: <http://www.mvmsoft.de/content/plugins/ checkclipse/ checkclipse.htm>. Acesso em: 09 ago. 2008.

MYATT, A. **Pro NetBeans IDE 6 Rich Client Platform Edition.** Berkeley: Apress. 2008.

PUGH, B.; HOVEMEYER, D.; LANGMEAD, B. University of Maryland. **FindBugs.** Disponível em: <http://findbugs. sourceforge.net/>. Acesso em: 05 ago. 2008.

SAUBRECHET. **Checklist-Task List Integration.** Disponível em: <http://plugins.netbeans.org>. Acesso em: 08 ago. 2008.

SCHNEIDER, D.; KÖDDERITZSCH, L. **Eclipse Checkstyle plugin.** Disponível em: <http:// eclipse-cs.sourceforge. net/>. Acesso em: 09 ago. 2008.

REIMERS, S. **Software Quality Environment.** Disponível em: < http://kenai.com/projects/sqe/>. Acesso em: 20 out. 2009.