03

Incluindo plugins

Transcrição

Conhecemos os ciclos de vida do *build* de um projeto, suas respectivas fases e, em algum destes momentos, podemos gerar relatórios variados, e não apenas os de testes.

Um deles, que pode ser gerado pelo Maven, é o PMD (https://maven.apache.org/plugins/maven-pmd-plugin/), que analisa o código fonte e detecta possíveis margens de *bug* no código. Para gerar este relatório utilizamos o comando `pmd:pmd` no terminal:

```
mvn pmd:pmd
```

Como o relatório PMD segue o padrão do Maven, o `.jar` desse plugin é encontrado automaticamente, ou seja, não precisamos efetivar nenhum tipo de configuração, já que o download dos conteúdos é realizado sem problemas. Ao final, será gerado um arquivo `pmd.html`, armazenado no diretório "produtos > target > site". Vejamos o conteúdo do relatório:

```
Resultados do PMD

O seguinte documento contém os resultados do PMD 5.3.5.

Arquivos

br/com/alura/maven/Produto.java

     Violação                                Linha
Avoid unused private fields such as 'nome'.5
Avoid unused private fields such as 'preco'.6
```

O relatório nos aconselha a não usar campos como `nome` e `preco`, presentes na classe `Produto`, afinal os valores desses campos não são usados. Se temos uma variável local ou membro que não é usada, isso pode ser interpretado como uma margem de *bug*. A boa prática de programação recomenda que **as variáveis devem ser definidas quando elas de fato são necessárias no escopo**.

Em nosso caso, as variáveis `preco` e `nome` seriam utilizadas por meio de *getters*. Na classe `App` formataremos o código para o padrão Java e criaremos um produto chamado `bala juquinha sabor tangerina`, que custa `0.15` centavos.

```
package br.com.alura.maven;

/**
*Hello world!
*
*/
public class App {
    public static void main(String[] args) {

        new Produto("Bala juquinha sabor tangerina", 0.15);
```

```
        System.out.println("Hello World!");
    }
}
```

Não precisaremos de uma variável `produto`, portanto não iremos criá-la. Criar uma variável inútil implica em **prejudicar o desempenho do CPU**, ou gerar um efeito colateral em tempo de execução. Criaremos a variável `produto` apenas para verificar o relatório PMD. E no terminal, utilizaremos o comando `pmd:pmd`, e teremos o seguinte relatório:

```
Resultados do PMD

O seguinte documento contém os resultados do PMD 5.3.5.

Arquivos


br/com/alura/maven/App.java

     Violação                                Linha
Avoid unused local variables such as 'produto'.10


br/com/alura/maven/Produto.java

     Violação                                Linha
Avoid unused private fields such as 'nome'.5
Avoid unused private fields such as 'preco'.6
```

Podemos configurar o PMD para diferentes fins, então analisemos outras possibilidades: em `App` imprimiremos a frase `A bala que eu gosto é:` seguida do nome da bala, isto é, `produto.getNome()`.

```
package br.com.alura.maven;

/**
*Hello world!
*
*/
public class App {
    public static void main(String[] args) {

        Produto produto = new Produto("Bala juquinha sabor tangerina", 0.15);

        System.out.println("A bala que eu gosto é: "+ produto.getNome());
    }
}
```

Incluiremos o método `getNome()` na classe `Produto`:

```
package br.com.alura.maven;

public class Produto {
    private final String nome;
    private final double preco;
```

```
    public Produto(String nome, double preco) {
        super();
        this.nome = nome;
        this.preco = preco;
    }

    public String getNome() {
        return nome;
    }
}
```

No terminal, geraremos um novo relatório PMD. Em seu conteúdo, veremos que os avisos relativos a `nome` e `produto` sumiram:

```
Resultados do PMD

O seguinte documento contém os resultados do PMD 5.3.5.

Arquivos

br/com/alura/maven/Produto.java

     Violação                                Linha
Avoid unused private fields such as 'preco'.6
```

Na documentação do Maven relativa ao relatório PDM (https://maven.apache.org/plugins/maven-pmd-plugin/pmd-mojo.html) encontraremos as configurações possíveis e exemplos de uso, tais como análise de código JavaScript, Java Server Pages (JSP), como usar *Rule Sets*, ou regras determinadas. Esse recurso pode ser usado de diferentes maneiras para detectar possíveis pontos problemáticos no programa em desenvolvimento.

Com o comando `pmd:pmd` conseguimos gerar relatórios, mas de que forma verificamos a qualidade do nosso projeto? Lembrando que a verificação é uma fase do ciclo de vida do *build*.

Se simplesmente utilizarmos o comando `mvn verify` no terminal, não teremos uma verificação efetiva, afinal não configuramos o PMD para ser utilizado no momento da verificação. Para isso, utilizaremos o comando `pmd:check`, que realiza uma varredura no *build* à procura de erros, inclusive interrompendo o projeto caso as regras definidas para o código não sejam cumpridas.

Ao executarmos `pmd:check`, receberemos uma mensagem de erro no terminal (`BUILD FAILURE`), e a alegação é de que temos uma violação (`You have 1 PMD violation`). Estamos utilizando o PMD padrão, que possui diversas regras. Para customizá-lo precisamos ir ao *Rules Set* e aplicar nossas preferências.

No PMD padrão não são permitidas variáveis inutilizadas, como é o caso de `preco`. Sendo assim, atribuiremos um uso a esta variável na classe `App`, por meio de `produto.getPreco()`.

```
public class App {
    public static void main(String[] args) {

        Produto produto = new Produto("Bala juquinha sabor tangerina", 0.15);

        System.out.println("A bala que eu gosto é: "+ produto.getNome() + "e custa " + produto.g
```

```
    }
}
```

Escreveremos o método na classe `Produto`:

```
public class Produto {

    private final String nome;
    private final double preco;

    public Produto(String nome, double preco) {
        super();
        this.nome = nome;
        this.preco = preco;
    }

    public String getNome() {
        return nome;
    }

    public double getPreco() {
        return preco;
    }
}
```

Feitas as modificações, executaremos novamente o comando `pmd:check`, e teremos o resultado positivo, ou seja, corrigimos os erros apontados pelo PMD em nosso programa. Tal comando corresponde à fase de verificação do projeto de acordo com sua documentação. Primeiramente é executado o `pmd` e, depois, o `pmd:check`.

Temos uma questão: todas as vezes em que quisermos executar o PMD precisamos utilizar `pmd:check` no terminal, o que pode se tornar cansativo ao longo do desenvolvimento do programa. No arquivo `pom.xml`, podemos realizar configurações que permitam a execução automática do PDM durante o *build* do projeto. Começaremos criando a tag `<build>`, em que adicionaremos `<plugins>`.

```
<build>
    <plugins>

    </plugins>
</build>
```

Na documentação do Maven PMD plugin (https://maven.apache.org/plugins/maven-pmd-plugin/usage.html), clicaremos na opção "Usage" no menu. Encontraremos alguns exemplos de inclusão desse plugin no projeto, na sessão `<reporting>`.

Contudo, nosso interesse está na sessão `<build>`, como mostra o exemplo da documentação:

```
<project>
  <!-- ... -->
  <build>
    <plugins>
      <plugin>
```

```
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.6</version>
      </plugin>
    </plugins>
  </build>
  <!-- ... -->
</project>
```

Usaremos as linhas que contêm as informações de `<groupId>`, `<artifactId>` e `<version>`, e as colaremos no arquivo `pom.xml` do nosso projeto. Antes de acionarmos o comando `mvn verify` na linha de comando, iremos incluir a variável `comida` na classe `Produto`, apenas para que o PMD aponte um possível erro em nosso código.

```
public class Produto {
    private final String nome;
    private final double preco;
    private final String categoria = "comida";

    public Produto(String nome, double preco) {
        super();
        this.nome = nome;
        this.preco = preco;
    }

    public String getNome() {
        return nome;
    }

    public double getPreco() {
        return preco;
    }
}
```

No terminal, executaremos o comando `mvn verify`. Embora tenhamos um erro propositalmente colocado em nosso projeto, a verificação não consegue detectá-lo. Isso porque o PMD **não** é executado.

Isto indica que não basta mencionarmos o uso do plugin; devemos indicar que ele alterará o ciclo de vida do projeto. No arquivo `pom.xml`, adicionaremos a tag `<executions>` para especificarmos quando o plugin deverá ser executado, afinal podem haver múltiplas execuções ao longo do *build*. Em nosso caso, será apenas uma execução na fase (`<phase>`) de verificação (`verify`), cujo objetivo (`<goals>`) é `check`.

```
<project>
  <!-- ... -->
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>3.10.0</version>
        <executions>
            <execution>
                <phase>verify</phase>
                    <goals>
```

```
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
  </build>
  <!-- ... -->
</project>
```

Quando chegarmos na fase de verificação, serão executados todos os objetivos declarados no arquivo `pom.xml`, no caso, apenas `pmd:check`. A partir dessa modificação, todas as vezes que executarmos o comando `mvn verify` passaremos pelas fases de validação, compilação, pacotes, e então chegaremos à verificação.

Ao executarmos o comando no terminal, veremos que foi detectada uma violação às regras do PMD, e nós podemos verificar esse erro no relatório. Deste modo, somos capazes de utilizar um plugin explicitamente, como `pmd:pmd` ou `pmd:check`, e de alterar o padrão do nosso projeto no ciclo de vida. Da mesma maneira que trabalhamos com o PMD, podemos fazê-lo com quaisquer outros plugins que nos sejam úteis.