07

Um pouco das várias opções do Maven

Transcrição

Aprendemos a executar um plugin que escolhemos em uma fase do *build* da nossa preferência. Nesta aula conheceremos mais um exemplo de plugin de **cobertura de testes**, muito utilizado para verificar a porcentagem de cobertura dos testes aplicados em um projeto. Desse modo conseguiremos saber qual trecho falta ser testado.

Buscaremos no Google o termo "maven java test coverage plugin". Dentre as opções, é exibida JaCoCo - Maven Plugin - EclEmma (https://www.eclemma.org/jacoco/trunk/doc/maven.html). O ***JaCoCo*** é o plugin padrão de cobertura de testes em Java.

No tópico "Usage" da página de documentação, encontraremos o código de implementação desse plugin, e o incluiremos no nosso arquivo `pom.xml`.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.6-SNAPSHOT</version>
</plugin>
```

É importante conferir a versão utilizada. Por padrão, o Maven sempre fará o download da última versão disponível do plugin. Em nosso terminal executaremos o *goal* `help` — todos os *goals* do plugin estão registrados na documentação — por meio do comando `mvn jacoco:help`.

O plugin JaCoCo será baixado automaticamente em sua última versão. Contudo, **a versão mais atualizada de um plugin nem sempre pode ser a opção mais interessante para o nosso projeto**. Caso tenhamos uma biblioteca ou plugin que funciona de uma determinada maneira em uma versão, isto pode acarretar em mudanças substanciais no funcionamento do projeto, gerando resultados diferentes.

É arriscado deixarmos a versão de um plugin ou biblioteca em aberto. Na prática, é comum **fixarmos a versão de uma biblioteca no projeto**, que no caso será `0.7.5.201505241946`.

```
<****!****>

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.5.201505241946</version>
</plugin>
```

Em cada projeto podemos tomar a decisão de atualizarmos ou não nossas bibliotecas, e em ambos os casos encontraremos vantagens e desvantagens. No caso de fixarmos a biblioteca, podemos encontrar problemas em "buildar" o projeto caso desejemos realizar pequenas modificações, por exemplo.

Voltemos nossa atenção para o JaCoCo — para o plugin gerar a cobertura do nosso código, existem duas fases a serem cumpridas: **preparação do agente (*prepare-agent*)** e a **geração do relatório (*report*)**.

No arquivo `pom.xml` criaremos a tag `<executions>`, que abrigará `<execution>`. Nesta última tag incluiremos o `<goal>` a ser executado, isto é, `prepare-agent`.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.5.201505241946</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

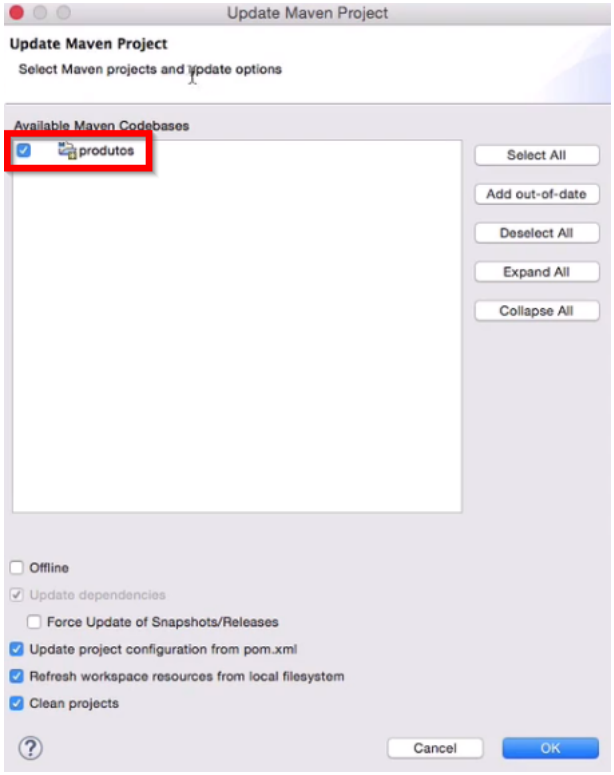
Neste caso não especificaremos a fase em que deve ocorrer a execução do plugin, então será mantida a fase padrão. Acionaremos o comando `mvn verify`, e o JaCoCo será executado pouco antes da compilação do projeto. No entanto, nosso programa não foi compilado porque possuímos uma violação do PDM: a variável inutilizada `categoria`, da classe `Produto`, que removeremos.

Feito isso, executaremos novamente o comando `mvn verify`, e a compilação ocorrerá sem problemas. O próximo objetivo a ser executado é a geração de um relatório (`report`).

```
<executions>
    <execution>
        <goals>
            <goal>prepare-agent</goal>
            <goal>report</goal>
        </goals>
    </execution>
</executions>
```

No Eclipse, teremos o seguinte aviso de erro na parte inferior da tela: `Maven Problems (1 item) > Project configuration is not up-to-date with pom.xml.` . Ou seja, configuração do projeto não está atualizada com o arquivo `pom.xml`.

Para resolvermos este problema acionaremos o atalho "Ctrl + 3" e acessaremos o buscador do Eclipse. Procuraremos pela opção "Update Project" e clicaremos sobre ela. Na caixa de diálogo teremos marcado o projeto `produtos`, que deverá ser atualizado. Pressionaremos o botão "OK".



Executaremos a verificação novamente, na linha de comando, por meio de `mvn verify`. Poderemos acessar o relatório de cobertura que foi armazenado em "produtos > target > site > jacoco > index.hrml".

Element	Missed Instructions Cov	Missed Branches Cov	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App	0%	n/a	2	2	4	4	2	2	1	1
Produto	0%	n/a	3	3	6	6	3	3	1	1
Total	40 of 40	0 of 0 n/a	5	5	10	10	5	5	2	2

A classe `Produto` não possui nenhuma cobertura, afinal os testes não fazem nada quando são executados. Criaremos um novo teste básico que execute algo. Removeremos o teste `AppTest.java`, que utiliza uma versão muito antiga do JUnit (`3.8.1`), e iremos atualizar nosso arquivo `pom.xml` para que seja utilizada uma nova versão, a `4.12`.

> Para sabermos qual a versão mais recente basta consultar o *Maven Repository*, como já aprendemos.

```
<****!****>
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<****!****>
```

No pacote `br.com.alura.maven`, criaremos uma nova classe de teste denominada `ProdutoTest`, em que implementaremos o método de teste `verificaPrecoComImposto()`.

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

public class ProdutoTest {

    public void verificaPrecoComImposto() {

    }
}
```

Se temos um produto `bala` que custa `0.10`, e o acréscimo de `10%` de imposto, o valor será `0.11`. Vamos usar o método `assertEquals()`, que recebe como parâmetro `0.11, bala.getPrecoComImposto()`.

```
public class ProdutoTest {

    public void verificaPrecoComImposto() {
        Produto bala = new Produto("juquinha", 0.10);
        assertEquals(0.11, bala.getPrecoComImposto());
    }
}
```

Implementaremos o método `getPrecoComImposto()` por meio do atalho "Ctrl + 1", clicando sobre a opção "Create method". Na classe `Produto`, declararemos que o método devolverá `double` e retorna `preco * 1.10`.

```
public double getPrecoComImposto() {
    return preco * 1.10;
}
```

O Eclipse alega que o método `assertEquals()` não existe. Não iremos criá-lo, mas sim importá-lo:

```
import static org.junit.Assert.assertEquals;

public class ProdutoTest {

    public void verificaPrecoComImposto() {
        Produto bala = new Produto("juquinha", 0.10);
        assertEquals(0.11, bala.getPrecoComImposto());
    }
}
```

O método `assertEquals()` está *deprecated* ("depreciado", em português), e quando comparamos dois *doubles*, precisamos declarar o erro aceitável, uma vez que *double* possui erro. Incluiremos uma margem de erro de `0.00001`.

```
import static org.junit.Assert.assertEquals;

public class ProdutoTest {

    public void verificaPrecoComImposto() {
        Produto bala = new Produto("juquinha", 0.10);
        assertEquals(0.11, bala.getPrecoComImposto(), 0.0001);
```

```
    }
}
```

Adicionaremos `@Test` e utilizaremos "Ctrl + Shift + O" para realizar a importação.

```
import static org.junit.Assert.assertEquals;

import org.jnuit.Test;

public class ProdutoTest {

    @Test
    public void verificaPrecoComImposto() {
        Produto bala = new Produto("juquinha", 0.10);
        assertEquals(0.11, bala.getPrecoComImposto(), 0.0001);
    }
}
```

Desse modo criamos nosso próprio teste, o qual executaremos indo à área "Project Explorer" do Eclipse e clicando sobre `ProdutoTest.java` com o botão direito do mouse. Assim feito, selecionaremos as opções "Run As > JUnit Test".

Para realizar a execução do Maven, iremos até a linha de comando e escreveremos `mvn verify`. O Maven fará o download das versões mais recentes de relatórios do JUnit e depois irá gerar o relatório referente ao nosso projeto. Verificamos que `31%` do código está coberto pelo teste. Ao clicarmos em `br.com.alura.maven`, verificaremos a cobertura da classe `Produto`, de `70%`.

Podemos, ainda, analisar os pormenores da classe `Produto`, e ao clicarmos no método `getPrecoComImposto()`, conseguiremos averiguar que o construtor foi invocado, assim como o método `getPrecoComImposto()`, diferentemente dos *getters*, que não foram invocados.

O foco principal da demonstração que fizemos não é aprender exatamente sobre o plugin JaCoCo ou como escrever testes, mas sim **a possibilidade de uso de plugins variados para alterar o ciclo de programação**. Continuamos trabalhando no Eclipse normalmente, mas quando queremos gerar o *build* completo, podemos utilizar o `mvn verify`.

O interessante é que é possível alterar o ciclo de vida do *build* do projeto por meio do `pom.xml` ao configurarmos os plugins e as execuções de acordo com a necessidade do projeto. Conseguimos, inclusive, mudar a versão do JUnit, e executar o PMD com o Eclipse.