

Curso Introdução ao Teste de Software - USP

Plataforma Coursera

Leonardo Simões

14/04/2019

1. Cenário geral do CamelCase

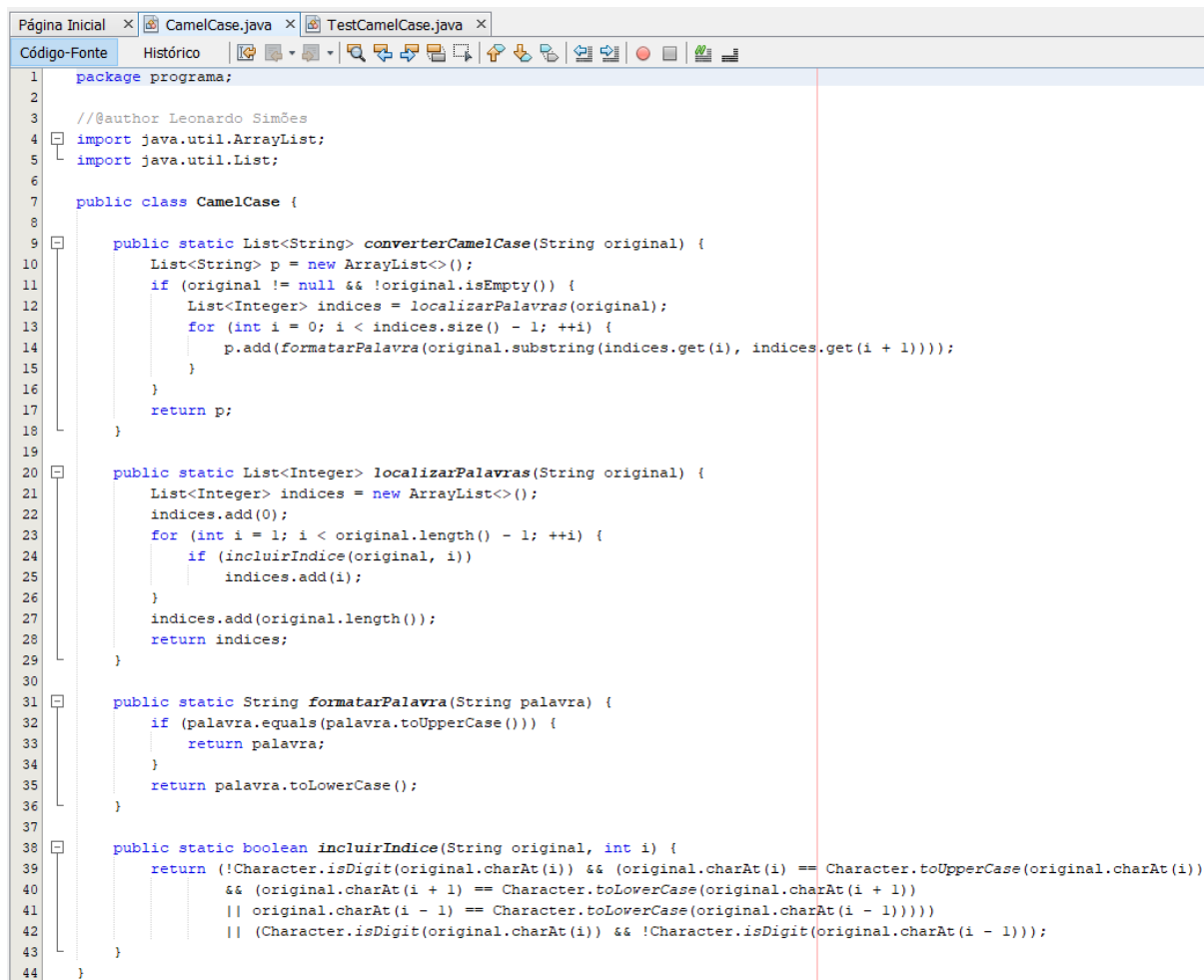
Será testado o algoritmo CamelCase, que recebe uma String e devolve uma lista de palavras separadas pelo critério de CamelCase. Esse critério consiste em detectar palavras diferentes constando que o primeiro caractere de uma String; uma letra maiúscula precedida de uma minúscula ou de um dígito; ou dígito precedido por uma letra; ou letra precedida por dígito, indicam o primeiro caractere de uma “palavra”. Nesse caso, palavra, além de palavra propriamente dita, também denota um número (que é formado por um ou mais dígitos). A palavra capitalizada na String original é retornada toda em minúscula na lista, e palavras com todas as letras maiúsculas permanecem inalteradas. Após detectadas as palavras na String original, estão são divididas e adicionadas a uma lista que será o retorno do método.

Esse algoritmo deve ser implementado em um método estático de uma classe, com um parâmetro String chamado original, e com retorno de uma lista de String. Dentro deste método podem ser chamados outros métodos.

O algoritmo deve ser implementado de preferência na linguagem Java.

A funcionalidade testada será a saída para determinadas entradas.

O único módulo verificado será o método principal do algoritmo, não diretamente os métodos que são chamados internamente.



```
1 package programa;
2
3 // @author Leonardo Simões
4 import java.util.ArrayList;
5 import java.util.List;
6
7 public class CamelCase {
8
9     public static List<String> converterCamelCase(String original) {
10         List<String> p = new ArrayList<>();
11         if (original != null && !original.isEmpty()) {
12             List<Integer> indices = localizarPalavras(original);
13             for (int i = 0; i < indices.size() - 1; ++i) {
14                 p.add(formatarPalavra(original.substring(indices.get(i), indices.get(i + 1))));
15             }
16         }
17         return p;
18     }
19
20     public static List<Integer> localizarPalavras(String original) {
21         List<Integer> indices = new ArrayList<>();
22         indices.add(0);
23         for (int i = 1; i < original.length() - 1; ++i) {
24             if (incluirIndice(original, i))
25                 indices.add(i);
26         }
27         indices.add(original.length());
28         return indices;
29     }
30
31     public static String formatarPalavra(String palavra) {
32         if (palavra.equals(palavra.toUpperCase())) {
33             return palavra;
34         }
35         return palavra.toLowerCase();
36     }
37
38     public static boolean incluirIndice(String original, int i) {
39         return (!Character.isDigit(original.charAt(i)) && (original.charAt(i) == Character.toUpperCase(original.charAt(i))
40             && (original.charAt(i + 1) == Character.toLowerCase(original.charAt(i + 1))
41             || original.charAt(i - 1) == Character.toLowerCase(original.charAt(i - 1)))))
42             || (Character.isDigit(original.charAt(i)) && !Character.isDigit(original.charAt(i - 1)));
43     }
44 }
```

Figura 1 - Implementação do algoritmo CamelCase

2. Estratégia(s) de Teste (como será testado)

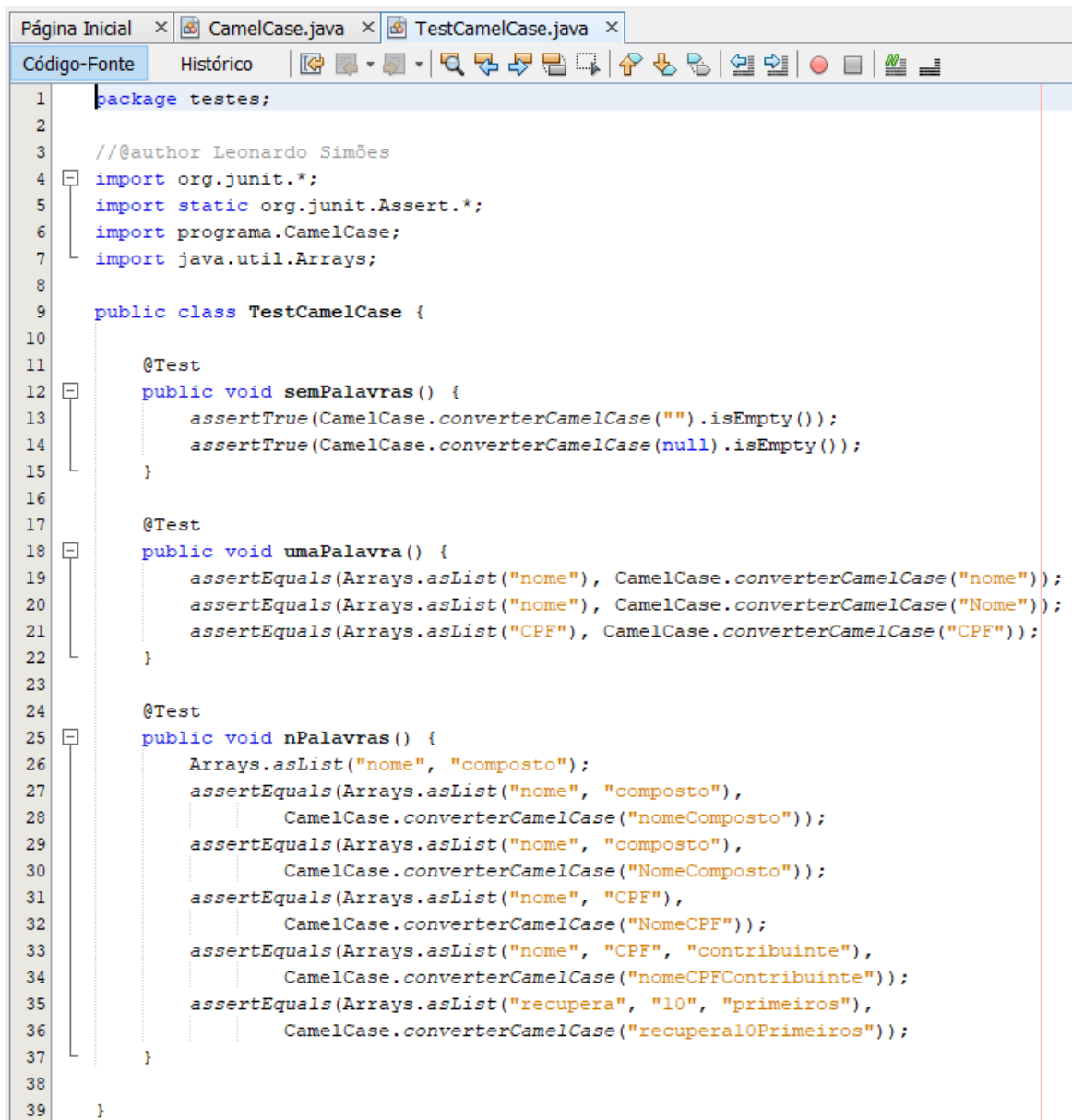
Serão realizados testes unitários funcionais que verificarão se a saída é válida para determinados casos de entrada. O framework JUnit será utilizado, através de uma classe de teste com métodos de teste e uso das devidas annotations.

3. Projeto de Casos de Teste (como será testado)

Para os testes unitários será utilizado o JUnit.

Os testes funcionais serão feitos considerando as classes de Equivalência:

- String vazia ou nula: retorna uma lista vazia
- String com uma palavra: retorna uma lista com uma palavra
- String com n palavras: retorna uma lista com n palavras



```
1 package testes;
2
3 // @author Leonardo Simões
4 import org.junit.*;
5 import static org.junit.Assert.*;
6 import programa.CamelCase;
7 import java.util.Arrays;
8
9 public class TestCamelCase {
10
11     @Test
12     public void semPalavras() {
13         assertTrue(CamelCase.converterCamelCase("").isEmpty());
14         assertTrue(CamelCase.converterCamelCase(null).isEmpty());
15     }
16
17     @Test
18     public void umaPalavra() {
19         assertEquals(Arrays.asList("nome"), CamelCase.converterCamelCase("nome"));
20         assertEquals(Arrays.asList("nome"), CamelCase.converterCamelCase("Nome"));
21         assertEquals(Arrays.asList("CPF"), CamelCase.converterCamelCase("CPF"));
22     }
23
24     @Test
25     public void nPalavras() {
26         Arrays.asList("nome", "composto");
27         assertEquals(Arrays.asList("nome", "composto"),
28             CamelCase.converterCamelCase("nomeComposto"));
29         assertEquals(Arrays.asList("nome", "composto"),
30             CamelCase.converterCamelCase("NomeComposto"));
31         assertEquals(Arrays.asList("nome", "CPF"),
32             CamelCase.converterCamelCase("NomeCPF"));
33         assertEquals(Arrays.asList("nome", "CPF", "contribuinte"),
34             CamelCase.converterCamelCase("nomeCPFContribuinte"));
35         assertEquals(Arrays.asList("recupera", "10", "primeiros"),
36             CamelCase.converterCamelCase("recuperal0Primeiros"));
37     }
38 }
39 }
```

Figura 2 - Classe de teste para o algoritmo CamelCase

4. Execução (quando e como será testado)

Os testes foram realizados utilizando o JUnit através da NetBeans IDE.
O algoritmo passou em todos os testes implementados.

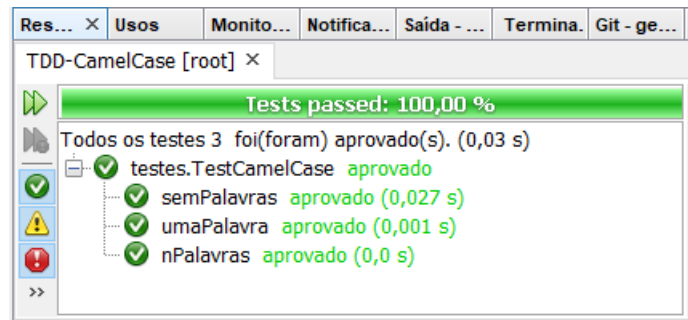


Figura 3 - Resultados dos teste do algoritmo CamelCase

5. Análise dos Resultados e próximos passos

Após os testes eu cheguei à conclusão que estes testes funcionais cobrem grande parte da análise quantitativa e qualitativa do problema, mas a implantação de outros testes poderia vir a “garantir” que o algoritmo e esses testes estão devidamente corretos e cobrem casos suficientes. Os testes estruturais viriam a acrescentar dados de testes, mas não são essenciais nesse caso, uma vez que o algoritmo não possui (na minha implementação) mais que uma estrutura condicional. Os testes de mutação seriam interessantes para esse caso porque ao percorrer índices é feita uma verificação com uma quantidade considerável de operadores relacionais.

Um próximo passo seria realizar testes de mutação para os operadores relacionais através de alguma outra ferramenta de teste de software.