Relatório Atividade 1

Desenvolvimento de Software Guiado por Testes - ITA

Começamos com o teste inicial abaixo, que verifica a resposta para uma string vazia.

```
@Test
public void testaStringVazia() {
	ConversorCamelCase c = new ConversorCamelCase();
	List<String> lista = c.converterCamelCase("").
	assertTrue(lista.isEmpty());
}
```

Permitimos que o Eclipse crie as classes e métodos faltantes, e então vemos o teste acima falhar. Implementamos à seguir o seguinte trecho de código:

```
import java.util.ArrayList;
import java.util.List;

public class ConversorCamelCase {

	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		return lista;
	}
}
```

E nosso primeiro teste passa.

Adicionamos então o seguinte caso teste, e vemos ele falhar.

```
@Test
public void testaStringMinuscula() {
	ConversorCamelCase c = new ConversorCamelCase();
	List<String> listaRecebida = c.converterCamelCase("nome");
	List<String> listaEsperada = new ArrayList<String>();
	listaEsperada.add("nome");
	assertEquals(listaRecebida, listaEsperada);
}
```

Para passar no novo caso de teste, o código é reformulado dessa forma:

```
import java.util.ArrayList;
import java.util.List;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		lista.add(string);
		return lista;
	}
}
```

Mas o primeiro caso teste já não passa mais. Percebemos que, na verdade, uma string vazia deve retornar uma lista com uma string vazia, não uma lista vazia, e portanto o erro não era o código mas sim o próprio caso de teste. Reformulamos o teste, como visto abaixo, e agora ambos passam.

```
@Test
public void testaStringVazia() {
	ConversorCamelCase c = new ConversorCamelCase();
	List<String> listaRecebida = c.converterCamelCase("");
	List<String> listaEsperada = new ArrayList<String>();
	listaEsperada.add("");
	assertEquals(listaRecebida, listaEsperada);
}
```

Antes de criar o próximo teste, refatoramos a classe de testes separando as declarações de variáveis e inicializações que são usadas em todos os testes, como visto abaixo.

```
private List<String> listaEsperada;
private ConversorCamelCase c;
private List<String> listaRecebida;

@Before
public void inicializador() {
	c = new ConversorCamelCase();
	listaEsperada = new ArrayList<String>();
}
```

Então adicionamos o próximo teste, visto abaixo.

```
@Test
public void testaUnicaPalavra() {
	listaRecebida = c.converterCamelCase("Nome");
	listaEsperada.add("nome");
	assertEquals(listaRecebida, listaEsperada);
}
```

E o teste falha, retornando “Nome” ao invés de “nome”.
Com a seguinte alteração no código principal, todos os testes passam:

```
import java.util.ArrayList;
import java.util.List;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		lista.add(string.toLowerCase());
		return lista;
	}
}
```

Nosso próximo teste será o seguinte:

```
@Test
public void testaPalavraComposta() {
	listaRecebida = c.converterCamelCase("nomeComposto");
	listaEsperada.add("nome");
	listaEsperada.add("composto");
	assertEquals(listaRecebida, listaEsperada);
}
```

Após a seguinte alteração no código principal, dois testes falham (testaPalavraComposta e testaUnicaPalavra), pois a ação de split está eliminando os caracteres maiúsculos.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		lista = Arrays.asList(string.split("[A-Z]"));
		return lista;
	}
}
```

Alterando a string de regex usada no split para "(?=[A-Z])", esse problema é resolvido, mas os testes ainda falham porque algumas palavras da listaRecebida são maiúsculas. Consertamos esse problema alterando o código da seguinte forma:

```
import java.util.ArrayList;
import java.util.List;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		String[] listaComMaiusculas = string.split("(?=[A-Z])");
		List<String> lista = new ArrayList<String>();
		for(String palavra: listaComMaiusculas) {
			lista.add(palavra.toLowerCase());
		}
		return lista;
	}
}
```

E todos os testes passam novamente. Lembrando da regra do máximo de 10 linhas por metodos, é feita a seguinte refatoração (somente estética, mantendo todos os comandos do jeito que estão):

```
public List<String> converterCamelCase(String string) {
	String[] listaComMaiusculas = string.split("(?=[A-Z])");
	List<String> lista = new ArrayList<String>();
	for(String palavra: listaComMaiusculas)
		lista.add(palavra.toLowerCase());
	return lista;
}
```

E o método agora tem apenas 5 linhas em seu corpo (embora, na minha opinião, esteja mais difícil de ler do que antes).

Passamos para o próximo teste:

```
@Test
public void testaPalavraCompostaMaiuscula() {
	listaRecebida = c.converterCamelCase("NomeComposto");
	listaEsperada.add("nome");
	listaEsperada.add("composto");
	assertEquals(listaRecebida, listaEsperada);
}
```

Que já passa automaticamente. Criamos, então, mais um teste:

```
@Test
public void testaPalavraInteiraMaiuscula() {
	listaRecebida = c.converterCamelCase("CPF");
	listaEsperada.add("CPF");
	assertEquals(listaEsperada, listaRecebida);
}
```

E esse teste, como esperado, falha, pois nosso método retorna uma lista contendo (“c”, “p”, “f”).
Contornamos o problema com a seguinte solução:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.*;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		if(string.toUpperCase().equals(string)) { //verifica se toda maiuscula
			lista.add(string);
		} else {
			String[] listaComMaiusculas = string.split("(?=([A-Z]+))");
			for(String palavra: listaComMaiusculas)
				lista.add(palavra.toLowerCase());
		}
		return lista;
	}
}
```

E passamos para o próximo teste:

```
@Test
public void testaPalavraInteiraMaisuculaComposta() {
	listaRecebida = c.converterCamelCase("numeroCPF");
	listaEsperada.add("numero");
	listaEsperada.add("CPF");
	assertEquals(listaEsperada, listaRecebida);
}
```

Que, como esperado, falha.
Através das seguintes mudanças, o novo teste passa:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.*;

public class ConversorCamelCase {

	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		Pattern pattern = Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find()) {
			lista.add(matcher.group(0));
		}
		return lista;
	}
}
```

Nota-se que agora, ao invés de usar o método ‘split(String regex)’ da classe String para dividir a string automaticamente, utilizamos as classes Pattern e Matcher, do pacote básico de expressões regulares da linguagem Java, para selecionar os grupos de interesse, No caso, a primeira parte do padrão `"([A-Z][a-z]+)”` padrão seleciona sequências começadas por uma letra maiúscula seguida por uma cadeia de letras maiúsculas. A segunda parte `“([a-z]+)”` seleciona uma cadeia de letras minúsculas qualquer, e a terceira parte `“([A-Z]+)"` faz o mesmo para cadeias de maiúsculas. Vale notar que o padrão está em ordem de prioridade.

Porém alguns dos testes antigos falham, pois as palavras que começavam com letra maiúscula não estão mais sendo retornadas com letra minúscula. Com a seguinte alteração, todos os testes passam:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.*;

public class ConversorCamelCase {
	public List<String> converterCamelCase(String string) {
		List<String> lista = new ArrayList<String>();
		if(string.isEmpty()) {
			lista.add("");
		}
		Pattern pattern = Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find()) {
			if(!matcher.group(0).toUpperCase().equals(matcher.group(0))) //se
palavra nao eh inteira maiuscula
				lista.add(matcher.group().toLowerCase());
			else
				lista.add(matcher.group(0));
		}
		return lista;
	}
}
```

Claramente, porém, é necessário efetuar refatoração para facilitar a leitura do código e diminuir o número de linhas do método principal. Ao fim da refatoração, obtivemos o código à seguir.

```
public class ConversorCamelCase {
	List<String> listaResposta;

	public List<String> converterCamelCase(String string) {
		listaResposta = new ArrayList<String>();
		if(string.isEmpty())
			listaResposta.add("");
		Pattern pattern = Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find())
			listaResposta.add(formataResposta(matcher.group(0)));
		return listaResposta;
	}

	private String formataResposta(String string) {
		if(!inteiraMaiuscula(string)) {
			return string.toLowerCase();
		} else {
			return string;
		}

	}

	private boolean inteiraMaiuscula(String palavra) {
		return palavra.toUpperCase().equals(palavra);
	}
}
```

Agora que o código está refatorado, passamos para o próximo teste.

```
@Test
public void testaPalavraInteiraMaiusculaEntrePalavras() {
	listaRecebida = c.converterCamelCase("numeroCPFContribuinte");
	listaEsperada.add("numero");
	listaEsperada.add("CPF");
	listaEsperada.add("contribuinte");
	assertEquals(listaEsperada, listaRecebida);
}
```

Que falha, pois o “C” de contribuinte é colocado no mesmo grupo de “CPF”, assim retornando (“numero”, “CPFC”, “ontribuinte”). Esse problema é corrigido simplesmente substituindo o último grupo do padrão de regex utilizado por [A-Z]+(?![a-z]+), utilizando um *negative lookahead* para fazer com que cadeias maiúsculas não selecionem caracteres seguidos por minúsculas.

Passamos, então, para o próximo teste:

```
@Test
public void testaNumeroEntrePalavras() {
	listaRecebida = c.converterCamelCase("recupera10Primeiros");
	listaEsperada.add("recupera");
	listaEsperada.add("10");
	listaEsperada.add("primeiros");
	assertEquals(listaEsperada, listaRecebida);
}
```

Que falha, pois o regex atual não pega números. Corrigimos isso acrescentando "|([0-9]+)" ao padrão.

Passamos para o próximo teste, que deve lançar uma exceção de CamelCase inválido quando a string passada é iniciada com um número:

```
@Test(expected=CamelCaseInvalidoException.class)
public void testaPalavraComecandoPorNumero() {
	listaRecebida = c.converterCamelCase("10Primeiros");
}
```

Criamos então a classe CamelCaseInvalidoException e fazemos com que ela seja lançada no caso descrito acima através das seguintes alterações no método principal:

```
public List<String> converterCamelCase(String string) {
	listaResposta = new ArrayList<String>();
	if(string.isEmpty()) {
		listaResposta.add("");
	} else {
		char[] firstCharacter = new char[1];
		string.getChars(0, 1, firstCharacter, 0);
		if(firstCharacter[0] > '0' && firstCharacter[0] < '9')
			throw new CamelCaseInvalidoException("Nao pode comecar com
numeros");
		Pattern pattern =
Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+(?![a-z]+))|([0-9]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find())
			listaResposta.add(formataResposta(matcher.group(0)));
	}
	return listaResposta;
}
```

Após a refatoração, temos o novo método auxiliar à seguir:

```
private boolean comecaComNumero(String string) {
	char[] firstCharacter = new char[1];
	string.getChars(0, 1, firstCharacter, 0);
	return (firstCharacter[0] > '0' && firstCharacter[0] < '9');
}
```

E o método principal fica da seguinte forma:

```
public List<String> converterCamelCase(String string) {
	listaResposta = new ArrayList<String>();
	if(string.isEmpty()) {
		listaResposta.add("");
	} else if (comecaComNumero(string)) {
		throw new CamelCaseInvalidoException("Nao pode comecar com numeros");
	} else {
		Pattern pattern =
Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+(?![a-z]+))|([0-9]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find())
			listaResposta.add(formataResposta(matcher.group(0)));
	}
	return listaResposta;
}
```

Passamos, então, para o último teste proposto:

```
@Test(expected=CamelCaseInvalidoException.class)
public void testaPalavraComCaracterEspecial() {
	listaRecebida = c.converterCamelCase("nome#Composto");
}
```

Que deve lançar uma exceção de CamelCase invalido caso sejam encontrados caracteres especiais na string recebida. Alcançamos esse objetivo através do seguinte método auxiliar:

```
private boolean contemCaracteresEspeciais(String string) {
	Pattern pattern = Pattern.compile("[^A-Za-z0-9]");
	Matcher matcher = pattern.matcher(string);
	return matcher.find();
}
```

Que é introduzido no método principal da seguinte forma:

```
public List<String> converterCamelCase(String string) {
	listaResposta = new ArrayList<String>();
	if(string.isEmpty()) {
		listaResposta.add("");
	} else if (comecaComNumero(string)) {
		throw new CamelCaseInvalidoException("Nao pode comecar com numeros");
	} else if (contemCaracteresEspeciais(string)) {
		throw new CamelCaseInvalidoException("Nao pode conter caracteres
especiais");
	} else {
		Pattern pattern =
Pattern.compile("([A-Z][a-z]+)|([a-z]+)|([A-Z]+(?![a-z]+))|([0-9]+)");
		Matcher matcher = pattern.matcher(string);
		while(matcher.find())
			listaResposta.add(formataResposta(matcher.group(0)));
	}
	return listaResposta;
}
```

Agora que todos os testes passam, só resta fazer a refatoração final para garantir a legibilidade do código e cumprir a proposta do exercício de não possuir métodos com mais de 10 linhas em seu corpo. Encapsulando as duas verificações da string recebida (se começa com números e se contém caracteres especiais) em um único método e separando as declarações e inicialização de atributos do método principal, colocando-as dentro do construtor da classe, obteve-se o código final que segue em anexo.