Lista 2 de Exercícios - Engenharia de Software II

Aluno: Mateus Filipe de Lima Souza **Data:** 27/09/20

1. E) Unidade e Integração

2. Um dos benefícios é o de encontrar bugs antes do lançamento final de um projeto ou de lançar alguma atualização do mesmo. Outro benefício muito importante, o fato do teste de unidade funcionar como uma rede de proteção contra a regressão no código, a regressão sendo a inclusão de um novo bug devido a alguma alteração no código. Além de tudo isso, os testes de unidade ajudam na documentação e especificação do código de produção.

3. *Código em anexo.
    a. fibonacci.java: implementação da função de fibonacci;
    b. Lista2_testes.java: função testaFibonacci() com o teste de fibonacci;

4. *Código em anexo.
    a. Lista2_testes.java: função testEmptyStackException() com o teste da Stack;

5. *Código em anexo.
    a. Lista2_teste.java: Todos os testes da questão.
        i. Teste 1: testaArrayVazia() testando se a ArrayList está vazia;
        ii. Teste 2: testaArrayComValor() testando se a ArrayList tem um valor;

iii. Teste 3: testaArrayTamanhoEValores() testando se a ArrayList tem o tamanho de acordo com a quantidade de valores e se os valores correspondem;

iv. Teste 4: testaArrayElementoRemovido() testando se a ArrayList teve um valor removido;

v. Teste 5: testaArrayVazioEDeTamanhoZero() testando se a ArrayList está vazia e tem 0 valores;

vi. Teste 6: testaArrayRemoverPosicaoAlem() testando se a ArrayList teve um valore removido.

6.

Chamada feita pelo Teste	Cobertura de Comandos	Cobertura de Branches
f (0,0)	25%	25%
f (1,1)	100%	50%
f (0,0) e f (1,1)	100%	75%

7.

a. Sim, quando a nota do aluno igual a 90 ou o fato de não receber valores não-inteiros;
b. Cobertura de Comandos: 100% e Cobertura de Branches: 100%
c. Falsa, 100% da cobertura de comandos não é o mesmo de funcionamento pleno do código, pode ocorrer de alguma linha possuir algo incorreto e ainda assim ser executado, ou pode-se ter a falta de alguma condicional na qual se deixa passar um erro, como um de divisão por zero.

8.

```
public void teste1(){
        LinkedList list = mock(LinkedList.class);
        when(list.size()).thenReturn(10);
        assertEquals(10, list.size());
}

public void test2() {
        LinkedList list = mock(LinkedList.class);
        when(list.get(0)).thenReturn("Engenharia");
        when(list.get(1)).thenReturn("Software");
        String result = list.get(0) + " " + list.get(1);
        assertEquals("Engenharia Software", result);
}
```

9. *Código em anexo.
    a. Pacote: ATV9:
        i. Aluno.java: com a classe de aluno utilizada no exemplo;
        ii. Atividade9.java: com o código “main” e a função alunoAprovado() de teste boleana;
        iii. TesteIntegração: testAlunoAprovado() testando se um aluno consegue ser aprovado com nota superior a 60;
        iv. TesteUnidade: init() seguindo o exemplo da aula de início com mock e mockito (sem funcionar devido a importação de biblioteca), testeAlunoAprovado() chamando a função de teste de nota da Atividade9.java e dando retorno de acordo.

Observação: Entendi o conceito de Mock, quanto a sua aplicação, tentei aplicá-la em um exemplo no código em anexo, mas a importação da

biblioteca não funcionou de forma alguma, nem mesmo nos testes com “mock” da questão 8. Com isso não consegui ter muita certeza se fiz da forma correta.