Programação Modular

Testes unitários

Prof Dr Johnatan Oliveira

2

Testes de Software

Contextualização

Software de qualidade

- Número e severidade dos defeitos aceitáveis
- Entregue dentro do prazo e custo;
- Atende aos requisitos/expectativas
- Pode ser mantido eficientemente após a implantação

(Rios; Moreira Filho, 2013)

Teste de software!

- Uma maneira de se garantir a qualidade!
  - Mostrar que um programa faz o que é proposto e descobrir defeitos antes do uso. *(Sommervile, 2019)*
  - Um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente. *(Pressman;Maxim, 2016)*

5

Teste Unitário

Teste unitário

- Testa cada componente (classe, método, rotina, página...) individualmente.

- Objetivo principal: garantir que cada unidade, isoladamente, funciona de acordo com sua especificação.

Teoria do teste unitário

- *“Para cada trecho de código operacional – uma classe ou um método –, o código operacional deve estar pareado com um código de teste unitário”*

“If you can’t write a test for what you are about to code, then you shouldn’t even be thinking about coding.” (George; Williams, 2003)

Teste unitário

- Chama o código operacional a partir da sua interface pública, de diversas formas, e verifica os resultados.

- Não precisa ser exaustivo.
  - código de teste já agrega valor, mesmo nos casos centrais.

9

Classe Relógio e teste unitário

Mãos na massa...

Teste unitário com xUnit e Junit

- xUnit: nome genérico para uma coleção de plataformas de testes unitários.
  - SUnit (Kent Beck, 1998)

- A primeira letra costuma indicar a plataforma.
  - JUnit, RUnit, CUnit, CppUnit, ShUnit, JSUnit...
  - xUnit.net, minitest, Pytest...

Testando a classe Relógio

- A hora é válida?
- A passagem do tempo está correta?
- A impressão está no formato correto?

Testando a classe Relógio

- A hora é válida?
  - O que acontece com hora inválida?
- A passagem do tempo está correta?
  - Teste dentro do minuto, virando minuto, virando hora?
- A impressão está no formato correto?
  - Qual o formato correto? Devemos testar para inválidos?

JUnit - Asserções

JUnit e asserts

Assert	Objetivo
`assertTrue(teste)`	falha se teste booleano é **false**.
`assertFalse(teste)`	falha se teste booleano é **true**.
`assertEquals(esperado, teste real)`	falha se valores não são iguais.
`assertNull(teste)`	falha se valor não for **null**.
`assertNotNull(teste)`	falha se valor for **null**.

JUnit e asserts

Assert	Objetivo
`assertThrows(classe esperada, execução)`	falha se a exceção não é lançada.
`assertSame(esperado, teste real)`	falha se valores não são os mesmos (por meio de ==).
`assertNotSame(esperado, teste real)`	falha se valores são os mesmos (por meio de ==).
`fail ()`	faz com que o teste interrompa sua execução e falhe.

String de informação

- Parâmetro adicional, exibido caso o teste falhe

```
assertTrue(metodo(a,b), “metodo com” +a+ “ e ” +b );
```

JUnit, anotações e informações

- Anotações para guia tanto do ambiente de testes como dos desenvolvedores

- *Strings* adicionais como informação de erros

@DisplayName

- Define um nome/descrição para a classe ou o método de teste

Inicialização e finalização

- Métodos executados antes ou depois de cada caso de teste

```
@BeforeEach
void setUp() throws Exception {
}

@AfterEach
void tearDown() throws Exception {
}
```

Inicialização e finalização

- Métodos executados uma única vez, antes ou depois da execução de toda a classe de testes.

```
@BeforeAll
static void setUpBefore() throws Exception {
}

@AfterAll
void tearDownAfter() throws Exception {
}
```

@TestMethodOrder

- Define a modo de ordem de execução dos testes:
  - MethodOrderer.MethodName.class
  - OrderAnnotation.class

    @Order(*n*)
  - MethodOrderer.Random.class