Testes Automatizados em Java e Python

Teste de Software

- Visa identificar possíveis defeitos que podem vir a acarretar falhas nas aplicações
- Aumento na garantia da qualidade do software sob teste

- Caso de teste:

  (**dado-de-entrada**, **saída-esperada**)

Teste de Software - Técnicas

- Teste Funcional
  - testa funcionalidades
- Teste Estrutural
  - utiliza informações sobre o código fonte para derivar os requisitos de teste
- Teste Baseado em Defeito
  - teste de mutação: gera cópias do código com defeitos e vê se os testes conseguem “matar os mutantes”.
  - (não é tão legal quanto o nome sugere)
- Teste Baseado em Modelo
  - deriva casos de teste automaticamente a partir de um modelo (diagrama UML, etc.)

Teste de Software - Fases

- Fases de teste:
  - Unidade (classes ou métodos testados isoladamente);
  - Integração (integração entre as unidades: método chamando método de outra classe, etc.);
  - Sistema (sistema como um todo, normalmente testam-se os requisitos não-funcionais do sistema);
  - Aceitação (mais relacionado com a validação, verificar se o sistema atende as expectativas do usuário).

Ferramentas

- Hoje usaremos:

coverage.py

Junit - Assertions

```
@Test
void testPowNotNull() {
    assertNotNull(Math.pow(3, 4));
}

@Test
void testPow3squared() {
    assertEquals(9, Math.pow(3,2));
}
```

Junit - Assertions

certifiqueQueIgual(valor_esperado, valor_obtido);

Se for igual, o teste passa, se for falso o teste falha

Junit - Assertions

- assertEquals() e assertNotEquals()
- assertNull() e assertNotNull()
- assertSame() e assertNotSame()
- assertTrue() e assertFalse()
- assertArrayEquals()
- assertThrows()
- assertDoesNotThrow()
- assertAll()
- assertTimeout()

Junit - Annotations

<- JUnit 4 ***JUnit 5 ->***

- @***Test*** demarca que a função é um teste
- @***Before*** função executada antes de cada teste; @***BeforeEach***
- @***After*** função executada após cada teste; @***AfterEach***
- @***BeforeClass*** função executada uma única vez antes de todos os testes; @***BeforeAll***
- @***AfterClass*** função executada uma única vez após todos os testes; @***AfterAll***
- @***Ignore*** teste desabilitado; @***Disabled***

Junit - Annotations

```
@BeforeEach
void setUp() {
    bst = new BinarySearchTree<>();
}

@Test
void testNotNull() {
    assertNotNull(bst);
}
```

Diferenças Junit4 e Junit5

```
@Test(expected = Exception.class)
public void shouldRaiseAnException() throws Exception {
    // ...
}

public void shouldRaiseAnException() throws Exception {
    Assertions.assertThrows(Exception.class, () -> {
        //...
    });
}
```

Exemplos

Baixe os arquivos do curso em:

https://github.com/TesteEstrutural/minicurso/

Exemplo 1: Hailstone

- O programa gera uma sequência partindo de um inteiro n conforme as regras a seguir:
  - Se n é ímpar, o próximo elemento é igual a 3n+1;
  - Se n é par, o próximo elemento é igual a n/2.
  - Esse processo é repetido até que se chegue a 1.
- Exemplo: n = 3
  - {3, 10, 5, 16, 8, 4, 2, 1}
- Vamos implementar esse caso de teste no JUnit.

Exemplo 1: Hailstone

- Veja quanto do código é coberto por esse caso de teste utilizando a ferramenta eclemma.
- Escreva novos casos de teste para alcançar a maior cobertura possível.

Exemplo 2 - Árvore Binária de Busca

Faça o mesmo para o código da árvore binária de busca. Tente cobrir a maior porcentagem possível do código.

Ferramentas

- Existem diversas outras ferramentas para o teste de software de programas em Java e Python:

Testes com python

Para realizarmos testes com a linguagem de programação python utilizaremos o unittest e a biblioteca coverage.

Unittest

- Biblioteca nativa do python que funciona de maneira similar ao Junit, nela o usuário pode declarar sua classe de testes e dentro da mesma definir uma sequência de métodos que serão utilizados para testar determinado código.

Uso

- inicialmente o usuário deverá importar no módulo onde está a criar seus casos de teste a biblioteca unittest, para isso, inserir o seguinte comando:

  import unittest

see that easy

Uso

- Depois de importar corretamente o módulo é preciso que se crie a classe com um nome qualquer só que é preciso fazer com que essa classe herde unittest.TestCase

  Portanto, a classe ficara algo próximo disso:

  class **CauseNowTheGuiltyIsAllMine(unittest.TestCase)**

Uso

- Dentro dessa classe que foi criada, serão definidos os métodos de teste

```
class CauseNowTheGuiltyIsAllMine(unittest.TestCase)

        def testEVA01(self):

                self.assertTrue(“bla”, “bla”)
```

Uso

Importante que para que o unittest execute os testes os métodos da classe precisam começar com o nome “test”

Uso

- Assim como no JUnit, o unittest oferece uma infinidade de métodos próprios para que se possam ser aferidos os testes. Alguns desses métodos são:
- **assertEqual(a, b)**
- **assertNotEqual(a, b)**
- **assertTrue(x)**
- **assertFalse(x)**
- **assertIs(a, b)**
- **assertIsNot(a, b)**
- **assertIsNone(x)**
- **assertIsNotNone(x)**
- **assertIn(a, b)**
- **assertNotIn(a, b)**
- **assertIsInstance(a, b)**
- **assertNotIsInstance(a, b)**

Coverage

- Biblioteca desenvolvida para python que é usada para medir a porcentagem de cobertura, geralmente se utilizando de casos de teste e em tempo de execução. A ferramenta utiliza bibliotecas e recursos nativos do python para chegar em tais resultados de cobertura.

Uso - Coverage

- pip install coverage

Uso

- Depois de ter feito seu módulo de testes e escrito o código a ser testado, é a hora de rodar a coverage e ver os resultados de cobertura

Uso

- vá para a pasta onde se encontram os scripts de teste e o código
- na pasta entre com o seguinte comando:
    - coverage run xxx.py

Uso

- Depois de rodar corretamente este comando, obter a resposta da coverage utilizando o seguinte comando:
  - coverage report - m

  É possível obter esta resposta em html também utilizando:

  coverage html

Uso

- Caso queira utilizar outros casos de teste ou programa, entrar com o comando
  - coverage erase

Mut@ç$o no código

O uso de teste de mutação envolve usar frameworks que geram pequenas alterações no código fonte original, a ideia é que seus casos de testes escritos previamente sejam capazes de barrar esses mutantes, ou seja, um caso de teste que passou anteriormente, em um código mutado deveria falhar.

DLC SLIDE - MUTAÇÃO NOS CÓDIGOS

- pip install mutpy
  - disponível para python 3.3+

Uso - mut.py

- utilizar o seguinte comando:
  - mut.py --target foo --unit-test unit -m
- foo é o módulo do código que “escrevemos”
- unit é o módulo de testes

Fim

- Quem gostou se inscreve no canal
- Gameplay de fortinite, minecraft e junit nos fim de semana

jk