UNIVERSIDADE FEDERAL DE SERGIPE

VINÍCIUS DIAS VALENÇA

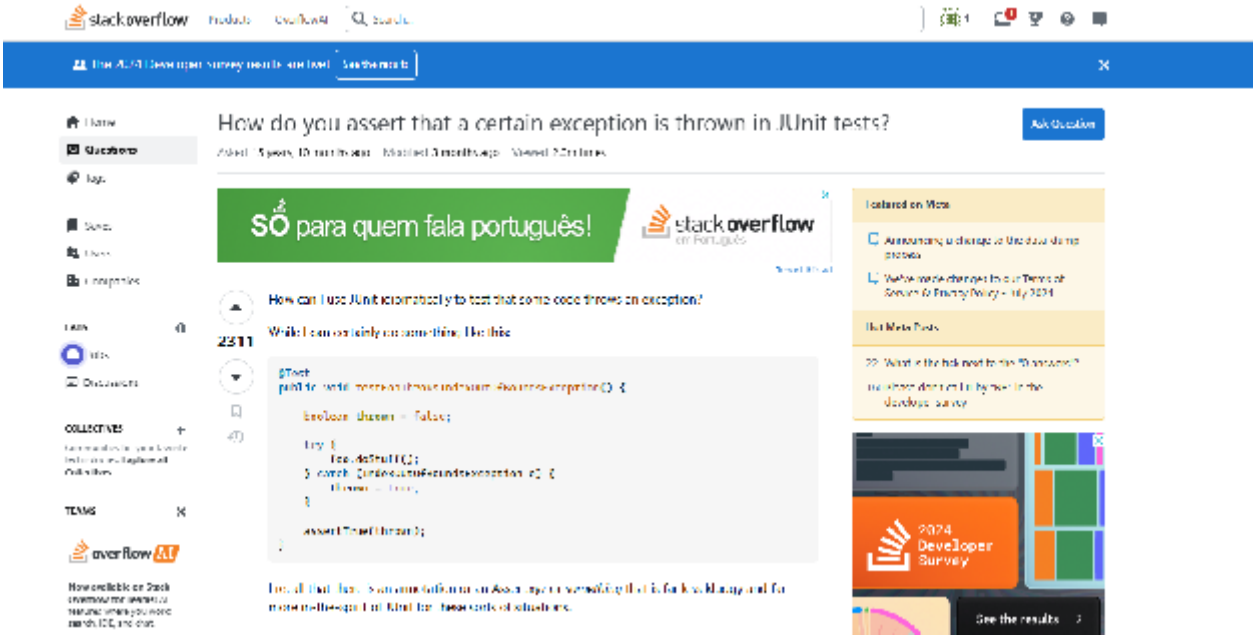
Teste DE SOFTWARE
ATIVIDADE 1

Problema do Stack Overflow

1. Motivação

How do you assert that a certain exception is thrown in JUnit tests?

https://stackoverflow.com/questions/156503/how-do-you-assert-that-a-certain-exception-is-thrown-in-junit-tests




```java
@Test
public void testFooThrowsIndexOutOfBoundsException() {

    boolean thrown = false;

    try {
        foo.doStuff();
    } catch (IndexOutOfBoundsException e) {
        thrown = true;
    }

    assertTrue(thrown);
}
```

Ao escrever testes unitários, um cenário comum é garantir que uma exceção específica seja lançada quando certas condições são atendidas. Por exemplo, se um método deve lançar uma exceção quando recebe parâmetros inválidos, queremos testar e garantir esse comportamento.

Problemas com a Abordagem Inicial

1. A necessidade de um bloco `try-catch` e a variável `thrown` adicionam código desnecessário, tornando o teste mais verboso e difícil de ler.
2. O teste depende da variável `thrown` ser corretamente definida no bloco `catch`. Qualquer alteração acidental pode fazer o teste falhar silenciosamente.
3. Se o teste falhar, a mensagem de erro padrão pode não ser muito clara sobre o que deu errado.

2. Reproduzindo o problema

```java
package com.example;
import java.util.List;

public class ClassToTest {
    public void checkNames(List<String> names) {
        for (String name : names) {
            if (name.length() <= 2) {
                throw new IllegalArgumentException("O nome precisa ter pelo menos 3 letras: " + name);
            }
        }
    }
}
```

```java
package com.example;
import org.junit.jupiter.api.Test;

import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class TestClass {

    @Test
    public void testCheckNamesThrowsExceptionForShortNames() {
        ClassToTest foo = new ClassToTest();
        List<String> names = Arrays.asList("Joe", "Al", "Sam");

        IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> foo.checkNames(names));
        assertEquals("O nome precisa ter pelo menos 3 letras: Al", exception.getMessage());
    }

    @Test
    public void testCheckNamesDoesNotThrowExceptionForValidNames() {
        ClassToTest foo = new ClassToTest();
        List<String> names = Arrays.asList("Joe", "Sam", "Ann");

        foo.checkNames(names);
    }
}
```

`testCheckNamesThrowsExceptionForShortNames`:

Cria uma instância de `ClassToTest` e uma lista de nomes onde um dos nomes é muito curto.

Usa `assertThrows` para verificar que `IllegalArgumentException` é lançada.

Verifica se a mensagem da exceção corresponde à esperada.

`testCheckNamesDoesNotThrowExceptionForValidNames`:

Cria uma instância de `ClassToTest` e uma lista de nomes válidos.

Chama `checkNames` e verifica implicitamente que nenhuma exceção é lançada.

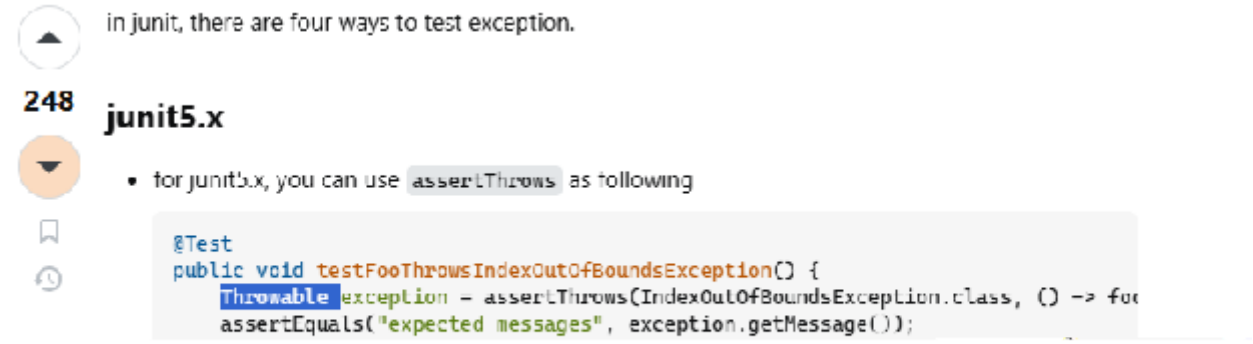
```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>atividade1-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>+
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M5</version>
            </plugin>
        </plugins>
    </build>
</project>
```

3. Outras soluções:



Utilizar uma exceção específica, como `IllegalArgumentException`, comunica claramente a intenção do código e o tipo de erro que está sendo tratado. Isso torna o código mais legível e fácil de entender para outros desenvolvedores que possam trabalhar nele no futuro.

`Throwable` é a superclasse de todas as exceções e erros em Java. Utilizá-la para capturar exceções pode levar a capturar exceções não intencionais, como `NullPointerException`, `IOException`, ou mesmo `OutOfMemoryError`, que são casos que normalmente não deveriam ser tratados da mesma forma.

ATIVIDADE 2 - Aplicação de Testes de Mutação em Python

Etapa 1: Acessando o Repositório

- Acesse o repositório do exemplo disponível no GitHub. (https://github.com/alura-cursos/2622-python-tdd)
- **Clone o repositório**:
  - https://github.com/alura-cursos/2622-python-tdd.git

Etapa 2: Preparando o Ambiente de Desenvolvimento

- Descompacte o conteúdo do repositório em uma pasta local.
- Abra o editor de código de sua preferência. No vídeo, é utilizado o **Visual Studio Code** (VS Code).
  - **Nota**: Para utilizar o VS Code com Python, algumas extensões específicas devem estar instaladas.

Etapa 3: Configuração do Ambiente Virtual

- Instale o **Python 3.7** (ou superior) e o **virtualenv**.
- Crie um ambiente virtual para isolar as bibliotecas necessárias.
  - Comando para criar o ambiente virtual:

```
python3 -m venv venv
```

  - Ative o ambiente virtual:

```
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate  # Windows
```

Etapa 4: Instalando as Dependências

- Instale as bibliotecas necessárias listadas no arquivo `requirements.txt`:

```
pip install -r requirements.txt
```

- As principais bibliotecas são:
  - **pytest**: Framework para testes.
  - **pytest-cov**: Plugin do pytest para medir a cobertura de código.
  - **mutmut**: Ferramenta para aplicar teste de mutação.

Etapa 5: Executando Testes e Medindo Cobertura

- Execute os casos de teste utilizando o pytest:

```
pytest -vv [caminho_para_o_arquivo]
```

- Verifique a cobertura de código:

```
pytest -vv [caminho_para_o_arquivo] --cov=cal
```

- Gere um relatório HTML de cobertura:

```
pytest -vv [caminho_para_o_arquivo] --cov=cal --cov-report=html
```

Etapa 6: Classe Funcionário

O código define uma classe `Funcionario` que representa um funcionário com alguns atributos e métodos para manipular e acessar suas informações.

Atributos

- `nome`: Nome do funcionário.
- `data_nascimento`: Data de nascimento do funcionário no formato 'DD/MM/AAAA'.
- `salario`: Salário do funcionário.

Métodos

- **`idade`**: Calcula a idade do funcionário com base no ano de nascimento.
- **`sobrenome`**: Retorna o sobrenome do funcionário, assumindo que é a última palavra no nome completo.
- **`_eh_socio`**: Método privado que verifica se o funcionário é considerado um "sócio" com base no salário e sobrenome.
- **`decrescimo_salario`**: Aplica um desconto de 10% no salário se o funcionário for um "sócio".
- **`calcular_bonus`**: Calcula um bônus de 10% sobre o salário. Se o valor do bônus for maior que 1000, lança uma exceção.

Etapa 7: Resumo do Código de Testes

O código fornece uma série de testes unitários para a classe `Funcionario`, utilizando o framework de testes `pytest`. Os testes verificam o comportamento dos métodos da classe `Funcionario` em diferentes cenários.

1. Teste de Idade

- **Descrição**: Verifica se o método `idade` retorna o valor correto para uma data de nascimento específica.
- **Entrada**: `'13/03/2000'`
- **Esperado**: 24 (nota: o valor original no código é 22, indicando um erro)
- **Verificação**: Compara o resultado da idade com o valor esperado.

2. Teste de Sobrenome

- **Descrição**: Verifica se o método `sobrenome` retorna o sobrenome correto a partir de um nome completo.
- **Entrada**: `' Lucas Carvalho '`

- **Esperado**: `'Carvalho'`
- **Verificação**: Compara o sobrenome obtido com o valor esperado.

3. Teste de Desconto no Salário

- **Descrição**: Verifica se o método `decrescimo_salario` aplica corretamente um desconto no salário se o funcionário for um "sócio".
- **Entrada**: Salário de `100000` e nome `'Paulo Bragança'`
- **Esperado**: `90000`
- **Verificação**: Compara o salário após o desconto com o valor esperado.

4. Teste de Cálculo de Bônus

- **Descrição**: Verifica se o método `calcular_bonus` calcula corretamente o bônus para um salário dentro do limite.
- **Entrada**: Salário de `1000`
- **Esperado**: `100`
- **Verificação**: Compara o bônus calculado com o valor esperado.

5. Teste de Exceção no Cálculo de Bônus

- **Descrição**: Verifica se o método `calcular_bonus` lança uma exceção quando o salário é muito alto.
- **Entrada**: Salário de `100000000`
- **Esperado**: Lançar uma exceção
- **Verificação**: Certifica-se de que uma exceção é lançada.

Etapa 8: Executando o Teste de Mutação

- Utilize o **Mutmut** para aplicar o teste de mutação e verificar a robustez dos casos de teste:

```
mutmut run --paths-to-mutate [caminho_para_o_arquivo]
```

- O Mutmut criará várias versões mutantes do código original e executará os testes contra essas versões.

Etapa 9: Analisando os Resultados do Teste de Mutação

- Ao final da execução, o Mutmut fornecerá um resumo dos mutantes que foram "mortos" e aqueles que sobreviveram.
- Utilize comandos do Mutmut para obter mais detalhes sobre os mutantes que sobreviveram:

```
mutmut results
```

- **Dica**: Utilize para gerar um arquivo html contendo todos os mutantes sobreviventes. Dessa forma, é possível analisá-los de forma mais detalhada.

```
mutmut html
```
