Testes de Software com Pytest

Prof. Me. Lucas Bruzzone

Aula 04

Objetivos da Aula

- Implementar testes unitários usando pytest
- Criar testes de integração eficazes
- Configurar ambiente de testes
- Aplicar fixtures e parametrização
- Medir cobertura de código
- Interpretar relatórios de teste

Por que Pytest?

Vantagens do pytest:

- Sintaxe simples e intuitiva
- Descoberta automática de testes
- Fixtures poderosas e reutilizáveis
- Plugins extensivos
- Relatórios detalhados
- Compatibilidade com unittest

Instalação necessária:

- `pip install pytest pytest-cov pytest-html`

Estrutura Básica de Teste

Convenções de nomenclatura:

- Arquivos: `test_*.py` ou `*_test.py`
- Funções: `test_*()`
- Classes: `Test*`

Estrutura básica:

- Import do pytest e módulos a testar
- Definir funções de teste com assert
- Usar `pytest.raises()` para exceções
- Aplicar decoradores para parametrização

Passos para Testes Unitários

1. **Criar a classe a ser testada:**
   - Implementar classe Calculadora
   - Métodos: soma, subtração, multiplicação, divisão
   - Tratar exceção para divisão por zero
2. **Criar arquivo de teste:**
   - Arquivo: `test_calculadora.py`
   - Importar pytest e a classe Calculadora
   - Criar classe TestCalculadora
   - Implementar `setup_method()` para inicialização
3. **Implementar casos de teste:**
   - Testes para números positivos e negativos
   - Teste de divisão normal e por zero
   - Verificar resultados com assert

Fixtures e Parametrização

Usando fixtures:

- Decorador `@pytest.fixture`
- Retornar objeto reutilizável
- Injetar como parâmetro nos testes
- Evitar duplicação de código de setup

Parametrização de testes:

- Decorador `@pytest.mark.parametrize`
- Definir lista de tuplas com dados de teste
- Executar mesmo teste com múltiplos valores
- Reduzir repetição de código

Sistema Bancário - Estrutura

Classe ContaBancaria:

- Atributos: número, saldo
- Métodos: depositar, sacar
- Validações de valor positivo e saldo suficiente

Classe SistemaBancario:

- Gerenciar múltiplas contas
- Método criar_conta
- Método transferir entre contas
- Validar contas existentes e saldo

Testes de Integração - Implementação

Setup do teste:

- Fixture para sistema bancário
- Criar contas com saldos iniciais
- Retornar sistema configurado

Cenários de teste:

- Transferência com sucesso
- Transferência com saldo insuficiente
- Transferência entre contas inexistentes
- Verificar saldos após operações

Verificações:

- Assert no resultado da operação
- Assert nos saldos das contas envolvidas
- Verificar que não houve alteração em falhas

Comandos Essenciais

Execução básica:

- `pytest` - executar todos os testes
- `pytest tests/` - pasta específica
- `pytest tests/test_calculadora.py` - arquivo específico
- `pytest -k "soma"` - filtrar por nome

Relatórios e cobertura:

- `pytest -v --cov=src` - execução com cobertura
- `pytest --cov=src --cov-report=html` - relatório HTML
- `pytest --html=reports/report.html` - relatório de testes

Execução por categoria:

- `pytest -m unit` - apenas testes unitários
- `pytest -m integration` - apenas integração

Configuração com pytest.ini

Criar arquivo pytest.ini na raiz:

- Definir diretórios de teste
- Configurar padrões de arquivos
- Adicionar opções padrão
- Criar marcadores personalizados

Principais configurações:

- `testpaths` - onde encontrar testes
- `addopts` - opções sempre aplicadas
- `markers` - categorizar testes
- Configurar relatórios de cobertura

Princípios dos Bons Testes

Características FIRST:

- **Fast**: Execução rápida
- **Independent**: Independentes entre si
- **Repeatable**: Resultados consistentes
- **Self-validating**: Assert claro
- **Timely**: Escritos junto com código

Estrutura AAA:

- **Arrange**: Preparar dados e objetos
- **Act**: Executar ação testada
- **Assert**: Verificar resultado esperado

Organização dos Testes

Estrutura de diretórios:

- `src/` - Código fonte
- `tests/` - Todos os testes
- `pytest.ini` - Configuração do pytest
- `requirements.txt` - Dependências
- `.gitignore` - Arquivos ignorados pelo git

Nomenclatura descritiva:

- Usar verbos: "deve_fazer_algo"
- Ser específico sobre o cenário
- Incluir resultado esperado
- Manter consistência no projeto

Exercício - Contador

Implementar classe Contador:

- Atributo: `valor` (inicia em 0)
- `incrementar()` - adiciona 1
- `decrementar()` - subtrai 1
- `reset()` - volta para 0
- `definir_valor(numero)` - define valor específico

Testes necessários:

- Testar incremento e decremento
- Verificar reset funciona
- Usar fixture para instância do contador
- Parametrizar teste com múltiplos valores
- Meta: 100% de cobertura

Próxima Aula

Métricas de Qualidade de Software

Estudaremos como medir e avaliar a qualidade do software através de métricas objetivas