

```
re File.expand_path("../../config/environment", __FILE__)
# Prevent database truncation if the database needs
# clearing for testing -- uncomment this line if necessary.
# ActiveRecord::Base.establish_connection("test")
# require 'spec_helper'
# require 'rspec/rails'

# require 'capybara/rspec'
# require 'capybara/rails'

Capybara.javascript_driver = :webkit
Category.delete_all; Category.create!
Shoulda::Matchers.configure do |config|
  config.integrate do |with|
    with.test_framework :rspec
    with.library :rails
  end
end

# Add additional requires below this line to include
# more frameworks or libraries in your spec files.
#
# Requires supporting ruby files with custom matchers and
# helper methods can be found in spec/support/
# run as spec files by default. This means they will be
# run twice. It is recommended that most of these
# end with _spec.rb. You can configure the
# `:support` keyword option to .rb files here.
#
# in _spec.rb will both be required as
# run twice. It is recommended that most of these
# end with _spec.rb. You can configure the
# `:support` keyword option to .rb files here.
#
# action on the command line.
#
# Mongoid
# Mongoid::Buffer
```

O que são testes de software?

Testes de software são um conjunto de atividades que têm como objetivo identificar possíveis erros, bugs ou problemas em um programa computacional. Eles são realizados por diversos profissionais da área de Tecnologia da Informação e são essenciais para garantir que um sistema esteja funcionando corretamente.



by **Jéssica Oliveira**



Made with Gamma

Benefícios dos testes

Redução de bugs e erros

A realização de testes ajuda a identificar problemas no código antes mesmo de o software ser colocado em produção, evitando assim que clientes encontrem erros.

Melhoria na qualidade do software

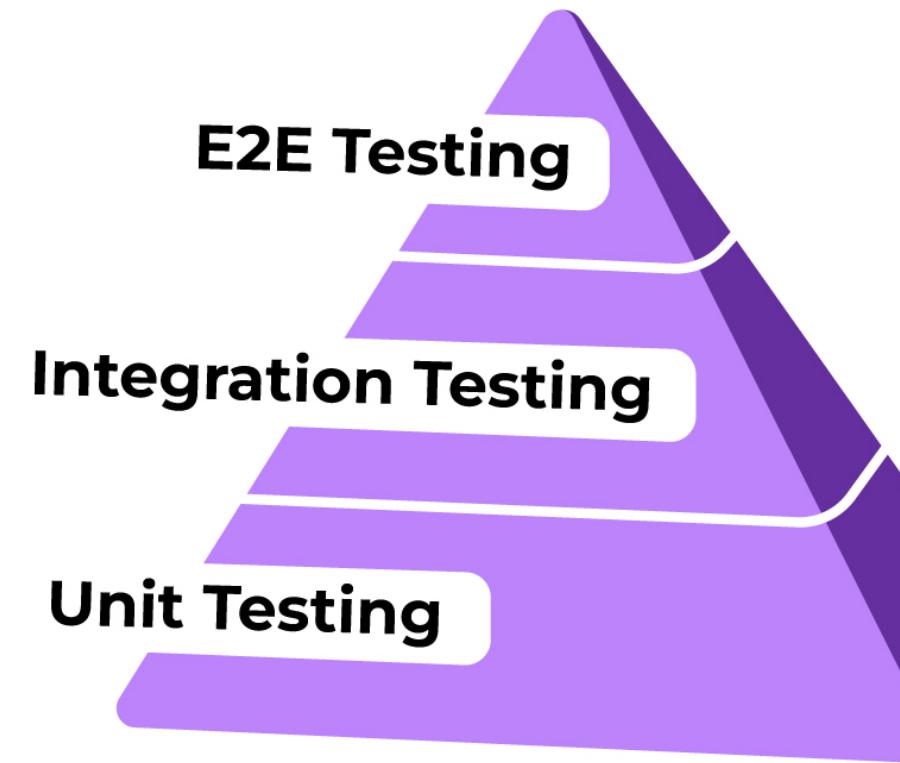
Testes contribuem para a melhoria contínua do software, garantindo que ele atenda aos requisitos especificados.

Economia de tempo e recursos no longo prazo

Identificar e corrigir erros antes de o software ser lançado pode poupar tempo e dinheiro a longo prazo.

O ciclo de vida dos testes





Piramide de Testes

Representa a distribuição ideal dos diferentes tipos de testes em um projeto. Dando ênfase maior em testes unitários e de integração, que resultam em um feedback rápido e detecção precoce de problemas. Os testes E2E, apesar de serem menos numerosos, são fundamentais para validar o sistema como um todo e garantir que ele atenda aos requisitos do usuário.

Ao seguir essa estratégia de testes, as equipes de desenvolvimento podem reduzir o tempo gasto na correção de erros mais graves e garantir que o software entregue alta qualidade, confiabilidade e funcionalidade.



Tipos de testes

Testes Unitários

São realizados em pequenas unidades do código, em geral funções ou métodos. São usados para garantir que o código funcione de acordo com as especificações.

Testes de Integração

São realizados para garantir que as diferentes unidades do software funcionem bem juntas, após os testes unitários terem sido realizados.

Testes E-2-E

Estes testes visam garantir que o software esteja funcionando de acordo com as especificações do projeto e as necessidades do usuário.

Anatomia de um Teste Unitário

```
import unittest

class TestNomeDaClasse(unittest.TestCase):
    def setUp(self):
        # Configurações iniciais (opcional)

    def test_nome_do_metodo_de_teste(self):
        # Preparação do ambiente de teste
        # Chamada do método ou função a ser testada
        # Asserções para verificar o resultado esperado

    def tearDown(self):
        # Limpeza após o teste (opcional)

if __name__ == '__main__':
    unittest.main()
```

Boas práticas de testes Unitários

1

Nova funcionalidade, novos testes

Adicione testes unitários a cada nova funcionalidade do seu projeto. Eles ajudam a garantir que as mudanças não quebrem outras partes do software.

2

Crie testes simples

Crie testes curtos e específicos para garantir que eles validem exatamente o comportamento esperado para a função ou classe sendo testada.

3

Execute testes com frequência

Execute seus testes unitários com frequência para garantir que todas as unidades do código funcionem corretamente e identificar problemas o mais cedo possível.



Made with Gamma

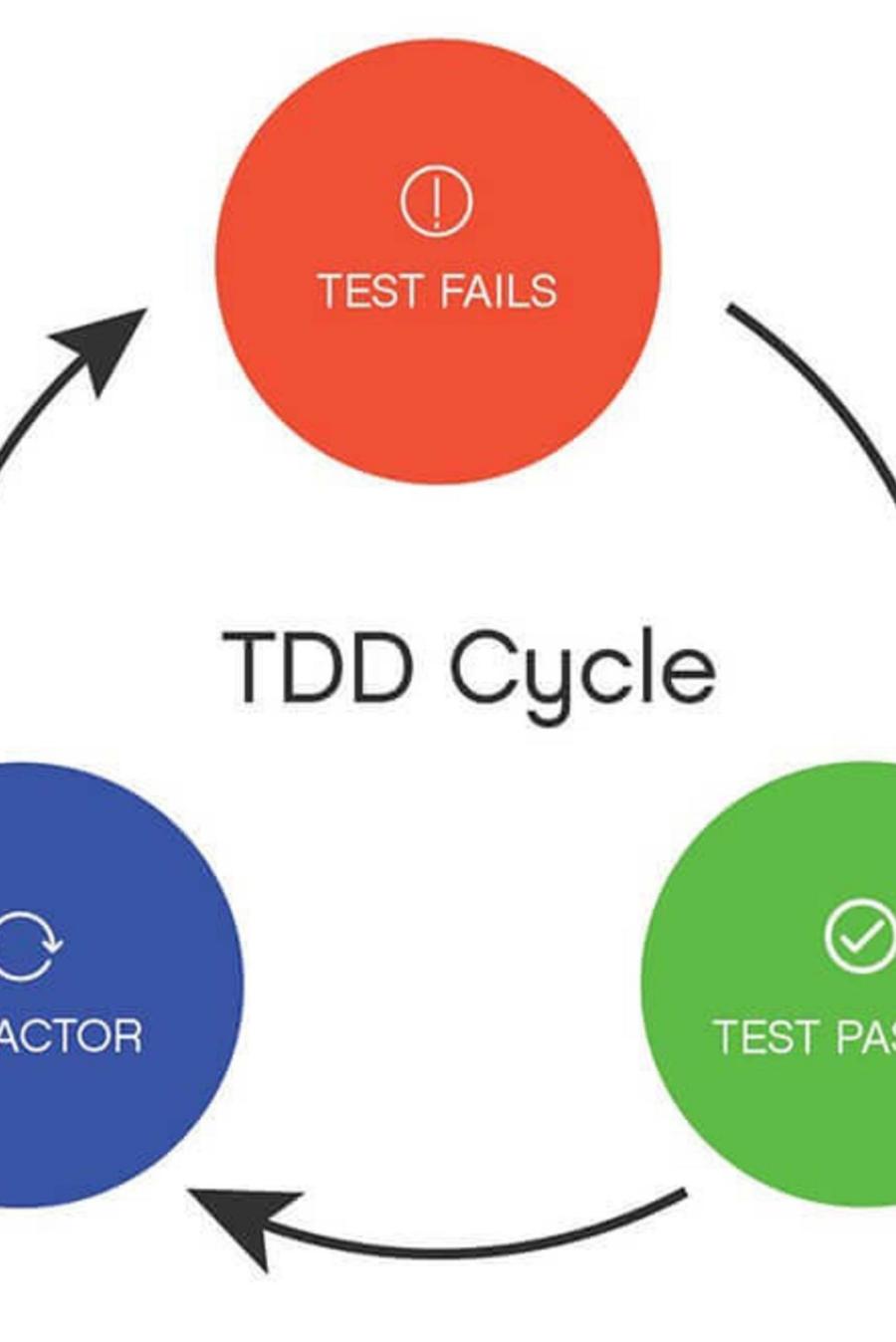
Ferramenta de Test

Unittest

A biblioteca `unittest` é um módulo integrado ao Python que fornece um framework para escrever e executar testes unitários, gerando relatórios informando quantos passaram ou falharam.

Como funciona?

- Em `unittest`, os testes são organizados em classes que herdam de `unittest.TestCase`.
- Cada classe de teste contém métodos que são testes individuais.
- Você escreve testes na classe criando métodos cujos nomes começam com "test". Por exemplo, `test_somar`.
- Dentro dos métodos de teste, você usa métodos de assertão fornecidos pela `unittest.TestCase` para verificar o comportamento do código.
- Você executa os testes chamando `unittest.main()` no final do arquivo de teste



Test Driven Development

TDD

O TDD é uma metodologia de desenvolvimento de software que prioriza a realização dos testes antes mesmo da escrita do código.

Cenários de Teste

Descrição	Resultado Esperado
Calcular média com uma lista vazia	Quando a lista for vazia é retornado 0
Calcular média com apenas um número na lista	Quando houver apenas um número na lista o retorno deve ser o mesmo número listado.
Calcular média com apenas números positivos na lista	Quando houver apenas números positivos na lista o retorno deve ser positivo



Code Coverage x Test Coverage

- Code coverage é quanto do código está coberto por testes (happy path, unhappy path, etc)
- Test coverage é mais ligado aos diferentes tipos de testes que existem no projeto (Unit, integração, e-2-e)

E se não Testar?

Voo 447 da Air France (2009)

Um Airbus A330, caiu no Oceano Atlântico, matando todas as 228 pessoas a bordo. As investigações revelaram que as sondas de velocidade estavam congeladas e forneceram dados inconsistentes à aeronave. Isso confundiu os pilotos e levou a erros nas decisões de voo, resultando no acidente.

Sony PlayStation Network Outage (2011)

PSN da Sony sofreu uma interrupção prolongada devido a uma violação de segurança, expondo informações pessoais e financeiras de milhões de usuários. A falha resultou em um desligamento de mais de três semanas, prejuízos financeiros significativos para a Sony e uma grande violação de privacidade.

Knight Capital Group Trading Glitch (2012)

Uma falha de software no algoritmo de negociação da Knight Capital Group causou a execução errônea de ordens, resultando em uma série de negociações indesejadas. Em um período de apenas 45 minutos, a empresa acumulou uma perda de mais de US\$ 450 milhões.

