Conceitos gerais e motivação sobre testes de software

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Este texto apresenta noções introdutórias e as motivações que envolvem a disciplina de testes de software. Como tópicos principais desta unidade inclui-se as atividades do teste de software, a pirâmide de software e uma breve introdução sobre testes de unidades.

Introdução

Atualmente, os desenvolvedores de software têm procurado cada vez mais aplicar boas práticas no intuito de melhorar a qualidade do software em desenvolvimento. De maneira geral, as empresas têm ciência de que as atividades de testes de software podem contribuir para aumentar a confiança e a qualidade do software.

Este texto tem o objetivo de apresentar a disciplina de testes automatizados, que é uma técnica voltada para a melhoria da qualidade de software. As atividades envolvidas nesta disciplina consistem em criar programas que testam outros programas com o objetivo de revelar falhas.

Para isso, torna-se necessário aprender métodos, técnicas e ferramentas para realizar a atividade de testes de software fundamentados em práticas conhecidas da área de engenharia de software.

Por que testar?

O desenvolvimento de software é uma atividade complexa e está sujeita a diversos tipos de problemas, o que acaba resultando na construção de um produto de software diferente do que foi especificado (DELAMARO, MALDONADO, JINO, 2017).

É possível identificar diversas causas para estes problemas, contudo é importante entender que a maioria deles está relacionada ao fato de que os desenvolvedores, que são os analistas funcionais e os programadores de software, cometem erros. As atividades de levantamento de requisitos, análise, projeto e construção de software dependem muito da capacidade do analista de interpretar as necessidades do solicitante e da habilidade dos programadores de construir o software de acordo com o que foi especificado.

Para que tais erros sejam identificados antes que o software seja disponibilizado para a produção, existe uma série de atividades de verificação, validação e testes do software, que são utilizadas para assegurar que o software esteja em conformidade com a especificação. A verificação avalia se o software atende os requisitos funcionais e não funcionais que foram especificados. A validação avalia se o produto atende as expectativas do cliente. Os testes de software, que é o principal objetivo desta disciplina, examinam o comportamento do software por meio da sua execução (SOMMERVILLE, 2007).

A atividade de testes de software

Após a escrita de um trecho do código de software, é natural que se deseja saber se o código funciona corretamente, ou não. Existem três maneiras para descobrir isso:

- Testar automaticamente o código;
- Testar manualmente o código;

- Não testar o código e, depois, aguardar para ver se o cliente reclama do software.

É claro que esta última opção não é uma boa ideia. Os testes manuais, são testes executados por um profissional em qualidade de software (analista de teste ou engenheiro de testes). Testes manuais são aplicados em diversos projetos de desenvolvimentos de software, porém, são propensos a erros, pois dependem de recursos humanos, o que influencia no tempo e no custo, que aumentam conforme o tamanho do projeto. Além disso, é cansativo e inviável executar os testes manuais diversas vezes.

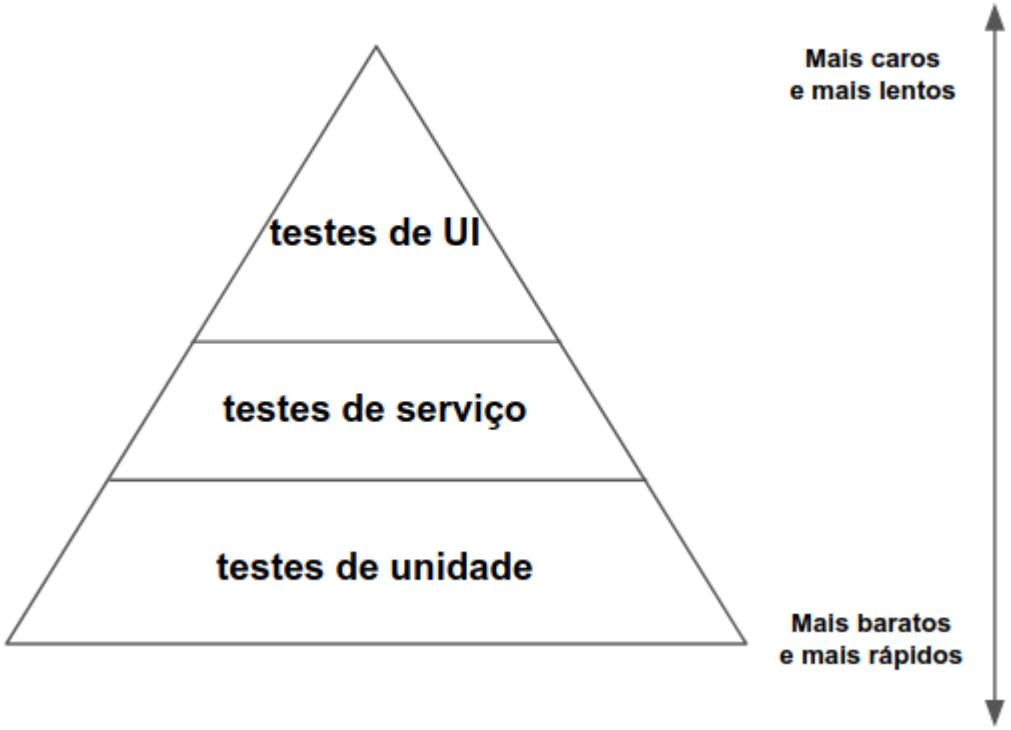
Portanto, a automação de testes de software torna-se uma opção viável para contribuir na tarefa de verificar a qualidade do código. Para isso, executa-se o código utilizando algumas entradas de dados escolhidas de acordo com algum tipo de critério e verifica-se o comportamento do código de acordo com o esperado. Caso a execução apresente valores diferentes daqueles que eram esperados, diz-se que houve um erro ou defeito. E nesta situação, é possível corrigir o erro e, assim, aumentar a confiança nas respostas obtidas pela execução do código.

É importante enfatizar que os testes automatizados são confiáveis, pois são scripts que executam testes sempre da mesma maneira. Cada script de teste indica um roteiro de uso das funcionalidades do software, isto é, uma documentação viva do software, o que é muito útil para aqueles programadores que estão iniciando no projeto. E por fim, os testes automatizados executam rotineiramente várias vezes ao dia. Caso alguma alteração modifique o comportamento do software, os testes que estavam funcionando, agora deixarão de funcionar. E, então, pode-se identificar mais rapidamente a presença de um bug de software causado por alguma modificação no código.

Pirâmides de testes

Segundo (COHN, 2019; FOWLER, 2012), antes da ascensão dos métodos ágeis como scrum, a automação de testes era considerada uma atividade custosa e poderiam demorar meses, ou até mesmo anos, para serem desenvolvidos após a escrita do código do sistema. Um dos possíveis motivos é que os times de desenvolvimento utilizavam estratégias erradas para o desenvolvimento de testes e, além disso, não tinham o hábito de construir os testes o mais cedo possível. Uma estratégia mais efetiva para construção de testes automatizados pode ser realizada em três diferentes níveis, como apresenta a Figura 1.1.

Figura 1.1. Pirâmide de testes de Cohn (adaptado)



Fonte: do autor, 2022.

Observe pela Figura 1.1 que a pirâmide de testes é dividida em três camadas, descritas como testes de unidade, testes de serviço e testes de UI (do inglês *User Interface*), os quais podem ser conceituados como:

- **Testes de unidade**: Também chamado de testes unitários, é a base da pirâmide e tem como objetivo testar a menor unidade do sistema,

por unidade entende-se componente, função, procedimento, ou até mesmo classe. O objetivo deste nível de teste é validar a lógica interna e estruturas de dados utilizados em cada módulo ou componente. Estes tipos de testes podem ser desenvolvidos em paralelo para diversos componentes.

- **Testes de UI**: O objetivo destes testes é exercitar o sistema de maneira que através da interface do usuário, os diversos campos de entrada sejam preenchidos, e após algum tipo de ação, os resultados obtidos desta ação sejam comparados com os resultados esperados.
- **Testes de serviço**: Um serviço é uma ação do sistema em reação a uma entrada ou a um conjunto de entradas. Os testes realizados neste nível servem para testar os serviços separadamente da interface do usuário. Portanto, ao invés de se aplicar diversos testes no nível de interface, adota-se aplicar testes no nível de serviço.

Esta estratégia de Cohn sugere concentrar maior esforço nos testes de unidade, que é a base da pirâmide, pois os testes de unidade são considerados fáceis de serem escritos, estáveis e rápidos de serem executados, enquanto que testes de UI são mais difíceis de se escrever, frágeis e são mais lentos ao se executar. Escrevendo mais testes de unidade, minimiza o número de testes nos níveis mais altos da pirâmide.

Ambiente de desenvolvimento

Antes de iniciar o processo de escrita dos testes, é uma boa ideia verificar algumas ferramentas que serão utilizadas para o desenvolvimento. Na sequência, segue alguns softwares necessários para o acompanhamento das unidades:

- Python: https://www.python.org/downloads

- Visual Studio Code: https://code.visualstudio.com/downloads

A linguagem de programação adotada para o desenvolvimento dos testes será o Python, no link acima é possível realizar o *download* da última versão da linguagem. Outra ferramenta é o Visual Studio Code, que é um editor de código que será utilizado nos exemplos e nos vídeos do curso. No entanto, não é essencial e nem obrigatório utilizar o Visual Studio Code, tendo a liberdade de utilizar qualquer outro editor de código de preferência.

- Pip: http://www.pip-installer.org

O Pip gerenciador de pacote do Python. O Pip permite a instalação de pacotes do Python a partir do repositório PyPi.

- Virtualenv: http://www.virtualenv.org

O virtualenv, como o próprio nome menciona, é um ambiente virtual. Neste espaço é possível instalar qualquer tipo de pacote sem precisar alterar as configurações de Python original. Neste sentido, o virtualenv é um ambiente que ajuda a deixar a instalação do python mais limpa para todo projeto. Para criar um ambiente virtual:

```
$ virtualenv venv
```

Após criar o ambiente virtual é necessário ativar o ambiente.

Mac OS/Linux

```
$ source venv/bin/activate
(venv)$
```

Windows

```
c:\> venv\Scripts\activate
(venv) c:\>
```

Para encerrar o ambiente virtual simplesmente utilize a instrução deactivate.

Teste de unidade

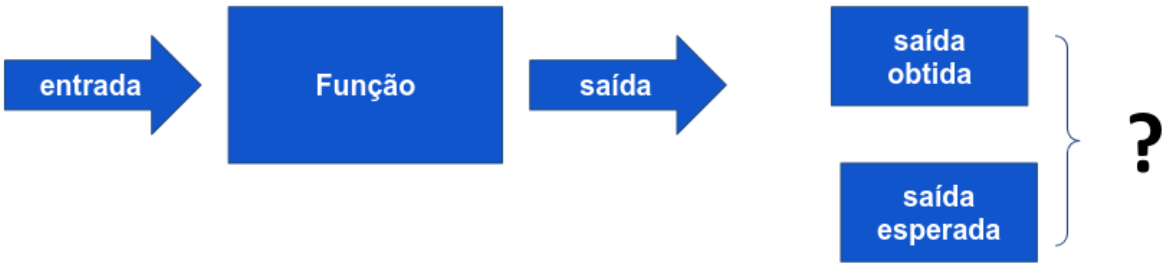
Seu objetivo é garantir que uma função, método ou classe funcione corretamente. Cada teste de unidade deve testar apenas uma funcionalidade específica.

Um teste de unidade possui três etapas:

- **Configuração**: planejamento do teste e definição dos valores de entrada e saída.
- **Chamada**: chamada da função a ser testada.
- **Afirmação**: comparação do retorno da função com o resultado esperado.

Para ilustrar a aplicação do teste de unidade, suponha uma função que receba uma determinada entrada de valores, e após o seu processamento ele devolva um resultado. O teste de unidade consiste em comparar essa saída obtida com uma saída esperada, que determina como a função deveria se comportar dado os valores de entrada. A Figura 1.2 apresenta as etapas do teste de unidade, como segue:

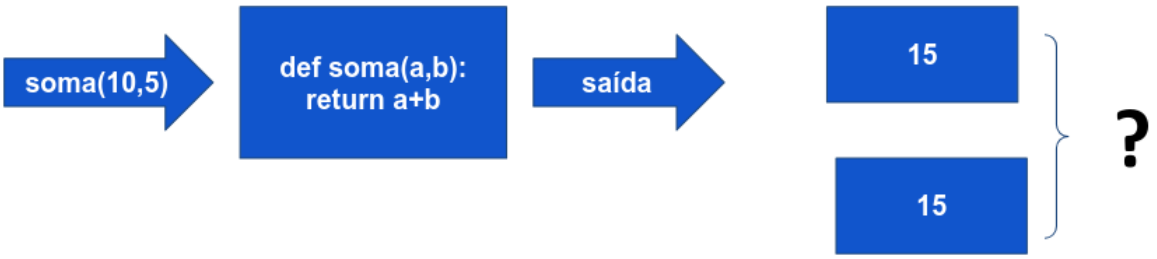
Figura 1.2. Etapas do teste de unidade



Fonte: do autor, 2022.

Suponha uma função chamada soma que receba como parâmetro duas variáveis a e b, e devolva como resultado o valor de a somado com o valor de b como apresenta a Figura 1.3.

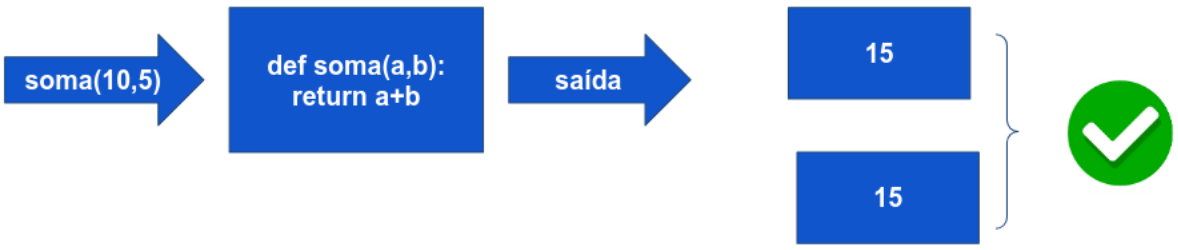
Figura 1.3. Teste de unidade da função soma



Fonte: do autor, 2022.

Ao submeter o teste com parâmetros a=10 e b=5, o resultado da função soma devolve como saída obtida o valor 15. Como o valor esperado da soma entre 10 e 5 é o valor 15, o teste deve indicar que a função devolve o valor correto. Ou seja, a função comporta-se como esperado para os valores 10 e 5. Nesta situação, o teste de unidade deve informar de alguma maneira que o teste da unidade ocorreu com sucesso, como apresenta a Figura 1.4 que é o valor esperado.

Figura 1.4. Teste de unidade da função soma com sucesso para a=10 e b=5

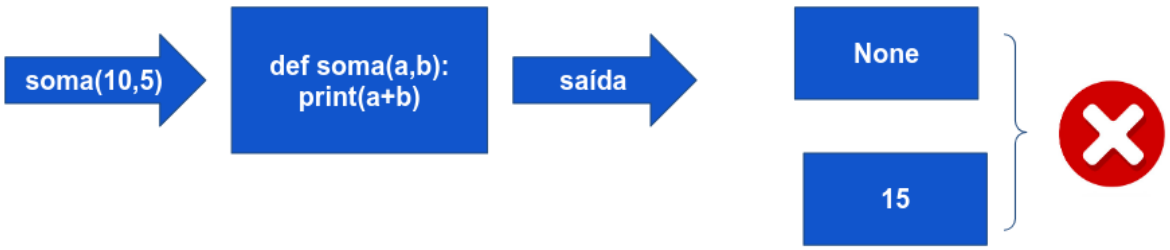


Fonte: do autor, 2022.

Normalmente, as ferramentas de testes de unidade executam vários casos de teste, para cada caso de testes, se os valores de entrada determinarem uma saída obtida igual a saída esperada então o caso de teste passou e a função executou como esperado. A Figura 1.4 apresenta o sinal de checagem positiva com a cor verde para o caso de teste em que o valor de a=10 e o valor de b=5.

Em outra situação, suponha que ao invés de a função soma devolver o valor de a somado com o valor de b, ela apenas utilize uma instrução para mostrar o resultado da soma, como apresenta a Figura 1.5.

Figura 1.5. Teste de unidade da função soma, com erro



Fonte: do autor, 2022.

Neste exemplo, como a função soma não devolve valor, o teste de unidade compara um valor da saída obtida igual a None com o valor esperado 15. Como os dois valores são diferentes, o Python deve indicar que o teste de unidade falhou. A falha no teste de unidade indica um erro na função sob teste e deve ser corrigida.

Teste de unidade sem framework

Esta seção apresenta uma maneira de realizar testes de unidade por meio de asserções, e utilizando a instrução assert. A instrução assert toma como entrada uma condição booleana e deve verificar:

- Se a condição for verdadeira, passa pela asserção e não faz mais nada.
- Se a condição for falsa, aborta a execução com uma exceção AssertionError.

O trecho da Codificação 1.1 apresenta um teste da função soma utilizando a instrução assert.

Codificação 1.1. Teste de unidade utilizando a instrução assert

```
def soma(a, b):
    return a + b
```

```
# Programa principal
try:
    resultado = soma(10, 20)
    assert resultado == 30
    print('Soma correta!')
except AssertionError:
    print('Soma errada!')
```

Fonte: do autor, 2022.

Perceba que o programa realiza um teste da função soma com parâmetros 10 e 20, e verifica se o resultado devolvido foi 30. Como a função devolve corretamente o resultado, o programa mostra a mensagem ‘Soma correta!’.

Agora, verifique o trecho da Codificação 1.2, que apresenta uma função semelhante ao da Codificação 1.1, porém a função soma apenas mostra o resultado e não devolve nenhum valor. Com isso o programa principal atribui um valor nulo (None) para a variável resultado, e realiza a asserção, a condição resultado==30 é uma expressão falsa, e portanto, gera uma exceção AssertionError. A mensagem de saída para este exemplo será ‘Soma errada!’. Porém, como a função soma imprime o resultado da soma, o programa também mostra o resultado da soma igual a 30.

Codificação 1.2. Teste de unidade utilizando a instrução assert com exceção

```
def soma(a, b):
    print(a + b)

# Programa principal
try:
    resultado = soma(10, 20)
    assert resultado == 30
```

```
        print('Soma correta!')
except AssertionError:
        print('Soma errada!')
```

Fonte: do autor, 2022.

Aplicando vários casos de testes

Para garantir a qualidade do código é importante realizar diversos casos de testes, conforme apresenta a Codificação 1.3.

Codificação 1.3. Realizando diversos casos de testes

```
def soma(a, b):
        return a + b

# Programa principal
# caso de teste 01
try:
        resultado = soma(10, 20)
        assert resultado == 30
        print('Soma correta!')
except AssertionError:
        print('Soma errada!')

# caso de teste 02
try:
        resultado = soma(3, 5)
        assert resultado == 8
        print('Soma correta!')
except AssertionError:
        print('Soma errada!')
```

```
# caso de teste 03
try:
        resultado = soma('Olá ', 'mundo!')
        assert resultado == 'Olá mundo!'
        print('Concatenação correta!')
except AssertionError:
        print('Concatenação errada!')


# caso de teste 04
try:
        resultado = soma(1.4, 3.2)
        assert resultado == 4.6
        print('Soma correta!')
except AssertionError:

        print('Soma errada!')
```

Fonte: do autor, 2022.

Como se pode observar, como a linguagem Python não faz verificação de tipos é possível realizar soma de números inteiros, reais ou até mesmo concatenação de strings.

Vamos praticar?

1) Importe o módulo abaixo para um programa de teste e escreva testes unitários para as funções do módulo:

```
# calcula o volume de uma caixa retangular
def calcula_volume(comprimento, largura, altura):
    return comprimento * largura * altura
```

Utilize os valores abaixo como parâmetros de entrada e saída da função:

`calcula_volume(comprimento, largura, altura)`	
Entrada	**Saída**

`comprimento = 1`
`largura = 1`
`altura = 1`
	`1`

`comprimento = 2`
`largura = 4`
`altura = 3`
	`24`

`comprimento = 5`
`largura = 5`
`altura = 2`
	`50`

2) Importe o módulo abaixo para um programa de teste e escreva testes unitários para as funções do módulo:

```
def converte_para_celsius(fahrenheit):
    celsius = (5.0/9.0) * (fahrenheit - 32)
    return celsius

def converte_para_fahrenheit(celsius):
    fahrenheit = 1.8 * celsius + 32

    return fahrenheit
```

Utilize os valores abaixo como parâmetros de entrada e saída de funções:

`converte_para_fahrenheit(celsius)`	
Entrada	**Saída**
`0`	`32.0`
`27`	`80.6`

`converte_para_celsius(fahrenheit)`	
Entrada	**Saída**
`32`	`0`
`41`	`5.0`

Referências

COHN, Mike. **The forgotten layer of the test automation pyramid**. 2009. Disponível em: <https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid>. Acesso em: 28 mai. 2022.

DELAMARO, M; MALDONADO, J; JINO, M. **Introdução ao teste de software**. Campus, 2007.

FOWLER, Martin. **Test pyramid.** Disponível em: <https://martinfowler.com/bliki/TestPyramid.html>. Acesso em: 28 mai. 2022.

SOMMERVILLE, Ian. **Engenharia de software**. 8. ed. São Paulo: Pearson AddisonWesley, 2007.