JÚLIO DE LIMA

TESTES DE SOFTWARE PARA INICIANTES

O livro dedicado a quem quer saber por onde começar na disciplina de testes

VOLUME 1

Sumário

SOBRE O AUTOR

Júlio de Lima possui experiência de mais de 10 anos em testes envolvendo aplicações Web, Desktop, Mobile e serviços. É formado em Tecnologia em Informática, especialista em Docência no Ensino Superior e é mestrando em Engenharia Elétrica e Computação com foco em Inteligência Artificial no Mackenzie com o intuito de solucionar problemas de teste com IA e ML. Atua como professor em cursos de pós graduação em universidades do Paraná, Recife e Manaus. Ajudou mais de 14 empresas a implementar testes e automação usando ferramentas privadas e open-source usando PHP, Java, Ruby e Javascript. É palestrante assíduo em diversos eventos de tecnologia, disseminando a disciplina de testes de software em mais de 30 eventos, além de ser o co-fundador do meetup garoaQA e professor de mais de 3.500 alunos em cursos na internet. Além disso é Youtuber, falando sobre testes de software todas as terças e quintas em seu canal "Júlio de Lima".

Redes sociais

facebook.com/juliodelimasoficial

twitter.com/juliodelimas

instagram.com/juliodelimasinsta

youtube.com/c/juliodelimas

linkedin.com/in/juliodelimas/

COMO ESSE LIVRO ESTÁ ORGANIZADO

Os tópicos do livro foram escolhidos baseando-se nas ações realizadas diariamente por pessoas que testam aplicações. Logo, após ler os 7 tópicos você irá adquirir a noção clara de como iniciar seus testes.

No *Tópico* 1 fazemos uma discussão sobre a mentalidade de alguém que testa aplicações baseando-se nos diversos contextos existentes atualmente. O *Tópico 2* demonstra como identificar quais testes fazer de maneira sistemática. Já o *Tópico 3*, mostra como é a abordagem empírica de testes. No *Tópico 4*, temos a prática de leitura de código e criação de testes com base nele. Já o *Tópico 5* traz a você, leitor, uma proposta de como gerir defeitos encontrados na aplicação. O *Tópico 6* apresenta conceitos de testes não funcionais. Finalmente, o Tópico 7 fala sobre automação de testes, seu valor, benefícios e seu maior segredo.

Termos em *italic* representam termos em outro idioma. A ênfase é dada a palavras através de formatação **em negrito**. Linhas de comando são representadas com `essa fonte`. Textos sublinhados serão apresentados quando houverem conteúdos complementares na web.

CAPÍTULO 1: **A MENTALIDADE DE ALGUÉM QUE TESTA APLICAÇÕES**

Há diversas definições de quais são os comportamentos e forma de pensar de alguém que testa aplicações, elas baseiam-se basicamente em uma explosão de curiosidade, criticismo e empatia com o cliente que receberá a aplicação. A pessoa que testa aplicações tem como um de seus objetivos principais, a antecipação de problemas que só seriam identificados pelos clientes, atrapalhando sua experiência na aplicação que desenvolvemos. Lembre-se que **alguém que testa uma aplicação pode ter diferentes clientes** (KANER et al., 2008), por exemplo: o usuário, o desenvolvedor, o *product owner,* um sistema terceiro que utilizará sua API, etc. Logo, há três características que predominam na mentalidade de quem testa:

1	2	3
Preciso validar que o produto **atenda às expectativas do meu cliente**	Tenho que **antecipar comportamentos inesperados** e alertar os riscos ao meu time	É minha missão **disseminar essa mentalidade** dentre meus amigos

Vejamos alguns exemplos de o que chamo de atender as expectativas do meu cliente:

I. Um site terceiro tem a expectativa de que as integrações estão funcionando;

II. Um usuário tem a expectativa de que o site seja de simples compreensão e que responda de forma instantânea durante a compra de produtos.

Em ambos exemplos, vemos que é necessário conhecer o que há por detrás das expectativas do cliente e exercitar o produto buscando entender se elas são atendidas. As expectativas podem ter diversos níveis de detalhe, por exemplo, o cliente poderia esperar que regras específicas fossem parte do sistema, como "apenas usuários VIP tem desconto de 15% nas compras".

Olhando pela perspectiva de antecipar comportamentos inesperados, poderíamos pensar em exercitar combinações de dados de entrada na aplicação para identificar como ela se comporta. Por exemplo, enquanto o teste que valida a expectativa do cliente que usuários VIP tenham desconto de 15% nas compras, eu exercito um cenário onde o usuário não é VIP, para entender como o produto se comporta. A identificação de cenários que buscam identificar comportamentos inesperados pode ocorrer a partir de duas abordagens: **empírica ou sistemática**. Na primeira, quem testa deve escolher o que testar baseando-se em sua experiência nesse ou em outros produtos semelhantes. Já a abordagem sistemática baseia-se em técnicas de teste pré-definidas em que, dado um regra, identifica-se o que deve-se testar.

Por fim, é essencial que a pessoa que testa seja capaz de **compartilhar seus conhecimentos em testes** com seus colegas de time para que todos possam testar de maneira apropriada. Dessa forma é possível alimentar a cultura de testes, trazendo mais qualidade aos produtos. Assista esse episódio onde o Júlio fala mais sobre a mentalidade de alguém que testa aplicação.

É muito importante destacar que a pessoa que testa, ao identificar uma inconsistência, seja capaz de **comunicar claramente o nível de risco relacionado a existência dessa inconsistência ou mesmo de corrigi-la**, caso haja *know-how* para executar tal atividade.

CAPÍTULO 2: **IDENTIFICANDO TESTES DE MANEIRA SISTEMÁTICA**

Sim, há uma forma de olhar para a definição das regras de um determinado software e extrair de lá quais testes precisam ser executados para validar que a aplicação foi construída e tem o comportamento esperado. Vejamos um exemplo simples:

Todo cliente VIP que faz compras que totalizam R$ 3.500 a R$ 4.200, incluído, recebem desconto de 15%. Clientes VIPs com compras superiores a R$ 4.200 recebem desconto de 20%. Os demais clientes com compras acima de R$ 4.200 recebem 5% de desconto.

A primeira ação antes de aplicar-se uma técnica sistemática é descobrir quais são os atributos de entrada contidos na regra descrita. Uma forma simples de fazer isso é pensar como um usuário e imaginar quais dados seriam informados por ele. Nesse caso, temos dois atributos, são eles:

I. **tipo de cliente:** *[VIP, Não VIP];*

II. **total das compras:** *[menor que R$ 3.500, maior ou igual que R$ 3.500 e menor ou igual que 4.200, e maior que R$ 4.200].*

O que acabamos de fazer para cada atributo de entrada foi identificar as classes de equivalência (GRAHAN et al., 2008), uma técnica de teste sistemática.

Uma vez que identificamos os atributos e suas classes, podemos usar a permutação, ou seja a combinação entre todas as classes de cada atributos, para identificar todos os testes a serem feitos.

Teste 1: VIP com total de compras menor que R$ 3.500 não deve receber desconto;

Teste 2: VIP com total de compras maior ou igual que R$ 3.500 e menor ou igual que 4.200 deve receber desconto de 15%;
Teste 3: VIP com total de compras maior que R$ 4.200 deve receber desconto de 20%;
Teste 4: Não VIP com total de compras menor que R$ 3.500 não deve receber desconto;
Teste 5: Não VIP com total de compras maior ou igual que R$ 3.500 e menor ou igual que 4.200 não deve receber desconto;
*Teste 6**:*** Não VIP com total de compras maior que R$ 4.200 deve receber desconto de 5%.

Percebemos algo muito interessante ao aplicar técnicas sistemáticas: a identificação de cenários que não estavam explícitos na regra de negócios, por exemplo, como vemos nos testes 1, 4 e 5. Não havia definição clara de qual é o resultado esperado para essas combinações de dados. Ao conversar com o responsável pela definição da regra, conseguimos a informação que **clientes desse tipo com esses valores totais de compra não deveriam receber desconto**.

Uma vez que todos os 6 testes são executados, conseguimos validar se o requisito funciona como esperado e antecipamos possíveis comportamentos inesperados. Por exemplo, digamos que nosso time não codificou corretamente o que deveria acontecer ao termos um cliente Não VIP, com valor acima de R$ 4.200, e que o desconto fornecido a essa combinação foi de 20%. Ao executar o Teste 6, saberíamos que a aplicação possui uma inconsistência e que esta precisa ser corrigida.

Há uma diversidade de técnicas de teste utilizadas por pessoas que testam aplicações, uma delas é a Tabela de Decisão, que você pode aprender através deste vídeo. A técnica apresentada nesse Tópico traz a você, leitor, insumos suficientes para começar a testar e aprofundar mais seus estudos e pesquisas.

CAPÍTULO 3: IDENTIFICAÇÃO DE TESTES DE MANEIRA EMPÍRICA

Identificar testes de maneira empírica significa dar liberdade ao conhecimento obtido no passado para exercitar testes no presente. Esse conhecimento geralmente baseia-se em:

I. Experiência adquirida em softwares semelhantes;

II. Defeitos identificados no passado;

III. Idéias de teste adquiridas com base na experiência de outrem.

A regra de negócio que desejo testar poderia ser a mesma que vimos no Capítulo 2, mas a forma de testar que leva a caminhos diferentes, por exemplo:

Todo cliente VIP que faz compras que totalizam R$ 3.500 a R$ 4.200, incluído, recebem desconto de 15%. Clientes VIPs com compras superiores a R$ 4.200 recebem desconto de 20%. Os demais clientes com compras acima de R$ 4.200 recebem 5% de desconto.

Poderíamos determinar uma série de testes complementares aos que foram elencados sistematicamente, baseando-se nos conhecimentos empíricos, como os que vemos abaixo:

- A influência do navegador no cálculo do desconto, dado que o Javascript usado no cálculo pode ser interpretado de diferentes maneiras dependendo do navegador;

- A variação de status do cadastro do usuário VIP antes e depois do cálculo do desconto, pois já ví usuários VIP com contas expiradas ou inativas ainda obterem benefícios como se suas contas estivessem ativas;

- A influência da redução ou aumento do valor total da compra e a acurácia do cálculo do desconto, dado que muitas vezes o aumento repentino do total da compra leva mais tempo para atualizar o desconto e esse acaba se perdendo.

A identificação do uso de uma abordagem empírica é, geralmente, complementar à abordagem sistemática e depende diretamente da experiência da pessoa que testa aplicações. Inclusive, testes exploratórios (HENDRICKSON et al., 2013), é o nome dado a atividade de explorar a aplicação buscando por comportamentos esperados e inesperados. Quando feito de maneira apropriada, traz uma série de benefícios.

CAPÍTULO 4: CRIAÇÃO DE TESTES COM BASE NO CÓDIGO DA APLICAÇÃO

O código fonte desenvolvido para construir a aplicação costuma passar, de forma direta ou indireta, por revisões de outras pessoas do time, com o intuito de verificar questões estáticas, como por exemplo, a forma com que os algoritmos são escritos. Esse tipo de verificação pode ser feita de forma manual ou através de ferramentas que o fazem de forma automatizada. De fato, cada uma dessas abordagens tem seus prós e contras: a verificação automática é rápida mas condicionada a validações prévias, já as manuais são lentas, mas provê feedback valiosos, visto que é feita por profissionais experientes.

Uma outra abordagem é a escrita de testes de unidade em código com o objetivo de exercitar os métodos contidos numa aplicação, de forma isolada, ou seja, sem fazer uso de classes ou recursos externos, com o objetivo de validar que o código desenvolvido funciona como esperado. Há várias formas de selecionar quais testes de unidade serão criados, a forma mais comum, conhecida como cobertura de sentença, visa exercitar todas as linhas contidas em um trecho de código.

```
1. function calcularDesconto (cliente, totalCompra) {
2.   if (totalCompra > 4200) {
3.      if (cliente && cliente.isVIP()) {
4.         return totalCompra * (20 / 100);
5.      } else {
6.        return totalCompra * (5 / 100);
7.      }
8.   }
9. }
```

Código-Fonte 1: *Método calcular desconto que dá desconto a alguns clientes dado o total da compra e o tipo de cliente.*

O ***Código-Fonte 1***, mostra parte do método utilizado para calcular o desconto do cliente, utilizaremos ele como base para identificação dos testes de unidade. Para exercitar a quantidade de código acima depende da utilização do método `calcularDesconto` com a variação dos dados de entrada do método com o objetivo de exercitar todas as linhas da métodos. Logo, o primeiro teste poderia ser aquele em que iremos exercitar as linhas 1, 2, 3, 4, 8 e 9.

Então devo pensar em como fazer para que seja possível exercitar essas linhas, vejamos:

Teste 1: Total da compra é 5000 e o cliente é VIP então o retorno é 20%, ou seja, 1000;

Esse teste exercita as linhas 1, 2, 3, 4, 8 e 9 linhas, então é necessário criar mais testes para exercitar as demais linhas. Para isso, o seguinte teste poderia ser escrito:

Teste 2: Total da compra é 5000 e o cliente não é VIP então o retorno é 5%, ou seja, 250;

Com o Teste 2 as linhas 5, 6 e 7, que ainda não possuíam testes para exercitá-las, agora são exercitadas, fazendo com que todas as 9 linhas sejam exercitadas sempre que os testes forem executados. O benefício de escrever testes como esse é que eles servem como um contrato entre quem escreveu os testes e o código escrito. Esses testes avisarão a todos caso alguém altere o método `calcularDesconto` fazendo com que ele deixe de comportar-se como esperado. Lembre-se, esses testes só tem valor agregado se forem executados, conforme o Júlio explica neste vídeo.

CAPÍTULO 5: **COMO GERIR INCONSISTÊNCIAS IDENTIFICADAS AO TESTAR**

Uma inconsistência é a disparidade entre o que era o comportamento esperado perante ao comportamento atual. Para a pessoa que a testa aplicações, identificar uma inconsistência deve significar a antecipação de problemas que acabariam sendo identificados pelos usuários do produto que desenvolvemos.

Em times tradicionais, a pessoa que testa a aplicação costuma ser diferente da pessoa que corrige uma inconsistência, **já em times mais inovadores e maduros, qualquer pessoa pode corrigir uma inconsistência, inclusive quem a identificou**. Em ambas é importante registrar a inconsistência, adicionando informações como as descritas na norma IEEE 829-1998, como um **título auto-descritivo, o comportamento esperado e um passo a passo para reprodução da inconsistência, os dados de entrada utilizados, o ambiente utilizado, a versão da aplicação, etc**. Essas informações irão auxiliar a pessoa que corrige a alimentar métricas que poderão ser utilizadas pelo time para entender as áreas em que eles podem melhorar de forma contínua.

Há diversas propostas de ciclo de vida de uma inconsistência, vemos a seguir um exemplo: **Inconsistência aberta, Inconsistência em análise, Inconsistência invalidada, Inconsistência sendo corrigida, Inconsistência sendo testada e Inconsistência corrigida**.

Quanto mais clara é a descrição da inconsistência, mais fácil fica sua compreensão e assim, menos vezes uma inconsistência tramita no processo para frente e para trás. Lembrem-se, precisamos descrever não só o comportamento inapropriado, mas também descrever claramente o risco atrelado a não correção da inconsistência.

CAPÍTULO 6: **TESTES NÃO-FUNCIONAIS**

Testes não-funcionais são testes que buscam avaliar como a aplicação, que já atende às expectativas funcionais (ou seja, que foi testada e mostrou-se funcionar como esperado), foi estruturada e/ou como ela comporta-se perante ao ambiente no qual encontra-se. Características de qualidade descritas pela norma ISO 9126-1 ajudam a direcionar testes não funcionais, dentre essas características podemos ver a **Confiabilidade, a Usabilidade, a Eficiência, a Manutenibilidade e a Portabilidade**.

Os testes relacionados abaixo tem o objetivo de avaliar cada uma dessas características sob uma das diversas formas de avaliar cada uma de suas sub-características de qualidade:

***Teste 1*:** Buscando atender à sub-característica de Recuperabilidade, umas das sub-características de qualidade da Confiabilidade, validaremos se em os dados parciais que estavam sendo informados durante a transação foram preservados mesmo após uma falha na infraestrutura;

Teste 2: Buscando atender à sub-característica de Apreensibilidade, umas das sub-características de qualidade da Usabilidade, definiremos tarefas que os usuários deverão executar na aplicação e mediremos a velocidade da execução e o feedback quanto a facilidade em operá-la (NIELSEN, 2019);

Teste 3: Buscando atender à sub-característica de Tempo, umas das sub-características de qualidade da Eficiência, simularemos a utilização da aplicação uma quantidade elevada de usuários simultâneos acessando a aplicação e o tempo de resposta à efetuação da compra não deverá levar mais que 300 milissegundos;

Teste 4: Buscando atender à sub-característica de Testabilidade, umas das sub-características de qualidade da Manutenibilidade, verificaremos se todos os métodos da aplicação possuem baixo acoplamento e alta coesão, pois essas são características que permitem a escrita de testes de unidade;

Teste 5: Buscando atender à sub-característica de Adaptabilidade, umas das sub-características de qualidade da Portabilidade, a aplicação web deve poder ser executa em todos os 5 navegadores utilizados pelos nossos usuários sem que haja a necessidade de realizar mudanças no código-fonte da aplicação.

Há muito mais formas de avaliar aplicações sob a perspectiva não-funcional, inclusive através do uso de ferramentas de teste voltadas especificamente a execução e suporte de testes como os descritos nesse tópico, por exemplo, o JMeter, para testes de performance, que o explicada passo a passo pelo Júlio neste vídeo. Quanto aos resultados esperados, podem ser números inteiros ou percentuais de aceitabilidade (por exemplo, 25% é ruim, 50% é aceitável, 75% é bom e 99% é ótimo).

CAPÍTULO 7: **USO E BENEFÍCIOS DA AUTOMAÇÃO DE TESTES**

Automatizar testes é uma habilidade que toda pessoa que testa aplicações precisa ter para que possa colaborar para **o aumento da cobertura de testes, a redução de tarefas repetitivas como re-teste e ações de depuração de código e a confiança do time de desenvolvimento em fazer mudanças mo código-fonte**. Basicamente, a automação de testes é a construção de scripts capazes de repetir as ações de um ser humano frente a um produto de software seguido da validação dos resultados pós ação de forma automatizada. Pode-se fazer automação e testes em diversas camadas do software: **métodos ou funções contidas no código-fonte, serviços como Rest e Soap, componentes da interface gráfica, a própria interface gráfica Web, Mobile ou Desktop, entre outros**.

Há uma variedade gigantesca de ferramentas voltadas a automação de testes para cada uma das camadas citadas no parágrafo anterior que te ajudam a não ter que executar todos os seus testes manualmente todas as vezes que a aplicação sofre uma manutenção em seu código. Podem ser comerciais (é necessário pagar pela licença de uso) ou de código aberto. Algumas das ferramentas ou frameworks de testes automatizados mais conhecidas, são explicadas pelo Júlio nesta lista de tutoriais práticos.

No passado, a atividade de automação de testes era executada por profissionais que eram dedicados a criação e manutenção de scripts de teste automatizados. **Hoje em dia, essa atividade precisa ser compartilhada entre todos os profissionais do time que conhecem codificação**, visto que todo o time é beneficiado pela existência dos testes e que mudanças no código-fonte podem fazer com que testes automatizados deixem de funcionar. Logo, todos precisam ser capazes de criar e dar manutenção nos testes.

A automação dos testes começa a fornecer valor ao time de desenvolvimento quando passa a ser executada automaticamente após o código da aplicação sofrer alterações. Ou seja, todas as vezes que alguém que fizer uma mudança na aplicação buscando evoluí-la ou mesmo corrigir algum comportamento inapropriado, os testes automatizados existentes em todas as camadas precisam ser executados buscando avaliar se a alteração fez com que algo que já funcionava, viesse a parar de funcionar. Se esse for o caso, a alteração feita na aplicação deve ser ignorada. Outra possibilidade é que a alteração no código fez com que o teste automatizado viesse a parar de funcionar trouxe uma nova funcionalidade ainda não conhecida. Nesse caso, o teste automatizado deve ser alterado visando adaptar-se à nova funcionalidade.

```
1. WebDriver navegador = new ChromeDriver();
2. navegador
        .get("http://www.cadastrarcontato.com.br");
3. navegador
        .findElement(by.id("campoTelefone"))
        .sendKeys("1199997777");
4. navegador
        .findElement(by.id("botaoSalvar"))
        .click();
5. String mensagem = navegador
        .findElement(by.id("mensagem"))
        .getText();
6. assertEquals("Sucesso ao cadastrar o contato!", mensagem);
```

Código-Fonte 2: *Cadastrando um contato com uso de Selenium WebDriver em Java.*

O ***Código-Fonte 2*** apresenta um exemplo de parte do código de um teste automatizado escrito com Selenium WebDriver, uma biblioteca de código aberto, usada para automação de testes em aplicações web, que acessa uma página, digita um texto em um campo e clica num botão de submissão do formulário. Em um de seus episódios, o Júlio explica passo a passo como automatizar seu primeiro teste usando WebDriver.

REFERÊNCIAS

- Kaner, Cem, James Bach, and Bret Pettichord. *Lessons learned in software testing*. John Wiley & Sons, 2008.

- Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans. Foundations of software testing: ISTQB certification. Cengage Learning EMEA, 2008.

- Hendrickson, Elisabeth. Explore it!: reduce risk and increase confidence with exploratory testing. Pragmatic Bookshelf, 2013.

- IEEE Standards Association. 829-1998 IEEE Standard for Software Test Documentation. Technical report, 1998.

- Associação Brasileira de Normas Técnicas. NBR ISO/IEC 9126-1 Engenharia de software: Qualidade de produto. Rio de Janeiro; 2003.

- Nielsen, J.: Usability Metrics (2001). https://www.nngroup.com/articles/usability-metrics/. 24 August 2019

AGRADECIMENTOS

Aos revisores Priscila Alves ❤, Rafael William, Rafael Sousa, Jonathan Santos, Jônatas Kirsch e Maira Consenso.

JL.