Implementação de Testes de Integração nos Frameworks de Desenvolvimento Mobile Flutter e React Native: um estudo comparativo

Kelvia Kolln[1], **José Carlos Toniazzo**[1], **Viviane Duarte Bonfim**[1]

[1]Universidade Comunitária Regional de Chapecó (UNOCHAPECÓ), Chapecó, SC, Brasil

kkolln@unochapeco.edu.br, zetoniazzo@unochapeco.edu.br, vividb@unochapeco.edu.br

Abstract. *This study compares the implementation of integration testing in mobile development frameworks Flutter and React Native, using the GQM (Goal-Question-Metric) paradigm to evaluate metrics such as code volume, resource consumption, execution time, and robustness. Through a case study of a basic registration system, was been analyzed ninety tests in Flutter and forty seven in React Native, measuring CPU and RAM usage, lines of code, and performance under load testing. Results indicate that React Native outperforms Flutter in execution time, while both frameworks show similar resource consumption and ease of use (mocks, documentation). However, React Native exhibited more failures during simultaneous load tests. The choice between frameworks should consider developer language proficiency, customization needs (Flutter), and runtime efficiency (React Native). This work provides practical insights for mobile developers selecting integration testing strategies in cross-platform environments.*

Keywords*: Integration Testing, Flutter, React Native, GQM, Mobile*

Resumo. *Este estudo compara a implementação de testes de integração nos frameworks de desenvolvimento mobile Flutter e React Native, utilizando o paradigma GQM (Goal-Question-Metric) para avaliar métricas como volume de código, consumo de recursos, tempo de execução e robustez. Por meio de um estudo de caso com um sistema de cadastro básico, foram analisados noventa testes no Flutter e quarenta e sete no React Native, medindo consumo de CPU e memória RAM, linhas de código e desempenho em testes de carga. Os resultados indicam que o React Native supera o Flutter em tempo de execução, mas ambos os frameworks apresentam consumo de recursos e facilidade de uso (mocks, documentação) similares. Verificou-se também que o React Native exibiu mais falhas em testes de carga simultâneos. A escolha entre os frameworks deve considerar a experiência do desenvolvedor com a linguagem, necessidades de customização (Flutter) e eficiência em tempo de execução (React Native). Este trabalho oferece esclarecimentos práticos para desenvolvedores na seleção de estratégias de teste de integração em ambientes mobile multiplataforma.*

Palavras-Chave*: Testes de Integração, Flutter, React Native, GQM, Mobile*

1. Introdução

O lançamento do iPhone em 2007 impulsionou a demanda por aplicativos móveis e soluções personalizadas [Rodriguez 2019]. Frameworks multiplataforma como *Flutter* e *React Native* se destacaram por permitir o desenvolvimento para iOS e Android

com um único código, otimizando tempo e recursos [Pignati 2021]. Contudo, a complexidade desses sistemas exige testes de integração para garantir a comunicação entre módulos e evitar falhas que afetem a experiência do usuário [Pressman and Maxim 2021, Lewis and Veerapillai 2004]. A automação desses testes é essencial, embora enfrente desafios como fragmentação de dispositivos e custos.

Este estudo compara a implementação de testes de integração em *Flutter* e *React Native* pelo paradigma GQM (*Goal Question Metric*), avaliando métricas como desempenho, facilidade de uso e consumo de recursos. A pesquisa, baseada em estudo de caso, analisa critérios como quantidade de código, dependências, tempo de execução e robustez em testes de carga, apresentando boas práticas, limitações e *insights* para auxiliar desenvolvedores na escolha da tecnologia mais adequada.

2. Trabalhos Relacionados

A comparação entre *Flutter* e *React Native* é um tópico recorrente na literatura de desenvolvimento móvel, porém com foco voltado para aspectos de desempenho em tempo de execução, experiência do usuário ou comparações de features. O estudo de [Kaur and Kaur 2022], por exemplo, realizou uma revisão sistemática da literatura sobre estimativas de testes de software no desenvolvimento de aplicações mobile.

Trabalhos como o de [Zahra and Zein 2022] exploraram estratégias de teste em *React Native* e *Flutter*, demonstrando uma similaridade grande entre ambos, o que também foi constatado neste estudo, com ênfase para testes de integração.

No entanto, há uma lacuna de pesquisa quanto a estudos comparativos que utilizem um paradigma estruturado de medição, como o GQM, para avaliar a implementação de testes de integração. Este artigo se diferencia ao preencher essa lacuna, aplicando o paradigma GQM para uma comparação baseada em métricas como quantidade de código, consumo de CPU e memória, e resiliência em testes de carga, contribuindo com evidências para auxiliar na escolha entre *Flutter* e *React Native* do ponto de vista de testes de integração, cruciais para o bom uso de aplicações que demandam sincronização de dados com ambientes online.

3. Fundamentação Teórica

3.1. Testes de Software

Segundo [Sommerville 2021], os testes visam comprovar que o software atende aos requisitos e identificar defeitos antes do uso, por meio da execução com dados sintéticos para detectar erros, anomalias ou problemas não funcionais. O processo busca tanto validar o cumprimento dos requisitos quanto encontrar entradas que provoquem comportamentos incorretos. A atividade envolve casos de teste, que definem condições, entradas e resultados esperados [Craig and Jaskiel 2002, Mili and Tchier 2015]; procedimentos de teste, que descrevem passos de execução [Craig and Jaskiel 2002, Graham et al. 2008]; e critérios de teste, que orientam a seleção e avaliação dos casos, como cobertura, adequação [Rocha et al. 2001, Desikan and Ramesh 2006] e geração de casos [Rocha et al. 2001, Ammann and Offutt 2017]. Conforme o IEEE (*Institute of Electrical and Electronic Engineers*), defeito é uma implementação incorreta; erro é sua manifestação no produto; e falha é o comportamento divergente do esperado. Defeitos,

resultantes de falhas humanas, geram erros que podem levar a falhas e comprometer o uso do software [Neto 2009].

3.2. Testes Manuais e Automatizados

Segundo [Carvalho 2022], os testes podem ser manuais, executados sem automação e seguindo etapas documentadas para identificar falhas, ou automatizados, que utilizam ferramentas para agilizar verificações e garantir conformidade. Os manuais têm menor custo, mas são menos precisos por dependerem do fator humano; já os automatizados, ideais para testes repetitivos e prolongados, exigem maior investimento e tempo de implementação. Os testes manuais são sujeitos a erros humanos, permitem a análise humana e são mais práticos em casos pouco repetitivos. Também são indicados para testes de usabilidade. Já os testes automatizados apresentam maior precisão por utilizarem ferramentas, sendo adequados para cenários específicos. Eles são ideais para testes repetitivos e de longa duração, além de indicados para testes de estresse e carga.

3.3. Técnicas de Testes de Software

Segundo [Myers et al. 2011], as técnicas de teste variam conforme a fonte de informação para definir requisitos, sendo recomendada a combinação de abordagens. O teste de caixa preta avalia entradas e saídas sem considerar a estrutura interna, verificando se funcionalidades e componentes operam corretamente. Já o teste de caixa branca examina a lógica e a estrutura do código, identificando falhas e garantindo cobertura de todas as partes do sistema.

3.4. Verificação e Validação

Segundo [Salomão 2016], a verificação assegura que o software seja desenvolvido corretamente, atendendo às especificações funcionais e não funcionais, enquanto a validação confirma se o produto final cumpre as expectativas do cliente [Souza and Gasparotto 2013]. Esses processos, complementares e contínuos ao longo do ciclo de vida, aumentam a confiabilidade do sistema. O modelo em V (Figura 1) é amplamente adotado por evidenciar a importância dessas atividades na detecção de defeitos e mitigação de riscos.

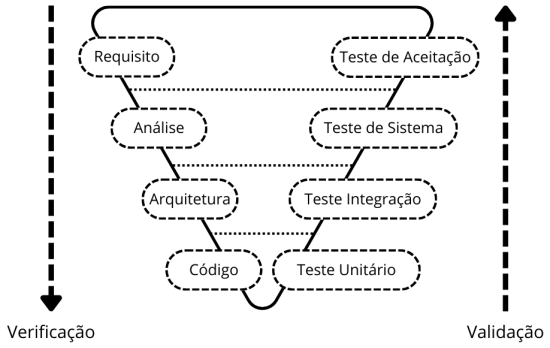

Figura 1. Fases do Desenvolvimento x Fases de Testes)

Conforme exposto na Figura 1, o planejamento de testes ocorre em diferentes níveis:

- Teste unitário: avalia módulos isolados, identificando falhas lógicas e de implementação [Neto 2009, Santos 2016, Bernardo and Kon 2008].
- Teste de integração: verifica falhas nas interfaces entre módulos (Dias Neto, 2015; Luft, 2012; Santos, 2016).
- Teste de sistema: checa se o software atende aos requisitos em uso real [Neto 2009, Luft 2012, Santos 2016].
- Teste de aceitação: realizado por usuários finais para validar o funcionamento [Neto 2009].
- Teste de regressão: garante que alterações não introduzam problemas, reaplicando testes anteriores [Neto 2009, Salomão 2016].

3.5. Testes de Integração

Segundo [Pressman and Maxim 2021], o teste de integração avalia a interação entre componentes, combinando técnicas de caixa-branca e caixa-preta para garantir que módulos individuais funcionem corretamente quando integrados. A automação desse processo aumenta a eficiência, permitindo execuções rápidas e repetitivas que reduzem custos e aceleram o desenvolvimento.

Segundo [Beizer 1995] os métodos de integração podem seguir diferentes abordagens: no *Bottom-up*, os testes começam pelos módulos de baixo nível e avançam para os superiores, utilizando drivers, sendo indicado para sistemas grandes, mas com visibilidade mais lenta do todo. No *Top-down*, parte-se dos módulos de alto nível para os inferiores, com uso de stubs, possibilitando detectar falhas de design precocemente. Já o *Big Bang* integra todos os módulos de uma vez, permitindo avaliar o sistema completo, mas dificultando o isolamento de erros.

3.6. Paradigma GQM

O paradigma GQM (*Goal Question Metric*), criado por Victor Basili, estrutura a medição na engenharia de software em três níveis: objetivo, questão e métrica. Conforme [Hussain and Kutar 2009], essa abordagem parte do princípio que a coleta de dados deve ser guiada por metas claras e fundamentadas logicamente, garantindo relevância nas medições. O processo segue as etapas:

1. Definir metas.
2. Formular questões para avaliar cada meta.
3. Definir métricas para coleta de dados.

As metas GQM são estruturadas no formato: "Analisar o <*objeto de estudo*> com a finalidade de <*objetivo*> com respeito ao <*enfoque*> do ponto de vista <*ponto de vista*> no seguinte contexto <*contexto*>."

- Objeto de estudo: o que será analisado (ex.: processo, projeto, sistema).
- Objetivo: finalidade da análise (ex.: avaliar, melhorar).
- Enfoque: atributo medido (ex.: confiabilidade, custo).
- Ponto de vista: usuário das métricas (ex.: desenvolvedores, gerentes).
- Contexto: ambiente da medição (ex.: projeto, departamento).

	Flutter	React Native
Bibliotecas	flutter_test (unitário e widget) e integration_test (integração)	Jest (unitário, componente e integração)
Confiabilidade, Velocidade e Custo	Sem diferenças significativas entre as plataformas	Sem diferenças significativas entre as plataformas
Funcionalidades	Testes de integração com simulação de interações e validação entre componentes	Suporte amplo em JavaScript, incluindo mocks; Flutter requer bibliotecas extras para mocks (ex.: Mockito)

Tabela 1. Comparativo entre Flutter e React Native na implementação de testes

4. Flutter e React Native e Suas Boas Práticas de Teste de Integração

4.1. Flutter e React Native

Flutter, criado pelo Google em 2015, é um SDK que permite desenvolver apps nativos para iOS e Android a partir de um único código compilado diretamente para a linguagem do dispositivo, garantindo alto desempenho e acesso a recursos nativos [Andrade 2020]. Baseado em widgets personalizáveis, o *Flutter* controla a interface sem depender de componentes nativos, simplificando o desenvolvimento. Segundo o *Stack Overflow*, 68,8% dos desenvolvedores preferem *Flutter*, usado por empresas como *Nubank* e *eBay*.

React Native, criado pela Meta também em 2015, usa *React JavaScript* para desenvolvimento multiplataforma, permitindo acesso a APIs nativas via "*Native Bridge*" que invoca componentes nativos para renderização [Charles 2023, Marques 2022]. Diferente do *React web*, ele utiliza componentes nativos da UI do dispositivo. Seu modelo de arquitetura inclui uma máquina virtual *JavaScript* com compilador JIT e execução da lógica em thread separada, garantindo interface fluida e desempenho. Segundo o *Stack Overflow*, 57,9% dos desenvolvedores optam por React Native, utilizado por *Facebook*, *Instagram* e *Discord*.

4.2. Boas Práticas e Ferramentas

Em *Flutter*, a organização clara de arquivos e pastas é fundamental para facilitar a manutenção dos testes, que devem ser isolados para evitar interferências. Interações do usuário são simuladas com funções como *tap*, *scroll* e *enterText*, e é importante usar *pumpAndSettle* para aguardar animações antes de validar com expect. A biblioteca *Mockito* é recomendada para mocks, desde que modelados adequadamente para maior realismo.

No *React Native*, a estruturação clara e a configuração do ambiente com *Jest* são igualmente importantes. Boas práticas incluem isolamento dos testes, ampla cobertura e adaptação contínua ao código. Recomenda-se criar componentes puros para minimizar estados locais e evitar re-renderizações desnecessárias. Para mocks, utilizam-se ferramentas como *react-native-testing-library* e *jest.mock()* para simular dependências e facilitar testes eficazes.

5. Metodologia

Este trabalho utilizou o paradigma GQM para definir as métricas aplicadas na comparação da implementação dos testes de integração nos frameworks *Flutter* e *React Native*. Foram

levantadas questões que orientaram a definição do objeto de estudo, do objetivo e dos demais atributos do método. A partir dessas questões, foram estabelecidas as métricas utilizadas para conduzir o estudo comparativo entre *Flutter* e *React Native*. Para este artigo foram escolhidas as métricas mais relevantes que contribuíram para os resultados do estudo.

O estudo define seis metas para avaliar os testes: a Meta 1 analisa a quantidade de código necessária, medida em linhas (Questão 1.1/Métrica 1.1); a Meta 2 avalia o consumo de CPU e memória, considerando a média desses recursos (Questão 2.1/Métrica 2.1); a Meta 3 verifica a viabilidade de testes de estresse e carga, medindo as falhas em requisições (Questão 3.1/Métrica 3.1); a Meta 4 trata da disponibilidade de documentação, medida pela facilidade de acesso (Questão 4.1/Métrica 4.1); a Meta 5 avalia o uso de mocks para simulação de dados, considerando sua facilidade (Questão 5.1/Métrica 5.1); e a Meta 6 analisa a quantidade de dependências necessárias, medida pelo número utilizado (Questão 6.1/Métrica 6.1).

5.1. Descrição geral do Estudo de caso

5.2. Requisitos

Requisitos Funcionais (RF): O sistema deve permitir que o usuário faça login (RF1); gerencie clientes (RF2); produtos (RF3); fornecedores (RF4) e pedidos (RF5), suportando para cada entidade as operações de cadastro, edição, listagem (busca geral ou específica) e exclusão. Além disso, o sistema deve comunicar-se com o banco de dados por meio de uma API (RF6).

Requisitos Não Funcionais (RNF): O sistema precisa ser compatível com Android e iOS (RNF1); funcionar em qualquer resolução de dispositivo móvel (RNF2) e suportar todas as versões dos sistemas operacionais contemplados (RNF3).

O modelo de domínio inclui o usuário, que gerencia os dados; o fornecedor, responsável pelo fornecimento dos produtos; o produto, disponível para compra; o cliente, que realiza as compras; o pedido, registro da compra feita pelo cliente com auxílio do usuário; e o itemPedido, que detalha os produtos de cada pedido. Um usuário e um cliente podem ter vários pedidos, cada pedido está associado a ambos e contém vários itens vinculados a produtos. A classe fornecedor não possui relações diretas com as outras entidades.

5.3. Metodologia de coleta de dados e comparação

Para a coleta dos dados, foram implementados testes em *Flutter* e *React Native*, registrando métricas como número de dependências, linhas de código, tempo de criação dos testes, confiabilidade, facilidade no uso de mocks e disponibilidade de documentação e exemplos. Após o desenvolvimento, avaliou-se também o consumo de CPU e RAM durante a execução, além da cobertura dos testes planejados. Foi utilizado para a realização dos teste um notbook Dell com um processador intel i7, com 8GB de memória RAM

O tempo de desenvolvimento foi medido com cronômetro, e as linhas de código contadas no *Visual Studio Code*. Um teste de estresse com 20 inserções e buscas simultâneas em tabelas do banco (cliente, produto, pedido e fornecedor) avaliou a robustez do sistema. A facilidade em encontrar documentação foi avaliada via pesquisas online, considerando fontes como o projeto **clean-flutter-app** para *Flutter* e a documentação do

Jest para *React Native*. Simplicidade no uso de mocks, quantidade de dependências e organização do projeto também foram analisadas para avaliar manutenção e usabilidade.

6. Resultados do estudo de caso

O estudo de caso definiu que ambos os projetos incluíssem testes para validação de campos na criação e login de clientes, pedidos, produtos e fornecedores; operações de adição, edição, exclusão e listagem; considerando respostas da API nos códigos 200, 400, 404 e 500; além de um teste de carga com 160 requisições diretas. O projeto em *Flutter* contou com 90 testes, enquanto o de *React Native* teve 47.

A avaliação considerou métricas como linhas de código, consumo médio de CPU e memória, número de falhas nas requisições, facilidade de acesso à documentação, uso de mocks e quantidade de dependências.

6.1. Número de linhas de código

A primeira métrica analisada foi a quantidade de linhas de código necessárias para implementar os mesmos testes em cada *framework*, conforme apresentado na imagem (Figura 2). Apesar de ser uma medida subjetiva, ela demonstra uma similaridade entre ambos os framewors.

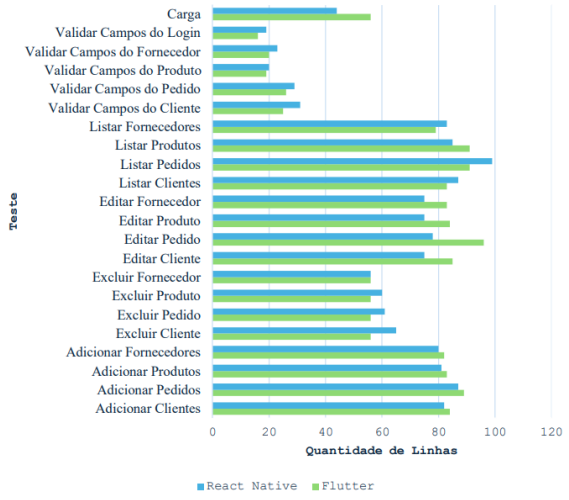

Figura 2. Quantidade de linhas de código de cada teste

6.2. Média de Consumo De CPU e Memória

A segunda métrica avaliada foi o consumo médio de CPU e memória RAM durante a execução dos testes para comparar o desempenho entre os frameworks. Esse parâmetro foi baseado em um notebook convencional e replicado em outros dois computadores, tendo resultados similares, respeitando-se as diferenças de hardware. No cenário exposto os testes em *Flutter* duraram 80 segundos, a CPU iniciou em 39% e a RAM em 86%, atingindo picos de 98% (CPU) e 93% (RAM) aos 31 segundos, com variações até o fim. No *React Native*, em 18 segundos de teste, o uso inicial foi 81% para CPU e RAM, chegando a 99% de CPU e 86% de RAM aos 7 segundos, mantendo-se acima de 80% por grande parte da execução. Os testes foram repetidos em ciclos de cinco iterações para garantir a precisão e coerência dos dados, sempre com valores próximos. As medições serviram para demonstrar o comportamento da utilização de recursos em ambos os *frameworks*.

6.3. Número de Requisições com Falhas

A terceira métrica analisada foi o número de requisições com falhas para avaliar a usabilidade da API em testes de *stress*. Individualmente, não houve falhas nas requisições em nenhum dos *frameworks*. Porém, na execução simultânea de todos os testes, o teste de carga apresentou falhas por timeout. Após os ciclos de execuções, o *Flutter* apresentou menos erros no teste de estresse, enquanto o *React Native* apresentou falhas em todos os ciclos executados.

6.4. Facilidade de Documentação

A quarta métrica analisada foi a facilidade de acesso à documentação e exemplos para auxiliar na criação dos testes. Em ambos os *frameworks*, foram encontrados diversos materiais, incluindo documentação oficial e exemplos práticos, como a biblioteca *React Native Faker* para dados simulados, além de blogs de desenvolvedores utilizados durante o desenvolvimento dos testes.

6.4.1. Facilidade de Utilizar os Mocks

A quinta métrica avaliada foi a facilidade de uso dos mocks, visando comparar a simplicidade na geração de dados sintéticos para testes. Ambos os projetos utilizaram a biblioteca *Faker* — faker no *Flutter* e faker-js no *React Native* —, que se mostraram simples de usar e sem restrições quanto aos tipos de dados gerados. Vale destar que, mesmo com métricas subjetivas, o estudo permite nortear os desenvolvedores que desejam entender e utilizar estas plataformas.

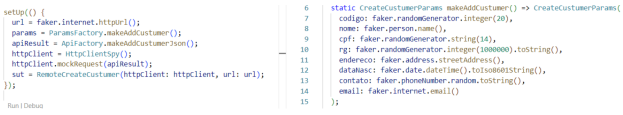

Figura 3. Representação da utilização do Mock no Flutter

Figura 4. Representação da utilização do Mock no React Native

6.5. Quantidade de Dependências

A última métrica analisada foi a quantidade de dependências utilizadas nos testes, avaliando o número de bibliotecas, classes ou funções importadas em cada *framework* para entender o impacto no desenvolvimento.

7. Conclusão e Trabalhos Futuros

A análise dos testes em *Flutter* e *React Native* mostrou poucas diferenças na quantidade de código, dependências e consumo de CPU e memória. Porém, o *React Native* apresentou tempo de execução significativamente menor que o *Flutter*.

A utilização de dados sintéticos e a disponibilidade de documentação foram semelhantes em *Flutter* e *React Native*. Apesar da curva inicial de aprendizado, a implementação dos testes se torna mais ágil com a experiência. Ambos os *frameworks*

mostram eficácia na implementação de testes de integração, com a escolha influenciada pela familiaridade com *Dart* ou *JavaScript*, pelas necessidades de customização — favorecendo *Flutter* — e pela otimização no uso de recursos — favorecendo *React Native*.

A relevância deste estudo vai além a comparação técnica entre os frameworks, pois fornece um guia para profissionais que desejam se aprofundar no assunto de testes de integração no desenvolvimento móvel. A crescente adoção de frameworks multiplataforma impõe a necessidade de implementar testes robustos e eficientes, que mitiguem os riscos inerentes à integração entre módulos e a complexidade da comunicação com APIs.

Também, a investigação metódica baseada no paradigma GQM demonstra como a engenharia de software pode ser aplicada para embasar decisões técnicas, elevando o patamar da atividade de teste de uma função meramente executiva para uma função estratégica e analítica. A constatação de que a complexidade inicial se equilibra entre os frameworks é um *insight* para o planejamento de treinamentos e a alocação de recursos humanos em projetos.

Por fim, a decisão entre *Flutter* e *React Native* deve considerar o domínio das linguagens, a personalização necessária e o uso de recursos computacionais. Recomenda-se que pesquisas futuras ampliem o escopo do sistema, realizem testes em ambiente produtivo e adotem TDD no *React Native* para melhor comparação. Conclui-se que ambos os *frameworks* são adequados para aplicações móveis, cabendo a escolha às prioridades do projeto.

Referências

Ammann, P. and Offutt, J. (2017). *Introduction to software testing*. Cambridge University Press, 2 edition.

Andrade, A. P. (2020). O que é flutter? https://www.treinaweb.com.br/blog/o-que-e-flutter. Acesso em: 16 ago. 2025.

Beizer, B. (1995). *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1 edition.

Bernardo, P. C. and Kon, F. A. (2008). A importância dos testes automatizados. *Engenharia de Software Magazine*, 1(3):54–57.

Carvalho, S. (2022). Testes manuais x testes automatizados. https://safewayconsultoria.com/testes-manuais-x-testes-automatizados/. Acesso em: 16 ago. 2025.

Charles, C. (2023). Desenvolvimento mobile: o que é, desafios e tendências. https://blog.brq.com/desenvolvimento-mobile/. Acesso em: 16 ago. 2025.

Craig, R. D. and Jaskiel, S. P. (2002). *Systematic Software Testing*. Artech House Publishers.

Desikan, S. and Ramesh, G. (2006). *Software testing: principles and practice*. Pearson Education India.

Graham, D., Vliet, H. V., Hoof, E., and Luka, T. (2008). *Foundations of software testing: ISTQB certification*. International Thomson Business Press, 1 edition.

Hussain, A. and Kutar, M. (2009). Usability metric framework for mobile phone application. *PGNet*.

Kaur, A. and Kaur, K. (2022). Systematic literature review of mobile application development and testing effort estimation. In *Journal of King Saud University - Computer and Information Sciences, Volume 34, Issue 2*. Fevereiro de 2022. https://www.sciencedirect.com/science/article/pii/S1319157818306074. Acesso em: 16 ago. 2025.

Lewis, W. E. and Veerapillai, G. (2004). *Software testing and continuous quality improvement*. Auerbach Publications.

Luft, C. C. (2012). Teste de software: uma necessidade das empresas. Master's thesis, Universidade Regional do Noroeste do Estado do Rio Grande do Sul.

Marques, S. (2022). React native: O que é, como funciona e quais as vantagens? https://uds.com.br/blog/react-native-o-que-e/. Acesso em: 16 ago. 2025.

Mili, A. and Tchier, F. (2015). *Software testing: concepts and operations*. John Wiley Sons.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley Sons, 3 edition.

Neto, A. C. D. (2009). Introdução a teste de software. (1):54–59.

Pignati, G. (2021). Demanda por desenvolvedor mobile cresce em 600 por cento 2021.

Pressman, R. S. and Maxim, B. R. (2021). *Engenharia de software: uma abordagem profissional*. AMGH, 9 edition.

Rocha, A. R. C., Maldonado, J. C., Weber, K. C., et al. (2001). *Qualidade de software – Teoria e prática*. Prentice Hall.

Rodriguez, M. (2019). A história dos aplicativos – quem usa e quem vive de desenvolver. Acesso em: 16 ago. 2025.

Salomão, R. G. (2016). Análise da relevância de teste de regressão para o mercado de desenvolvimento de software do triângulo mineiro. Master's thesis, Universidade Federal de Uberlândia. Acesso em: 16 ago. 2025.

Santos, D. B. (2016). Implantação de teste de software em empresa de pequeno porte: Um estudo de caso. Master's thesis, Centro Universitário Eurípides de Marília – UNIVEM. https://aberto.univem.edu.br/bitstream/handle/11077/1577/Monografia

Sommerville, I. (2021). *Engenharia de Software*. Pearson, 10 edition.

Souza, K. P. and Gasparotto, A. M. S. (2013). A importância da atividade de teste do desenvolvimento de software. In *XXXIII Encontro Nacional de Engenharia de Produção*, Salvador, BA, Brasil. 08 a 11 de outubro de 2013.

Zahra, H. A. and Zein, S. (2022). A systematic comparison between flutter and react native from automation testing perspective. In *2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Ankara, Turkey. 20 a 22 de outubro de 2022. https://ieeexplore.ieee.org/abstract/document/9932749. Acesso em: 16 ago. 2025.