UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA - CCET
DEPARTAMENTO DE COMPUTAÇÃO - DC

André Silveira Sousa

Teste Baseado em Modelo em Aplicativos Móveis: Uma Avaliação com as Ferramentas GraphWalker e Appium

SÃO CARLOS - SP
2025

André Silveira Sousa

Teste Baseado em Modelo em Aplicativos Móveis: Uma Avaliação com as Ferramentas GraphWalker e Appium

Trabalho de conclusão de curso apresentado ao Departamento de Computação da Universidade Federal de São Carlos, para obtenção do título de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. André Takeshi Endo

SÃO CARLOS - SP
2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Cienêcias Exatas e de Tecnologia - CCET
Departamento de Computação

Comissão avaliadora

Membros da comissão examinadora que avaliou e aprovou a Defesa do Trabalho de Conclusão de Curso do candidato André Silveira Sousa, realizada em 10/12/2025

Prof. Dr. André Takeshi Endo
Instituição: Universidade Federal de São Carlos

Prof. Dr. Daniel Lucrédio
Instituição: Universidade Federal de São Carlos

Profª. Drª. Érica Ferreira de Souza
Instituição: Universidade Tecnológica Federal do Paraná

Dedicatória

Dedico este trabalho aos estudantes e profissionais da área de QA, que estão sempre buscando novas e diversas formas de garantir a qualidade de sistemas que acompanham nossas vidas.

AGRADECIMENTO

Inicialmente agradeço à Michele Luiza de Bairros Zaboto, minha primeira gestora e referência profissional, que me apresentou à área de qualidade e testes de software, que me fascina e que me motivou à escolha do tema do trabalho.

Agradeço ao meu orientador Prof. Dr. André Takeshi Endo, pela paciência, pelos conselhos e pelo conhecimento, que me mostrou caminhos novos de aprendizado que agregaram muito em minha vida e não somente no andamento deste trabalho.

Agradeço também ao Prof. Dr. Auri Marcelo Rizzo Vincenzi, por ter me acompanhado durante o meu estágio, e que me mostrou as inúmeras oportunidades de crescimento na área de qualidade de software, que com certeza agregaram enormemente na minha vida e na minha carreira.

Agradeço aos meus colegas de trabalho Vitor, Bárbara, Priscilla, Jessyka e Waleska que me acompanharam em minha vida profissional, me apoiaram, motivaram e inspiraram diariamente.

Agradeço à Marcia João Pedro e aos meus amigos e colegas de faculdade Vinicius, Marciel, Marina, Jhulie, Matteus, Amanda e Juliana por serem pessoas incríveis, serem companheiros e terem e me auxiliado em períodos difíceis. Obrigado por serem a minha família de coração!

Agradeço também aos meus pais Jorge e Débora e ao meu irmão Danilo, por sempre terem incentivado os meus estudos, mesmo enfrentando dificuldades. Obrigado por terem acreditado em mim e nas minhas loucuras.

Por último, mas não menos importante, gostaria de agradecer a todas as pessoas que trabalham pela existência da universidade pública, gratuita e de qualidade, pelo sistema de cotas e pelos programas de assistência à permanência estudantil. Sem estes equipamentos eu nunca teria cogitado ser possível me tornar engenheiro. Obrigado por tudo.

Só por que alguma coisa não faz o que você planejou que ela fizesse, não quer dizer que ela seja inútil - Thomas Edison

RESUMO

Com o crescente mercado de software disponibilizado por meio de aplicativos móveis, tornam-se relevantes os estudos sobre a variedade de abordagens de testes para este ambiente de forma a reforçar a qualidade dos sistemas. Este estudo avalia o uso de Teste Baseado em Modelo (TBM) na automação de testes *end-to-end* em aplicativos Android, utilizando a ferramenta de TBM GraphWalker e o *framework* de automação Appium. Os testes foram implementados em três aplicativos *open source* utilitários pelo primeiro autor e seguiram passos bem definidos para modelagem, codificação, execução de testes e análise de falhas. Os resultados apontaram uma avaliação balanceada, que condiz com a elevada curva de aprendizado de TBM presente na literatura. Considerando que no contexto do estudo o nível de conhecimento prévio com as ferramentas era baixo, os melhores resultados foram obtidos em uma implementação de escopo reduzido, no qual o aplicativo possuía elementos bem indexados, o que facilitou o uso do Appium, e a modelagem feita com um único modelo enxuto, o que facilitou o uso do GraphWalker.

Palavras-chave: Teste Baseado em Modelo, Testes *end-to-end*, Testes automatizados, Android, GraphWalker, Appium

ABSTRACT

With the growing market of software available via mobile applications, studies about the variety of testing approaches for this environment become relevant in order to reinforce the quality of those systems. The purpose of this study is to evaluate the use of Model-Based Testing (MBT) on automation of end-to-end tests in Android applications, using GraphWalker as MBT tool and Appium as automation framework. The tests were implemented in three open-source utility applications by the first author and followed well-defined steps for modeling, coding, test execution, and failure analysis. The results indicated a balanced evaluation, consistent with the steep learning curve of TBM found in literature. Considering that the level of prior knowledge with the tools was low in the context of the study, the best results were obtained in a reduced-scope implementation where the application had well-indexed elements, which facilitated the use of Appium, and the tests were modeled with a single and lean model, which facilitated the use of GraphWalker.

Keywords: Model-Based Testing, End-to-End Testing, Automated Testing, Android, GraphWalker, Appium

Lista de Figuras

1 Pirâmide de testes 15
2 Exemplo simples do modelo de um multímetro em TBM 17
3 Exemplo de *generator* no GraphWalker em Java 18
4 Diagrama de blocos das atividades 22
5 Diagrama de blocos da etapa de exploração 25
6 Exemplo de modelo gerado no GraphWalker para o aplicativo Notepad . . 27
7 Diagrama de blocos da etapa de codificação 28
8 Comandos Maven para geração automática de interfaces pelo GraphWalker 29
9 Exemplo de classe para iniciação do driver Android 30
10 Exemplo de classe de *Page Objects* 31
11 Exemplo de classe de implementação dos testes 32
12 Exemplo de classe Main 33

Lista de Tabelas

1 Especificações dos equipamentos utilizados 23
2 Dados dos aplicativos escolhidos 24
3 Exemplo de registro e classificação de falhas 34
4 Experiência no uso das ferramentas 39
5 Dificuldade de uso das ferramentas 39
6 Tempo gasto por etapa por aplicativo 40
7 Métricas dos modelos 41
8 Métricas de código 42
9 Falhas encontradas nos testes do Notepad 43
10 Causas e soluções de falhas do Notepad 43
11 Falhas encontradas nos testes do Gallery 43
12 Causas e soluções de falhas do Gallery 44
13 Falhas encontradas nos testes do File Manager 44
14 Causas e soluções de falhas do File Manager 45
15 Tempo de execução dos testes 46
16 Análise do esforço na implementação dos testes 46

Conteúdo

Lista de Figuras 8

Lista de Tabelas 9

1 INTRODUÇÃO 12

1.1 Objetivos 14

1.2 Estrutura do Trabalho 14

2 REVISÃO BIBLIOGRÁFICA 15

2.1 Automação de Testes em Aplicativos Móveis 15

2.2 Teste Baseado em Modelo 16

2.3 Trabalhos Relacionados 19

3 METODOLOGIA DO ESTUDO 21

3.1 Etapas Realizadas 21

3.2 Seleção dos Aplicativos 23

3.3 Navegação Exploratória 24

3.4 Modelagem com GraphWalker 26

3.5 Codificação 27

3.6 Execução dos Testes 33

3.7 Depuração e Análise de Falhas 34

4 ANÁLISE DE RESULTADOS 36

4.1 QP1 — Qual a viabilidade de utilizar TBM para a automação de testes *end-to-end* em aplicativos móveis? 36

4.1.1 Java & Maven 36

4.1.2 GraphWalker 37

4.1.3 Appium 37

4.1.4 Resposta à QP1 38

4.2 QP2 - Qual o esforço envolvido na implementação dos testes? 39

4.2.1 Métricas de tempo gasto 39

4.2.2 Métricas dos modelos 40

4.2.3 Métricas de código 41

4.2.4 Falhas apontadas pelos testes 42

4.2.5 Métricas de execução 45

4.2.6 Resposta à QP2 46

4.3 QP3 - Quais os desafios observados no uso do TBM? 47

4.3.1 Vantagens 47

4.3.2 Dificuldades 48

4.3.3 Lições Aprendidas 49

5 CONSIDERAÇÕES FINAIS **50**

Referências **51**

1 INTRODUÇÃO

Dispositivos móveis estão presentes no dia-a-dia de milhões de pessoas mundialmente. Uma ampla gama de serviços, tanto públicos quanto privados, para as mais diversas necessidades estão disponíveis na palma das mãos, tornando a vida das pessoas cada vez mais prática. Dadas estas circunstâncias, nos últimos anos observou-se uma crescente demanda entre as empresas no fornecimento de seus serviços por este meio, possibilitando que eles sejam utilizados massivamente a qualquer hora e local (GSMA, 2025). A grande variedade de aplicativos disponíveis que competem por uma mesma função realça a necessidade de que critérios importantes para o usuário sejam atendidos, de modo a garantir a adoção do aplicativo pelo público e, consequentemente, o retorno esperado pelas empresas desenvolvedoras. Um dos critérios importantes é a qualidade do desenvolvimento, que é tema de estudo há décadas e ainda enfrenta muitos desafios (Silva et al., 2022).

No campo da Engenharia de Software, há uma área de pesquisa relacionada à garantia de qualidade (em inglês, *Quality Assurance* - QA) (Pressman e Maxim, 2021). A área de QA tem como uma das principais atividades o teste de software (Orso e Rothermel, 2014). Os testes são realizados por uma equipe de profissionais durante o ciclo de desenvolvimento e antes da disponibilização dos aplicativos para o usuário final, visando majoritariamente minimizar a quantidade e a criticidade de problemas (também conhecidos como *bugs*). Uma das abordagens de teste de software é o Teste Baseado em Modelo (Kong et al., 2019) que será investigado neste estudo.

O Teste Baseado em Modelo (TBM) é uma abordagem que utiliza modelos, como máquinas de estados finitos, para representar o comportamento de uma determinada aplicação (Neto et al., 2008). Os modelos gerados podem ser usados para diversas necessidades da atividade de teste, como geração de dados, geração de casos de teste e geração de *scripts* de testes automatizados, como explicado por Utting e Legeard (2010).

De acordo com Kramer e Legeard (2016), as atividades de testes, embora essenciais, nem sempre podem ser aplicadas por completo na prática, por dependerem de variáveis como custo, tempo de teste, conhecimento do profissional, detalhamento dos requisitos, complexidade da funcionalidade, entre outros. O tempo de teste é um fator importante que evidencia a necessidade de se desenvolver testes automatizados, principalmente para garantir que funcionalidades já desenvolvidas continuem sendo executadas corretamente após a disponibilização de uma nova versão do aplicativo, removendo a necessidade do profissional despender tempo em testes corriqueiros e repetitivos manualmente. Considerando estas informações, o TBM se torna uma opção interessante, pois além de reduzir o tempo gasto na documentação de casos de teste, ao utilizar modelos gráficos ao invés de texto, também pode incrementar a cobertura dos casos de teste através do reaproveitamento do próprio modelo (Silva et al., 2018).

Uma das ferramentas que permite a aplicação de TBM é o GraphWalker (GraphWal-

ker, 2025), que oferece uma interface de modelagem em navegador *web*, denominada GraphWalker Studio, o qual permite a construção de modelos de forma simples e visual. Os modelos podem ser exportados em formato JSON para integração com ambientes de desenvolvimento que possuem suporte às bibliotecas do projeto na linguagem Java. No presente trabalho, o GraphWalker foi adotado em função de sua facilidade de uso e pela existência de estudos recentes que exploram sua aplicação, como em Gudmundsson et al. (2016) e Matta e Garousi (2025).

O TBM pode ser aplicado em diferentes domínios, porém o contexto de aplicativos móveis apresenta especificidades que tornam a automação desafiadora, como abordado por Farto (2016). O autor expõe tópicos que necessitam avaliação especial, de modo a validar o uso característico dos aplicativos, como interface sensível ao toque, diferentes resoluções de tela, uso de recursos computacionais e sensores, por exemplo. Para que estes pontos sejam avaliados, portanto, é necessário que o profissional tenha acesso a tais dados dos dispositivos. Por conta disso neste trabalho o escopo se limita a dispositivos com o sistema operacional Android. O sistema Android possui um portal de desenvolvimento (Studio, 2025) que permite o acesso a diversos recursos além de também permitir emular uma ampla variedade de dispositivos, já que não se torna viável que o profissional possua aparelhos físicos suficientes para cobrir a fragmentação característica do sistema devido a grande diversidade de dispositivos presentes no mercado. Nesse cenário, a implementação de TBM pode contribuir para a atividade de testes, pois reduz o esforço manual e torna possível a execução sistemática dos casos de teste em diferentes versões a partir do reuso dos modelos (Farto e Endo, 2017).

A aplicação de TBM em aplicativos móveis seguirá o padrão sugerido por Utting e Legeard (2010), que indica a geração do modelo para automatizar os testes de sistema em nível caixa-preta (em inglês, *black-box testing*) no sistema sob teste (em inglês, *System Under Test* - SUT). Este tipo de teste é mais fiel à forma como o usuário final interage com o aplicativo, pois não considera a codificação ou a integração em nível de *back-end* do sistema. A intenção principal é fazer uma varredura em uma versão compilada do SUT com testes de ponta-a-ponta (em inglês, *end-to-end* - E2E), simulando o comportamento do usuário final e comparando o resultado obtido no teste com o resultado esperado daquela determinada ação, com o objetivo de indicar, quantitativamente, quantos casos de teste estão em conformidade com o esperado e quantos apresentam *bugs*. O modelo gerado, portanto, deve cobrir as funcionalidades de interesse do SUT, de acordo com o escopo definido.

Para fazer a integração do SUT com os modelos gerados pela ferramenta de TBM GraphWalker, será utilizado o *framework* Appium (Appium, 2025). Este *framework* é um projeto *open source* amplamente utilizado na automação de testes em nível de interface gráfica (em inglês *Graphic User Interface* - GUI) que pode ser integrado a diversos ambientes, que inclui dispositivos móveis (Singh, Gadgil e Chudgor, 2014). Este *framework*

possibilita a codificação dos *scripts* de teste em diferentes linguagens de programação, que inclui Java, o que possibilita o uso concomitante com as bibliotecas do GraphWalker.

1.1 Objetivos

Este trabalho teve como objetivo avaliar a adoção da abordagem de Teste Baseado em Modelo na automação de testes *end-to-end* em aplicativos móveis, analisando as dificuldades e o esforço envolvido em sua primeira implementação com a ferramenta de TBM GraphWalker e o *framework* de automação Appium. Os aplicativos foram selecionados a partir de repositórios *open source*. O desenvolvimento dos projetos e a codificação dos testes foi disponibilizado em um repositório aberto em conjunto com um guia de replicação do estudo.

1.2 Estrutura do Trabalho

A estrutura deste trabalho apresenta-se da seguinte forma: No Capítulo 2 há a revisão bibliográfica dos pontos chaves do estudo: automação de testes em aplicativos móveis, Teste Baseado em Modelo e trabalhos relacionados. No Capítulo 3 é detalhado o método de estudo. No Capítulo 4 são apontados e analisados os resultados obtidos e no Capítulo 5 são apresentadas conclusões e também sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

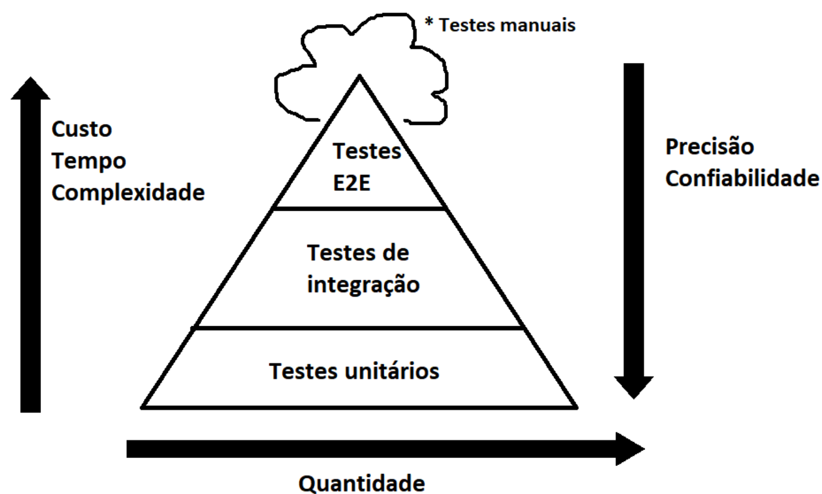
Neste capítulo são apresentados os conceitos abordados no trabalho, divididos nos tópicos: Automação de Testes em Aplicativos Móveis na Seção 2.1, e Teste Baseado em Modelo na Seção 2.2. Em sequência, são discutidos os Trabalhos Relacionados na Seção 2.3.

2.1 Automação de Testes em Aplicativos Móveis

Nos estudos clássicos de Engenharia de Software, Pressman e Maxim (2021) destacam a importância da etapa de testes no ciclo de desenvolvimento. No caso dos aplicativos móveis, enquanto produtos de software, considera-se que seguem padrões já consolidados em metodologias ágeis (Flora e Chande, 2013). Há diversos níveis de testes que podem ser identificados, porém, neste trabalho a abordagem será focada na camada mais externa: os testes de sistema *end-to-end*.

Estes testes, que são geralmente conduzidos pela equipe de qualidade após a compilação de uma versão do aplicativo, tem como objetivo principal simular a interação do usuário final, de modo a identificar os *bugs* que causam impacto direto na usabilidade, ou seja, aqueles que são visíveis para o usuário e podem prejudicar a sua experiência. Estes testes são frequentemente representados no topo da pirâmide de testes (Cohn, 2009), como ilustrado na Figura 1. A pirâmide classifica os testes em três níveis: unitários (nível de código), de integração (nível de API) e *end-to-end* (nível de interface gráfica), e os relacionam a variáveis como quantidade, custo, complexidade e precisão.

Figura 1: Pirâmide de testes



Fonte: Autoria própria, adaptado de Cohn (2009)

Como retratado na literatura, os testes *end-to-end* são considerados muito custosos

em termos de tempo, recursos e complexidade (Orso e Rothermel, 2014); e muitas vezes são testados manualmente. Por conta disso, há um esforço contínuo em automatizar esse tipo de teste, de modo a reduzir tais variáveis. A automação tem se tornado uma prática fundamental no ciclo de desenvolvimento de software, especialmente em sistemas de integração contínua (Labuschagne, Inozemtseva e Holmes, 2017), pois neste tipo de desenvolvimento as funcionalidades do sistema sofrem alterações com frequência, consequentemente levando a testes repetitivos e alto custo de tempo e esforço. Quando há automação nos testes *end-to-end* de funcionalidades consideradas críticas no SUT, os ganhos são muito positivos, pois possíveis problemas que podem impactar a experiência do usuário podem ser rastreados com facilidade, e a partir desta identificação, correções podem ser feitas antes da disponibilização da nova versão do produto para o cliente, garantindo maior qualidade (Karhu et al., 2009).

No ecossistema Android, algumas especificidades tornam os testes particularmente complexos e suscetíveis a falhas (Farto, 2016). Entre elas, destaca-se a ampla variedade de dispositivos disponíveis no mercado, que apresentam diferenças no tamanho de tela, sensibilidade a gestos e versões distintas do sistema operacional. Esse cenário implica que a validação de um teste em um dispositivo ou versão específica não garante necessariamente o mesmo resultado em outros, exigindo múltiplas execuções para assegurar a consistência entre plataformas. Idealmente, essa repetição deveria ser sistemática em todas as versões suportadas, mas na prática se torna inviável, sobretudo em contextos de testes manuais (Linares-Vásquez, Moran e Poshyvanyk, 2017). Farto (2016) também cita desafios relevantes no uso dos sensores dos aparelhos, que ampliam a complexidade dos testes. Recursos como o sensor biométrico, por exemplo, amplamente utilizado em diversas aplicações, não são adequados para automação, pois seria um grave problema de segurança permitir que uma identidade seja validada biometricamente por uma atividade que não seja do usuário real.

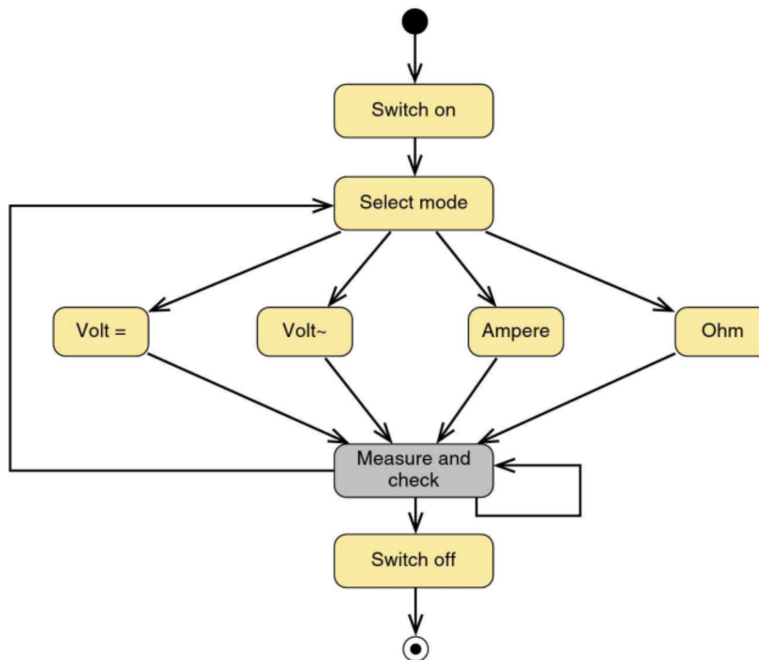
Diante desses obstáculos, destaca-se a importância de ferramentas adequadas para a automação de testes de interface gráfica em sistemas Android. O *Appium* (Appium, 2025), por exemplo, viabiliza a automação em cenários de grande diversidade. O trabalho de Singh, Gadgil e Chudgor (2014) explora o uso deste *framework*, elencando as vantagens que o Appium apresenta em relação aos testes manuais, além de compará-lo a outras ferramentas de automação.

2.2 Teste Baseado em Modelo

O teste baseado em modelo (TBM) é uma abordagem de teste de software que utiliza um ou mais modelos gráficos em sua execução. Os modelos representam funcionalidades do SUT, e são indicados na representação do comportamento do sistema para realização de testes funcionais, ou seja, percorrendo suas funcionalidades em alto nível, após a com-

pilação (Utting e Legeard, 2010). Os modelos podem ser representados por máquinas de estados, que são grafos constituídos de vértices e arestas, nos quais cada vértice representa um estado e cada aresta é unidirecional e representa uma ação que pode fazer o sistema ir de um estado a outro, ou permanecer no mesmo estado. Ao obter-se o modelo, é possível extrair os casos de teste realizando o caminho dado um vértice de partida até um vértice de chegada. Logo, nesta abordagem, ao utilizar-se ferramentas de TBM, a geração de casos de teste pode ser feita de maneira automática, sendo de competência do profissional de qualidade a síntese e a manutenção do modelo. Em Kramer e Legeard (2016), há um exemplo simples do modelo de um sistema de medição de um multímetro, ilustrado na Figura 2. Neste sistema, cada retângulo arredondado é um estado, e cada aresta, representado pelas setas é uma ação do usuário.

Figura 2: Exemplo simples do modelo de um multímetro em TBM



Fonte: Extraído de Kramer e Legeard (2016).

É importante observar que quando há pelo menos a presença de um laço de repetição (em inglês, *loop*) no modelo, o mesmo poderá gerar infinitos casos de teste. Na Figura 2, por exemplo, há dois *loops*, o que indica que o usuário pode permanecer utilizando o sistema e fazendo diversas medições indefinidamente, pois a execução do teste somente seria encerrada quando o usuário escolher a opção de desligar o *switch* após a medição, e esta escolha depende de como a lógica usada para percorrer os caminhos foi implementada. Portanto, em ferramentas de TBM é importante definir alguns parâmetros, como a cobertura (em inglês, *coverage*) que indica, em forma proporcional, quantos vértices devem ser visitados na execução, ou uma limitação de quantidades de vezes que se deve

percorrer um *loop*, caso presente no modelo. Em um exemplo prático, na ferramenta de TBM GraphWalker (GraphWalker, 2025), é recomendado o uso de uma anotação chamada *generator*, que é um algoritmo que trata da decisão de como percorrer o modelo. Na Figura 3 a anotação está indicando que a decisão da próxima ação será de forma aleatória (*random*) e que todos os vértices devem ser visitados (cobertura de vértices 100%). Desta forma, o modelo irá percorrer caminhos aleatórios em cada execução, e irá parar no momento que identificar que todos os vértices foram percorridos pelo menos uma vez.

Figura 3: Exemplo de *generator* no GraphWalker em Java

```java
@GraphWalker(value = "random(vertex_coverage(100))")
public class Test {
    ...
}
```

Fonte: Documentação do GraphWalker (2025)

Esta característica é uma grande vantagem do TBM, pois a cobertura de casos de teste pode ser facilmente configurada e diversos cenários alternativos podem ser mapeados automaticamente, já que cada execução percorrerá um caminho aleatório que pode ser diferente de uma execução anterior do mesmo teste (Farto e Endo, 2017).

Utting e Legeard (2010) apresentam duas estratégias de uso dos modelos em TBM. Na primeira delas, conhecida como *online*, os testes são executados à medida que são produzidos, de modo que a ferramenta de TBM gerencia o processo de execução dos testes e registra os resultados. Já na segunda, conhecida como *offline*, o modelo é utilizado apenas para a geração dos casos de teste, que em seguida podem ser exportados, o que permite que ferramentas e práticas de teste de sistemas que já possuem automação implementada possam continuar sendo utilizadas.

De acordo com Kramer e Legeard (2016) o TBM é tema de estudo desde os primórdios da Engenharia de Software e está presente em uma das certificações mais importantes da área, o ISTQB (*International Software Testing Qualifications Board*, em inglês) porém, apesar das vantagens, os autores também indicam dificuldades que limitam a adoção de TBM na prática. Segundo Németh (2020), essa limitação está relacionada principalmente à curva de aprendizagem elevada e à exigência de um raciocínio distinto das abordagens convencionais, o que pode ser especialmente desafiador em sistemas com alto número de funcionalidades pois, neste caso, é mais fácil pensar nos casos de teste como pequenas unidades do sistema completo, que geralmente são derivados de documentos de requisitos (Pressman e Maxim, 2021). Apesar disso, basear-se completamente em documentos de requisitos também pode apresentar limitações, pois Kramer e Legeard (2016) também apontam uma potencial dificuldade na manutenção e validação de documentos demasiadamente grandes, o que abre vantagem para a abordagem baseada em modelo.

Nos últimos anos foram publicadas diversas pesquisas referentes ao uso de TBM nas

atividades de teste de software. Na pesquisa de Bernardino et al. (2017) foram levantadas as principais ferramentas de TBM presentes na literatura em dez anos (de 2006 a 2016). Dentre ferramentas acadêmicas, comerciais e *open-source*, o GraphWalker (GraphWalker, 2025) está presente. Além de ser *open source* e de fácil instalação, também é uma ferramenta utilizada em estudos mais recentes, como em Garousi et al. (2021), que abordam a adoção do GraphWalker em uma empresa real voltada a testes de softwares Web e Mobile; e em Matta e Garousi (2025), que apresentam uma solução para geração de modelos no GraphWalker de forma automatizada, por meio de dados de cliques de usuários em um sistema Web. Por conta dos motivos citados, o GraphWalker foi a ferramenta de TBM escolhida para este trabalho.

2.3 Trabalhos Relacionados

No contexto específico de aplicativos móveis, Farto (2016) investiga a adaptação do TBM frente a desafios característicos dessa plataforma, como fragmentação de dispositivos, interações por gestos e uso de sensores. Um aspecto importante identificado pelo autor é a necessidade de modelos que considerem variações de contexto, de modo a representar diferentes condições de uso. Seguindo nessa linha, Farto e Endo (2017) exploram o reúso de modelos em aplicativos móveis, demonstrando que um mesmo modelo pode ser aproveitado para diferentes tipos de testes, o que reduz significativamente o retrabalho e o tempo de manutenção.

Além disso, Kong et al. (2019) realizam uma revisão da literatura sobre abordagens de teste para aplicativos Android, analisando um conjunto de 103 artigos relevantes publicados entre 2010 e 2016. Os autores apontam um crescimento consistente no número de pesquisas que utilizam TBM, que é dominante em relação a outras no contexto de testes a nível GUI, e representam 63% dos artigos revisados.

Em outros contextos, há iniciativas práticas interessantes que abordam o uso do TBM, como em sistemas Web. Garousi et al. (2021) descrevem um estudo sobre a implementação de TBM em uma empresa de software real, utilizando o GraphWalker na modelagem, e evidenciam os benefícios na adoção do TBM para os testes *end-to-end* em termos de cobertura de casos de teste, melhores práticas de design de testes e também maior eficácia na detecção de falhas.

De forma complementar, Matta e Garousi (2025) apresentam uma nova ferramenta que busca reduzir o esforço manual necessário para a geração e manutenção dos modelos de TBM, a qual registra dados de fluxos de cliques de usuários em um sistema Web e as utiliza como entrada para a geração de modelos no GraphWalker de forma automática. Os autores relatam que houve uma redução no esforço de projeto do modelo de TBM de mais de 90% nos sistemas ao qual a ferramenta foi utilizada, e que mesmo que ainda seja necessário que um profissional humano inspecione e refine os modelos gerados, o trabalho

é substancialmente menor do que criar os modelos do início.

Estes estudos, embora não focados em aplicativos móveis, são exemplos relevantes da utilização do GraphWalker, que é a mesma ferramenta a ser utilizada neste trabalho, e também trazem à tona questões importantes do uso de TBM para otimizar a etapa de testes *end-to-end*, com a prática na indústria e resultados reais.

Já no uso do GraphWalker junto ao Appium, Gudmundsson et al. (2016) fazem um estudo empírico da implementação de testes automatizados com TBM em um aplicativo Android complexo, que no caso é um jogo do tipo trivia. O desenvolvimento segue a estratégia de TBM *offline*, e os autores relatam que o *script* implementado para o projeto tem o objetivo de poder ser utilizado por indivíduos tanto experientes quanto inexperientes em TBM, já que o aplicativo em questão é de escopo comercial e já possuía uma equipe de testadores que se beneficiariam da implementação dos testes. Os autores também afirmam que o esforço total foi de três meses de trabalho, e que manter um único modelo comportamental para o aplicativo foi fundamental para testá-lo de forma eficiente.

Dessa forma, este trabalho busca investigar a avaliação prática do TBM aplicado à automação de testes *end-to-end* em aplicativos móveis adotando a estratégia *online*, de modo a contribuir com métricas de viabilidade dessa abordagem e o esforço necessário para sua implementação em escopos bem definidos para o SUT.

3 METODOLOGIA DO ESTUDO

Neste capítulo são descritos os passos do estudo. Para melhor análise dos objetivos propostos, foram formuladas as seguintes Questões de Pesquisa (QP):

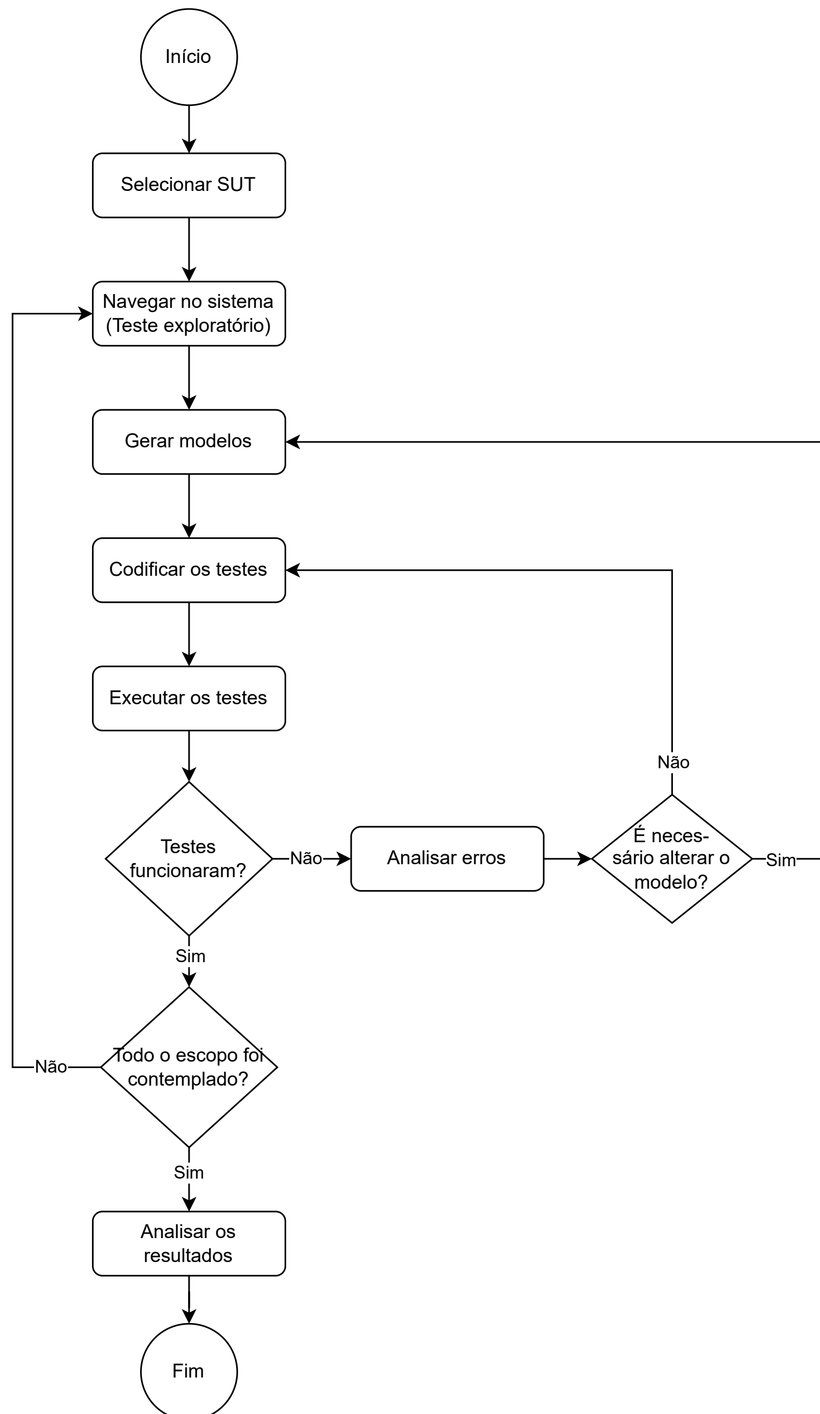
- **QP1 - Qual a viabilidade de utilizar TBM para a automação de testes *end-to-end* em aplicativos móveis?**: o intuito desta questão é avaliar as configurações necessárias para que os testes com TBM sejam implementados e executados de forma satisfatória.
- **QP2 - Qual o esforço envolvido na implementação dos testes?**: a proposta desta questão é avaliar o tempo gasto em cada etapa do desenvolvimento e execução do TBM.
- **QP3 - Quais os desafios observados no uso do TBM?**: esta questão tem como propósito avaliar o impacto da curva de aprendizado descrita na literatura no esforço envolvido na implementação do TBM a partir de dados qualitativos.

As seções a seguir compreendem a execução do estudo em cada passo. Na Seção 3.1 há a estruturação das atividades do trabalho, divididas em etapas específicas. Na Seção 3.2 há os critérios de seleção dos SUTs utilizados. Na Seção 3.3 apresenta-se o detalhamento da navegação exploratória do SUT, com a listagem de suas funcionalidades. Na Seção 3.4 é descrito o processo de geração dos modelos. Na Seção 3.5 há o desenvolvimento dos testes propriamente ditos de modo a integrar o modelo do TBM com o SUT. Na Seção 3.6 detalha-se o processo de execução e avaliação dos testes automatizados, e na Seção 3.7 há a explicação de práticas relevantes para a análise de falhas apontadas pela automação.

3.1 Etapas Realizadas

Considerando os objetivos estabelecidos para este trabalho, foi estruturado um fluxo metodológico que organiza as atividades em etapas sequenciais, de forma a garantir clareza e reprodutibilidade. A representação visual desse fluxo, detalhado na Figura 4, tem como propósito sintetizar as fases principais do estudo, permitindo compreender rapidamente a relação entre elas e como cada etapa contribui para o alcance dos resultados esperados.

Figura 4: Diagrama de blocos das atividades



Fonte: Autoria própria.

Inicialmente, há a seleção do SUT, seguida de uma navegação exploratória na aplicação com o objetivo de identificar suas principais funcionalidades e fluxos de interação. A partir dessas observações, é gerado um modelo no GraphWalker, que posteriormente serve de base para a codificação dos casos de teste automatizados. Na sequência, os testes são executados com o Appium, e os resultados são avaliados para verificar se o comportamento

esperado foi alcançado.

Caso ocorram falhas, realiza-se a análise dos erros e o processo de depuração, que pode levar à necessidade de ajustes tanto no código de teste quanto no próprio modelo. Após a estabilização, verifica-se se os testes estão cobrindo todo o escopo definido. Em caso negativo, novas iterações são conduzidas até que a cobertura desejada seja atingida. Por fim, os resultados obtidos são consolidados e analisados em função das métricas estabelecidas para o estudo.

É importante destacar que este ciclo metodológico será replicado integralmente para cada SUT selecionado. Dessa forma, busca-se avaliar a viabilidade do TBM em cenários distintos e com características funcionais variadas.

Na Tabela 1 são apresentadas as especificações dos equipamentos utilizados no trabalho.

Tabela 1: Especificações dos equipamentos utilizados

Critério	Computador	Smartphone
Modelo	Asus VivoBook X515DA	Motorola Moto g(20)
Processador	AMD Ryzen 5 3500U 2.10GHz 8 núcleos	2x 1.8 GHz Cortex-A75, 6x 1.8 GHz Cortex-A55
Memória	8GB RAM	4GB RAM
Armazenamento	256GB SSD 1TB HDD	64GB Interno
Sistema Operacional	Windows 10 Home	Android 11

Os artefatos e procedimentos para conduzir os testes estão disponíveis em <https://github.com/andresilveiras/mbt-study>

3.2 Seleção dos Aplicativos

A seleção do SUT é uma etapa fundamental para a condução do estudo, pois influencia diretamente na validade e na aplicabilidade dos resultados obtidos. Considerando que o objetivo deste trabalho é avaliar a viabilidade do uso de TBM em cenários distintos, optou-se por adotar mais de um aplicativo como objeto de estudo. Dessa forma, busca-se analisar o desempenho da abordagem em diferentes contextos funcionais, de modo a comparar o esforço necessário para implementação e a cobertura de testes alcançada.

Como critérios de seleção, optou-se por utilizar aplicativos *open source* disponíveis no GitHub, e que tenham relevância em termos de funcionalidades cotidianas para os usuários. Para tal, foram levantados alguns dados conforme apresentado na Tabela 2. Os principais dados levantados foram a quantidade de *downloads* na Google Play e o número de estrelas no repositório do GitHub[1].

[1]Dados extraídos dos Repositórios GitHub e Google Play Store em Setembro de 2025

Tabela 2: Dados dos aplicativos escolhidos

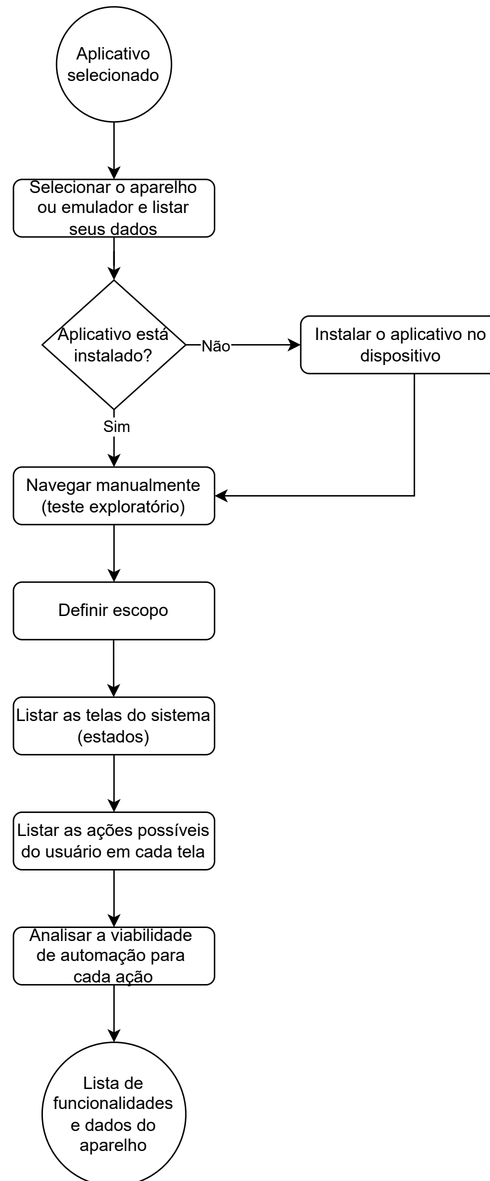
Aplicativo	Desenvolvedor	Downloads Google Play	Estrelas GitHub
Notepad	FarmerBB	500 mil+	366
Gallery	FossifyOrg	100 mil+	2.6k
File Manager	FossifyOrg	50 mil+	1.0k

Com a seleção de mais de um aplicativo, todas as etapas descritas na metodologia devem ser replicadas para cada SUT. Essa replicação sistemática permite a comparação de características que influenciam o uso do TBM na automação de testes *end-to-end* de modo a enriquecer as respostas às perguntas de pesquisa elaboradas.

3.3 Navegação Exploratória

A atividade de navegação exploratória tem como objetivo compreender o comportamento do SUT e identificar os elementos centrais de suas interfaces gráficas, servindo como base para a modelagem a ser conduzida posteriormente. A Figura 5 ilustra um diagrama com as atividades específicas desta etapa.

Figura 5: Diagrama de blocos da etapa de exploração



Fonte: Autoria própria.

O processo inicia-se pela instalação do aplicativo selecionado em um dispositivo físico ou emulador previamente configurado. Caso o aplicativo já esteja instalado, esta etapa é descartada. Em seguida, realiza-se a navegação manual pelas telas do sistema, caracterizando um teste exploratório inicial.

Após a navegação exploratória do aplicativo, realiza-se a definição de escopo, que tem como objetivo estabelecer quais funcionalidades ou fluxos do sistema serão efetivamente considerados na implementação dos testes automatizados. Neste estudo, a definição de escopo limita-se aos fluxos críticos de uso (*Critical User Journeys - CUJ*, em inglês)[2], que

[2] *Workflow* para testes end-to-end difundido pelo Google. Disponível em <https://testing.googleblog.com/2021/>

representam interações essenciais para o propósito central do aplicativo. Essa abordagem mantém o método flexível, de forma que futuros estudos possam selecionar outros fluxos e adicioná-los aos testes, combinando múltiplos modelos, o que resulta em uma cobertura mais ampla do SUT, além de permitir a implementação gradual dos testes.

A partir da definição de escopo, deve-se fazer o registro das telas (estados) e ações do usuário ao percorrer fluxo escolhido, como cliques, gestos e inserções de dados. Em seguida, realiza-se a análise de viabilidade de automação para cada ação identificada. Essa avaliação considera restrições técnicas, como acesso a sensores específicos do dispositivo, dependências externas ou fatores de segurança que inviabilizem a simulação automatizada.

Ao término desta etapa, obtém-se uma lista consolidada das funcionalidades e ações críticas do aplicativo, restritas ao escopo definido, acompanhada também dos dados do aparelho utilizado na exploração, necessários para integração com o Appium, como versão do Android, fabricante e modelo, por exemplo. Essa lista é então utilizada como insumo para a etapa subsequente de modelagem no GraphWalker, para que seja formalmente modelada e posteriormente automatizada no processo de teste.

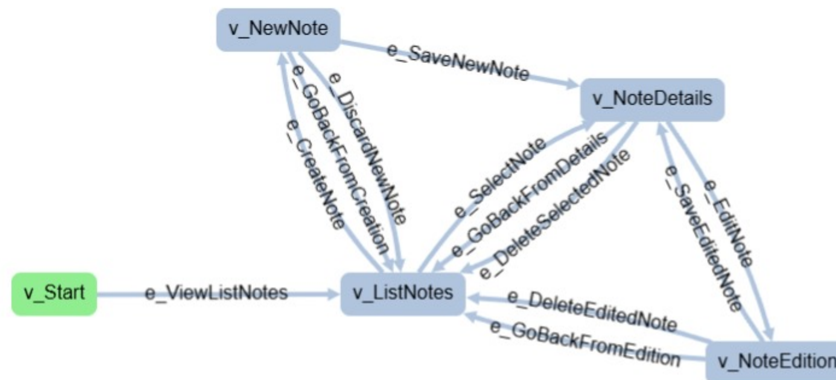
3.4 Modelagem com GraphWalker

A etapa de modelagem tem como objetivo representar formalmente, em forma de grafo, os elementos identificados na navegação exploratória. Cada tela do aplicativo é descrita como um vértice, enquanto as ações possíveis do usuário (por exemplo: clique em um botão, inserção de texto ou gesto de navegação) são representadas como arestas unidirecionais que conectam esses vértices. Dessa forma, obtém-se um modelo comportamental que reflete os possíveis fluxos de interação do usuário com o sistema.

O processo de modelagem foi realizado utilizando a ferramenta GraphWalker Studio, que oferece uma interface gráfica para a criação de modelos. A partir da lista de funcionalidades e transições levantadas, os vértices e arestas foram inseridos manualmente na ferramenta. Neste processo há importantes decisões que podem levar a variabilidade na geração dos modelos por diferentes atores, como por exemplo a quantidade de modelos gerados e a granularidade do modelo, ou seja, o quão detalhado ou genérico ele será. Após a finalização, os modelos podem ser exportados em formato JSON para integração com a automação via Appium.

Para ilustrar esse processo, tomou-se como referência a modelagem do primeiro SUT selecionado, o aplicativo Notepad. A Figura 6 apresenta o modelo construído nesta primeira experiência, no qual as principais telas foram representadas como vértices, enquanto as ações possíveis do usuário foram mapeadas como arestas.

Figura 6: Exemplo de modelo gerado no GraphWalker para o aplicativo Notepad



Fonte: Extraído do GitHub do autor.

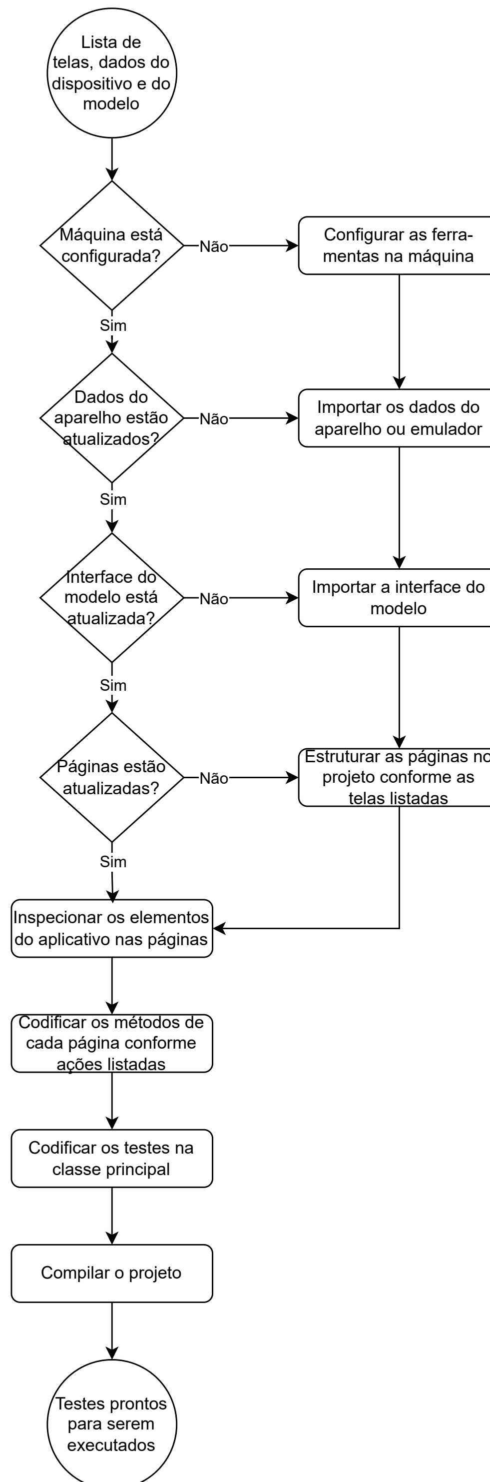
No modelo indicado pela Figura 6, o vértice inicial `v_Start`, que é obrigatório em modelos do GraphWalker, foi definido separadamente para representar a abertura do aplicativo no dispositivo, incluindo eventuais permissões ou validações que ocorrem apenas na primeira execução. Em seguida, foram mapeadas quatro telas principais: `v_ListNotes` (lista de notas), `v_NewNote` (criação de nova nota), `v_NoteDetails` (detalhes de uma nota existente) e `v_NoteEdition` (edição de nota). Cada vértice é conectado por arestas que representam as ações possíveis do usuário, como salvar, descartar ou excluir uma nota, além de transições de navegação entre telas.

Para manter a consistência no processo de codificação, foi adotada a convenção de nomear os vértices com o prefixo `v_` (do inglês *vertex*) e as arestas com `e_` (do inglês *edge*), mantendo os nomes também em inglês. Essa padronização facilita a leitura do modelo e sua integração com a automação no GraphWalker e no Appium. A lógica de construção aqui exemplificada foi replicada para os demais aplicativos analisados neste trabalho, sempre respeitando suas telas e fluxos específicos, mas mantendo a mesma estrutura de modelagem.

3.5 Codificação

A etapa de codificação teve como objetivo implementar, em código, o modelo comportamental definido no GraphWalker, de modo a permitir sua execução automática no ambiente de testes. Esta atividade envolve tanto a configuração inicial das ferramentas quanto a estruturação do projeto de software responsável pela automação. A Figura 7 apresenta o fluxo de atividades que compõem esta etapa.

Figura 7: Diagrama de blocos da etapa de codificação



Fonte: Autoria própria.

O processo inicia-se pela configuração das ferramentas necessárias: Java JDK, Maven, Android Studio e ADB, além do Appium (com o driver UIAutomator2) e do GraphWalker, conforme descrito no projeto inicial do autor[3]. Uma vez configurado o ambiente, os

[3]Disponível em: <https://github.com/andresilveiras/pilot-appium-graphwalker>

dados do dispositivo ou emulador utilizado nos experimentos e o modelo exportado do GraphWalker são utilizados nos próximos passos.

Para iniciar a implementação do código-fonte dos testes deve ser aberto um repositório de um projeto Java. A documentação das ferramentas citadas anteriormente indica as dependências Maven que devem ser incluídas no arquivo pom. O modelo gerado no GraphWalker Studio pode ser exportado no formato JSON e então incluído no projeto. Com o modelo importado no projeto e as dependências configuradas, é possível executar um comando nativo do GraphWalker no terminal para que as interfaces do modelo sejam geradas automaticamente, conforme a Figura 8, e então a lógica dos testes deve ser codificada em classes que implementam as respectivas interfaces.

Figura 8: Comandos Maven para geração automática de interfaces pelo GraphWalker

```
1 mvn clean
2 mvn graphwalker:generate-sources
```

Fonte: Documentação do GraphWalker (2025)

Para que a lógica dos testes seja feita de maneira organizada são necessários alguns recursos do Appium, como primeiramente o *driver* que irá fazer a conexão do programa com o dispositivo. É recomendado que haja uma classe separadamente que realiza a implementação deste *driver*, utilizando como insumo os dados que foram coletados na etapa de navegação. A Figura 9 exemplifica a classe para execução do *driver*.

Figura 9: Exemplo de classe para iniciação do driver Android

```
1  public class DriverRunner {
2
3      // Define driver
4      public static AndroidDriver driver;
5
6      // CREATE DRIVER
7      public static AndroidDriver createDriver() throws MalformedURLException{
8
9          // Define variables for appium device capabilities
10
11         String deviceName = "moto g(20)";
12         String deviceUdid = "0076732961";
13         String deviceOS = "Android";
14         String deviceOSVersion = "11";
15         String deviceAutomation = "uiautomator2";
16
17         // Define variables for SUT capabilities
18
19         String appPath = "Documents/GitHub/pilot-appium-graphwalker/appium/src/main/
               resources/notepad.apk";
20         String appPackage = "com.farmerbb.notepad";
21         String appActivity1 = "com.farmerbb.notepad.android.NotepadActivity";
22
23         String appiumServerURL = "http://127.0.0.1:4723";
24
25         // Setting up the capabilities
26
27         UiAutomator2Options capability = new UiAutomator2Options()
28
29         // DEVICE SETUP
30         .setDeviceName(deviceName)
31         .setUdid(deviceUdid)
32         .setAutomationName(deviceAutomation)
33         .setPlatformName(deviceOS)
34         .setPlatformVersion(deviceOSVersion)
35
36         // SUT SETUP
37         .setApp(appPath)
38         .setAppPackage(appPackage)
39         .setAppActivity(appActivity1)
40         .setNoReset(false);
41
42         // Connect Capabilities to Appium Server
43         URL url = new URL(appiumServerURL);
44         AndroidDriver driver = new AndroidDriver(url, capability);
45
46         System.out.println("App started");
47
48         return driver;
49     }
50 }
```

Fonte: Extraído do GitHub do autor.

O método *createDriver* definido na linha 7 da Figura 9 possui variáveis do tipo String com dados do aparelho utilizado no teste (linhas 11 a 15) e variáveis com dados do aplicativo (linhas 19 a 21) além da URL do servidor Appium. Então, na criação do novo driver, há a atribuição de todas estas variáveis por meio de *capabilities* do Appium (linhas 27 a 40) para então ser feita a conexão com o servidor. Este tipo de atribuição é utilizado na versão 2 do Appium.

Depois disso, cria-se um diretório com os *Page Objects*[4]. O objetivo de utilizar a estrutura de *Page Objects* é organizar a inspeção dos elementos na tela, de modo que a criação de métodos que realizam ações nestes elementos seja mais simplificada. Dessa forma, cada tela do aplicativo é representada por uma classe dedicada, que encapsula os elementos de interface e os métodos de interação correspondentes para serem utilizados nos casos de teste. Sendo assim, nesta etapa é necessário utilizar o software Appium Inspector, que tem o propósito de exibir toda a cadeia de elementos da tela com os dados que devem ser informados no código. Na Figura 10 há um exemplo resumido de uma destas classes. A inspeção de um elemento de texto (*text field*) pode ser visualizado na linha 6. Este elemento é atribuído a uma variável do tipo *RemoteWebElement*, importado de bibliotecas Selenium, para facilitar as atribuições na codificação das ações. Em seguida, na linha 15, é criado um método para inserir um texto no formato *String* neste elemento, que é passado por parâmetro.

Figura 10: Exemplo de classe de *Page Objects*

```
1
2  public class CreateNote extends BasePage {
3
4      // PAGE ELEMENTS
5
6      @AndroidFindBy(xpath="//android.widget.EditText")
7      RemoteWebElement textField;
8
9      ...
10
11     // PAGE ACTIONS
12
13     // Enter text --> Stay in the same page
14
15     public void EnterText(String text){
16         System.out.println(" Entering text: " + text);
17         textField.sendKeys(text);
18     }
19     ...
20 }
```

Fonte: Extraído do GitHub do autor.

Com os Page Objects implementados, deve ser feita a importação dos modelos. A importação é feita após a modelagem no GraphWalker Studio. A ferramenta possui uma opção de salvar, no qual é gerado um arquivo JSON com as informações de todos os modelos desenhados. Depois disso, basta copiar o arquivo para o diretório /src/main/-resources do projeto e o GraphWalker identifica os modelos automaticamente, caso os plugins Maven estejam devidamente configurados. Com o Maven, é possível executar um comando do GraphWalker para que ele gere automaticamente as interfaces dos modelos presentes no arquivo JSON importado.

Após a geração das interfaces, a lógica dos testes é codificada em classes que as imple-

[4] <https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/>

mentam. Nesta etapa é necessário utilizar a anotação de *Override* nos métodos de cada um dos vértices e arestas presente na interface, e adicioná-los a um contexto de testes do GraphWalker. Caso tenha sido desenvolvido mais de um modelo, então deve ser criada uma classe para cada modelo. A Figura 11 mostra um exemplo desta classe com alguns métodos e anotações relevantes.

Figura 11: Exemplo de classe de implementação dos testes

```
1  @GraphWalker(value = "random(edge_coverage(100))")
2  public class MainTest extends ExecutionContext implements NotepadTest {
3
4      AndroidDriver driver;
5
6      int numberOfNotes = 0;
7      int textLength = 50;
8      String text;
9      boolean isFirstNote = true;
10
11     ...
12
13     @BeforeExecution
14     public void initDriver(){
15         try {
16             driver = DriverRunner.createDriver();
17
18         } catch (MalformedURLException exc) {
19             System.out.println(exc.getCause());
20             System.out.println(exc.getMessage());
21         }
22     }
23
24     ...
25
26     @Override
27     public void v_Start() {
28         System.out.println("I'm on vertex START");
29     }
30
31     ...
32
33     @Override
34     public void e_SaveNewNote() {
35
36         System.out.println("I'm on edge SAVE NEW NOTE");
37
38         text = generateRandomText(textLength);
39         newNotePage.EnterText(text);
40         newNotePage.SaveNewNote(isFirstNote);
41         numberOfNotes++;
42         isFirstNote = false;
43         System.out.println("The number of notes was increased to: " + numberOfNotes);
44     }
45
46     ...
47
48     public String generateRandomText(int textSize){
49         System.out.println("Generating random text of length " + textSize);
50         boolean useLetters = true;
51         boolean useNumbers = true;
52         String generatedText = RandomStringUtils.random(textSize, useLetters, useNumbers)
               ;
53         System.out.println("Generated text: " + generatedText);
54         return generatedText;
55     }
56 }
```

Fonte: Extraído do GitHub do autor.

No exemplo da Figura 11 são identificados alguns blocos importantes da codificação. Primeiramente, a definição do *driver* e das variáveis de controle (linhas 4 a 9). Depois há um bloco com a anotação @BeforeExecution do GraphWalker que informa ao executor que o *driver* deve ser iniciado antes do início dos testes (linhas 13 a 22). Em sequência, há um exemplo da codificação de um vértice (linhas 26 a 29) e de uma aresta (linhas 33 a 44). Já no bloco das linhas 48 a 56, há a codificação de uma função auxiliar, que no caso é um gerador de texto, que preenche uma string de tamanho parametrizado com caracteres aleatórios, que pode ser utilizada na lógica dos testes, como presente na linha 38. Desta forma, a lógica pode ser implementada conforme o contexto de cada modelo e aplicativo.

3.6 Execução dos Testes

A configuração da execução dos testes deve ser feita na classe Main, separadamente da lógica dos testes. Nesta classe são importadas as outras classes de implementação, que então são atribuídas a um contexto de execução do GraphWalker, que pode ser utilizado também para gerenciar os resultados. O objetivo desta classe é ser enxuta, funcionando como um orquestrador, sem codificação de lógica. A Figura 12 apresenta um exemplo de implementação.

Figura 12: Exemplo de classe Main

```
1  public class Main {
2
3      public static void main(String[] args) throws IOException{
4
5          Executor executor = new TestExecutor(MainTest.class);
6          Result result = executor.execute(true);
7
8          if (result.hasErrors()) {
9              for (String error : result.getErrors()) {
10                 System.out.println(error);
11             }
12         }
13     }
14 }
```

Fonte: Extraído do GitHub do autor.

Com a classe main implementada, a execução é feita de forma simples. Preferencialmente com dois terminais simultaneamente, um deles irá executar o servidor Appium em plano de fundo, e no outro, o GraphWalker irá executar a implementação.

A execução dos testes deve ser realizada em ciclos repetidos, de forma a aumentar a confiabilidade dos resultados e reduzir o impacto de eventuais falhas transitórias (como lentidão do emulador ou perda momentânea de comunicação com o servidor do Appium). Ao final de cada ciclo, são coletadas métricas referentes ao tempo de execução, quantidade de passos cobertos no modelo, funcionalidades validadas e eventuais falhas encontradas.

Esses dados são armazenados para posterior análise comparativa entre os diferentes SUTs.

O processo garante, assim, uma execução controlada, permitindo que os experimentos sejam comparados sob os mesmos critérios metodológicos. A partir dessa sistematização, é possível avaliar de forma consistente a viabilidade do uso de TBM em aplicativos móveis, considerando esforço de implementação, cobertura obtida e comportamento dos testes.

3.7 Depuração e Análise de Falhas

A execução dos testes automatizados frequentemente resulta em falhas, que podem decorrer de duas origens distintas: *(i)* erros no próprio *script* de automação, como elementos mal identificados, inconsistências na lógica implementada ou comandos incompatíveis; e *(ii)* defeitos reais no aplicativo em teste. Assim, a etapa de depuração consiste em analisar sistematicamente cada falha registrada, de modo a de identificar sua causa e classificá-la adequadamente.

O processo inicia-se com a inspeção dos relatórios de execução e dos *logs* gerados pelas ferramentas utilizadas (Appium, GraphWalker e ADB). Cada ocorrência de falha é avaliada em relação ao contexto do modelo, à ação realizada e à resposta obtida do aplicativo. Quando a falha é atribuída ao *script*, o código correspondente é revisado e ajustado, garantindo maior robustez para execuções futuras. Já quando a falha corresponde a um comportamento inesperado do aplicativo, a ocorrência é registrada como um defeito potencial do SUT.

Para organizar esse processo de forma sistemática, foi adotado o uso de uma tabela de registro de falhas, conforme exemplificado pela Tabela 3, na qual cada ocorrência é documentada com informações como identificador único, tela e ação em que ocorreu, descrição resumida, classificação e status atual. Essa prática auxilia no rastreamento das falhas ao longo das execuções, possibilitando identificar padrões recorrentes, priorizar correções e diferenciar claramente erros de automação de defeitos do sistema.

Tabela 3: Exemplo de registro e classificação de falhas

ID	Ocorrência	Descrição da Falha	Classificação
01	v_NewNote e_Save	Botão "Salvar"não localizado	Falha no Código
02	v_NoteDetails e_Delete	Aplicativo fecha inesperadamente ao excluir nota	Falha do SUT
03	v_NoteEdition e_Discard	A opção de "Descartar"só é exibida após clicar no próximo botão	Falha na Modelagem

A depuração é realizada de forma iterativa, em ciclos que acompanham a execução dos testes. Isso significa que, após cada rodada de testes, as falhas são revisadas, classificadas

e corrigidas quando relacionadas à automação, para então serem executadas novas rodadas com o objetivo de validar os ajustes realizados. Esse processo contribui para a maturidade da suíte de testes e para a confiabilidade dos resultados.

Além disso, a análise detalhada das falhas permite gerar estatísticas relevantes, como a proporção de erros de automação em relação aos defeitos reais encontrados, o esforço investido na correção de *scripts* e o impacto dessas falhas na cobertura de teste. Tais informações são importantes para avaliar a viabilidade do TBM em cenários reais, uma vez que demonstram os custos de manutenção associados ao processo.

4 ANÁLISE DE RESULTADOS

Neste capítulo são apresentados os resultados obtidos durante a execução do estudo citado no capítulo anterior, bem como discussões e análises. O capítulo é estruturado da seguinte maneira: nas Seções 4.1 a 4.3 há a análise específica dos resultados de modo a responder às perguntas de pesquisa propostas.

4.1 QP1 — Qual a viabilidade de utilizar TBM para a automação de testes *end-to-end* em aplicativos móveis?

Para responder a esta questão de pesquisa, foi realizada uma análise qualitativa a partir da experiência prática com as ferramentas empregadas no experimento: Java + Maven, GraphWalker e Appium. A seguir, são discutidos os aspectos positivos e os desafios observados em cada tecnologia e na integração entre elas.

4.1.1 Java & Maven

Considerando a linguagem de programação Java e o gerenciador de pacotes Maven, destaca-se as seguintes observações:

- **Configuração do ambiente de desenvolvimento.** Em relação à instalação do kit de desenvolvimento Java, não houve dificuldade. A instalação é simplificada por meio de um arquivo executável fornecido pela Oracle. Já a configuração pode ser um pouco mais complexa devido a necessidade de alterações em nível de sistema operacional. Apesar disso, a ampla disponibilidade de documentação e recursos na Internet facilitaram a experiência.

- **Estrutura do projeto.** Ainda que houve dificuldade na compreensão da estrutura de pacotes e dos padrões de desenvolvimento em Java, especialmente na etapa inicial de planejamento da estrutura do projeto / repositório, o amplo suporte oferecido pelas IDEs modernas ajudou a mitigar o problema, tornando o processo de organização e construção do projeto mais intuitivo.

- **Gerenciamento de dependências.** Apesar do gerenciamento de dependências ser bastante simples no Maven, por conta das ferramentas utilizarem diversas bibliotecas, houve um problema de compatibilidade entre as versões delas e do Java. Foi necessário instalar uma versão anterior do Java para que as integrações fossem devidamente executadas, e embora este processo não seja difícil, os erros de compilação ao utilizar uma versão recente do Java que não era compatível com as bibliotecas utilizadas no projeto não foram explícitos quanto à causa, o que levou a um tempo alto de investigação, principalmente na integração com o GraphWalker. Portanto, a experiência foi consideravelmente afetada neste quesito.

4.1.2 GraphWalker

Considerando a ferramenta de TBM GraphWalker, destaca-se as seguintes observações:

- **Instalação e Configuração.** A instalação do GraphWalker foi muito simples. O website oficial da ferramenta disponibiliza dois arquivos executáveis: um para o uso do GraphWalker Studio, e outro para o uso em linha de comando, e como estes arquivos são nativamente Java, não foi necessário nenhuma configuração adicional. A instalação de dependências também foi bastante simples com o uso do Maven. Não houve dificuldades neste quesito.

- **Modelagem com GraphWalker Studio.** O GraphWalker Studio tem uma interface Web simples, porém houve uma certa dificuldade em seu manuseio no início com os atalhos de teclado. Por conta de depender da execução de um pacote Java local, apresentou queda em alguns momentos. Apesar disso, a criação de modelos foi realizada de forma satisfatória. Entretanto, foi notada a falta de algumas funcionalidades interessantes como mover arestas, por exemplo, o que pode dificultar a edição de um modelo, já que é necessário excluir a aresta e criar uma nova nesses casos.

- **Uso do GraphWalker no projeto Java.** A experiência geral dos recursos do GraphWalker no Java foi boa, principalmente na facilidade de configurar as dependências e na disponibilidade de comandos GraphWalker. Contudo, foi necessário replicar as interfaces e criar manualmente classes para a implementação da lógica dos testes. Ocorreram erros de compilação ao utilizar apenas as interfaces geradas automaticamente, o que causou retrabalho, além de levar um tempo considerável de implementação no caso do projeto possuir múltiplos modelos.

- **Execução de testes com GraphWalker.** Apesar de haver uma alta expectativa na execução dos testes automatizados por TBM, o uso do GraphWalker para este fim causou uma experiência negativa, principalmente devido a falta de documentação atualizada. É uma ferramenta poderosa que possui muitos benefícios caso os testes sejam implementados corretamente, porém quando há algum erro na implementação, foi extremamente difícil depurar e encontrar as causas de erro, o que pode gerar frustração em pessoas que são iniciantes no uso da ferramenta. Além disso, a execução depende de que todo o modelo esteja implementado, não sendo possível realizar execuções parciais, o que também dificulta o processo de depuração.

4.1.3 Appium

Considerando o *framework* de automação Appium, destaca-se as seguintes observações:

- **Instalação e Configuração.** A instalação e configuração da ferramenta pela primeira vez pode ser complexa, pois depende de pacotes Node.JS e, no caso do uso para automação de dispositivos Android, também é necessário configurar o ambiente de desenvolvimento da plataforma. Desta forma, caso o computador não possua nenhum ambiente de desenvolvimento instalado, é necessário realizar múltiplas configurações, o que pode levar bastante tempo dependendo da experiência e da máquina utilizada. Apesar disso, há muitos recursos de ajuda disponíveis na Internet para solucionar eventuais dúvidas no processo.

- **Inspeção de elementos com Appium Inspector.** A experiência de uso do Appium Inspector foi muito positiva. Houve uma pequena dificuldade inicial em configurar as *capabilities* do Appium, que são os dados do aparelho inspecionado, mas assim que configuradas, a utilização da ferramenta se tornou bastante intuitiva. Além disso, a inspeção de elementos da tela é muito rica e detalhada, o que foi um grande facilitador na implementação dos testes automatizados.

- **Utilização de bibliotecas Appium e Selenium nos Page Objects.** O *driver* UiAutomator2 do Appium já disponibiliza diversas ações que são muito úteis e otimizadas para o ambiente Android, como toque na tela, gestos para voltar e etc. Porém também é possível utilizar diversas outras ações que não estejam disponíveis pelo *driver*, devido a compatibilidade com o Selenium. Um exemplo de uso do Selenium foi a ação de pressionar e segurar, que não estava disponível diretamente pelas bibliotecas do Appium. A experiência foi muito boa, principalmente devido a facilidades do ambiente de desenvolvimento Java, como as sugestões de *auto-complete*, que listam as ações disponíveis do *driver* de forma fácil em tempo de implementação.

- **Integração Appium & GraphWalker.** A integração das duas ferramentas foi transparente por conta da estrutura de Page Objects adotada. Desta forma, os comandos Appium ficaram localizados exclusivamente nos Page Objects e a implementação da lógica dos testes no GraphWalker apenas utiliza métodos destas páginas, tornando o código simples e limpo.

4.1.4 Resposta à QP1

Para tornar a análise dos relatos informados mais objetiva, foram elaboradas duas tabelas: A Tabela 4 aborda a experiência no uso das ferramentas com níveis qualitativos seguindo níveis de satisfação, enquanto a tabela 5 aborda a dificuldade no uso das ferramentas. Ambas as tabelas abordam critérios comuns ao desenvolvimento do estudo, que foram avaliados de forma independente para cada ferramenta.

Tabela 4: Experiência no uso das ferramentas

Critério	Java	GraphWalker	Appium
Instalação	Neutra	Boa	Neutra
Utilização	Boa	Neutra	Boa
Integração	Neutra	Boa	Boa
Solução de Problemas	Ruim	Ruim	Neutra
Execução	Boa	Neutra	Boa
Geral	Neutra	Neutra	Boa

Tabela 5: Dificuldade de uso das ferramentas

Critério	Java	GraphWalker	Appium
Instalação	Intermediária	Fácil	Difícil
Utilização	Intermediária	Intermediária	Intermediária
Integração	Fácil	Difícil	Intermediária
Solução de Problemas	Intermediária	Muito Difícil	Intermediária
Execução	Fácil	Difícil	Fácil
Geral	Intermediária	Difícil	Intermediária

Com base na combinação entre a experiência de uso apresentada na Tabela 4 e a dificuldade relatada na Tabela 5, é possível elencar pontos positivos e negativos. As principais vantagens observadas são na execução e na integração entre as ferramentas, que apesar de certa dificuldade, apresentaram a melhor experiência. Já a solução de problemas foi o critério pior avaliado, apresentando tanto alta dificuldade quanto experiência ruim.

Portanto, uma resposta geral à QP1 indica uma viabilidade intermediária. As ferramentas são muito boas e mais vantajosas quanto menor o escopo e maior a experiência de uso, e se torna mais complexa quanto maior o escopo e menor a experiência. Estes aspectos reforçam a curva de aprendizado acentuada relatado na revisão bibliográfica.

4.2 QP2 - Qual o esforço envolvido na implementação dos testes?

Para responder a esta questão de pesquisa, foram coletados dados quantitativos durante o andamento do trabalho. Os dados coletados foram: Tempo gasto, métricas referente aos modelos, métricas de codificação, quantidade de problemas encontrados e tempo de execução dos testes. Cada um deles é descrito em detalhes a seguir.

4.2.1 Métricas de tempo gasto

As métricas de tempo gasto foram coletadas utilizando uma planilha de controle, na qual foram anotados os horários de início e fim de cada etapa da metodologia, para cada aplicativo. A partir desta planilha, foram contabilizados os minutos gastos em cada etapa,

conforme exibido na Tabela 6. No caso do Notepad as métricas de tempo são estimativas, já que foi um experimento inicial utilizado para elaborar a metodologia do trabalho. Já no caso do Gallery e File Manager, os dados foram coletados durante a execução.

Tabela 6: Tempo gasto por etapa por aplicativo

Atividade	**Notepad**	**Gallery**	**File Manager**
Navegação inicial	15 minutos	62 minutos	10 minutos
Modelagem	360 minutos	159 minutos	20 minutos
Codificação	590 minutos	875 minutos	245 minutos
Execução e Depuração	240 minutos	–	82 minutos
Correção de falhas	2400 minutos	–	97 minutos
Total	3605 minutos	1096 minutos	454 minutos

A análise do tempo gasto informado na Tabela 6 revela que o Notepad apresentou o maior esforço total uma vez que esse aplicativo foi o primeiro SUT utilizado e serviu como base para a construção e refinamento da metodologia do estudo. Dessa forma, boa parte do tempo registrado corresponde não apenas à implementação do TBM, mas também à configuração inicial e aprendizagem das ferramentas. Um ponto crítico notado na etapa de correção de falhas foi referente a um problema de compatibilidade entre as bibliotecas do GraphWalker e versões específicas do JDK, que foi a falha que levou mais tempo para ser solucionada. Em relação aos outros aplicativos, o Gallery teve o escopo mais complexo, dado que houve a decisão de utilizar mais de um modelo na etapa de modelagem. A dificuldade na solução de erros de compilação na etapa de codificação impediram a execução e depuração dos testes automatizados, consequentemente não foram registrados tempos para estas etapas. Já o File Manager apresentou o menor tempo total, uma vez que teve o escopo e modelagem reduzidos, o que diminuiu significativamente o esforço de modelagem, codificação e correção de falhas. Esses resultados sugerem que o tempo gasto para aplicar TBM em aplicativos móveis varia conforme a complexidade do SUT e tende a diminuir quanto menor o escopo ou maior a familiaridade com as ferramentas.

4.2.2 Métricas dos modelos

As métricas dos modelos foram extraídas diretamente do GraphWalker após a etapa de modelagem. Na Tabela 7 são apresentados os números de vértices e arestas dos modelos desenvolvidos para cada aplicativo.

Tabela 7: Métricas dos modelos

Aplicativo	Modelo	Nº de vértices	Nº de arestas
Notepad	NotepadTest	5	12
Gallery	MainFlow	7	11
	AnimatedImageOptions	7	13
	StaticImageOptions	6	12
	VideoOptions	7	13
	VideoPlayer	5	10
File Manager	FileManager	4	7

A análise das métricas dos modelos apresentada na Tabela 7 representa a evolução da etapa de modelagem ao longo da aplicação da metodologia nos três aplicativos. O Notepad, primeiro SUT utilizado, apresentou um modelo simples, com cinco vértices e doze arestas, cobrindo as funcionalidades de criar nota, editar nota, e apagar nota. Em seguida, ao modelar o Gallery, optou-se por uma estratégia mais elaborada, estruturando o comportamento do aplicativo em um modelo principal (*MainFlow*) complementado por modelos auxiliares representando funcionalidades específicas, por meio de Estados Compartilhados. Essa decisão resultou em um conjunto maior de vértices e arestas, alinhado à riqueza funcional do aplicativo. Por fim, no File Manager, buscou-se um modelo mais enxuto, no qual foi decidido modelar um escopo reduzido de funcionalidades, consequentemente levando a um menor número de vértices e arestas.

Uma observação importante sobre a análise das métricas dos modelos é que as decisões de modelagem foram totalmente definidas e executadas manualmente pelo primeiro autor, o que pode ser uma ameaça à validade dado a pouca experiência com a atividade.

4.2.3 Métricas de código

As métricas de código foram extraídas por meio de um *plugin* da interface de desenvolvimento IntelliJ, chamado *Metrics Reloaded*[5]. Na Tabela 8 são apresentadas as métricas em quantidade de linhas de código, conforme a estrutura de pacotes do repositório. No contexto do Java, os pacotes são utilizados para agrupar classes relacionadas de forma lógica, facilitando a modularização do código, a separação por responsabilidades e a manutenção do projeto. Para este trabalho, foram desenvolvidos pacotes para: *Page Objects*, interfaces de modelos do GraphWalker, implementação da lógica dos modelos, e um pacote raíz em relação a estes três que aborda as configurações de *driver* do Appium e os parâmetros de execução do GraphWalker.

[5]Disponível em: <https://plugins.jetbrains.com/plugin/93-metricsreloaded>

Tabela 8: Métricas de código

Componente	Pacote	Linhas de código	Arquivos
Notepad	com.notepad.po	364	7
	com.notepad.impl	237	1
	com.notepad.model	55	1
	com.notepad	113	2
Gallery	com.gallery.po	606	10
	com.gallery.impl	595	5
	com.gallery.model	222	5
	com.gallery	56	1
File Manager	com.filemanager.po	172	3
	com.filemanager.impl	103	1
	com.filemanager.model	30	1
	com.filemanager	113	2
Main	com	51	1
Total	13	2717	40

A análise das métricas de código apresentadas ta Tabela 8 evidencia diferenças substanciais sobre a estratégia de implementação adotada para cada aplicativo. O Gallery apresenta a maior quantidade de linhas de código, sendo o pacote de implementação dos *Page Objects* o que apresenta o maior volume, devido à maior quantidade de funcionalidades abordadas pelo modelo, o que aumenta o número de telas inspecionadas. Já o Notepad, embora mais simples em termos funcionais, registra um volume intermediário de código, pois em uma das análises de falha, foi notado que não havia sido considerado um estado específico para quando a lista de notas está vazia, o que causava falha na inspeção dos elementos da tela. Para correção, foi decidido dividir a página de lista de notas em dois *Page Objects* distintos, um para quando a lista está vazia e outro para quando a lista está populada, o que também causou aumento da codificação da lógica dos testes no pacote de Implementação, já que o modelo inicial não foi alterado. O File Manager, por sua vez, possui o menor número de linhas de código, resultado do escopo reduzido de funcionalidades abordadas nos testes automatizados.

4.2.4 Falhas apontadas pelos testes

A coleta dos dados de falhas nos testes ocorreu durante a etapa de Depuração, após o código ser compilado pela primeira vez. Nesta etapa foram registrados dados de duas atividades distintas: *(i)* Descrição de falhas encontradas, conforme descrito pela metodologia na etapa de depuração, baseando-se em logs de execução e *(ii)* Descrição de causas e soluções para cada falha, após análise e tentativas de resolução. Estes dados foram levantados para cada aplicativo:

Notepad: Na Tabela 9 são apresentadas as falhas apontadas após a primeira compilação dos testes do Notepad. Em sequência, na Tabela 10 são documentadas as causas

e soluções analisadas para cada falha.

Tabela 9: Falhas encontradas nos testes do Notepad

ID	Ocorrência	Descrição da Falha	Classificação
01	Início	Testes não executam com comando GraphWalker	Falha na configuração da ferramenta
02	Após o início	App fecha inesperadamente após 30 segundos de teste	Falha no código de teste
03	v_ListNotes e_SelectNote	Falha ao selecionar nota quando a lista está vazia	Falha no código de teste
04	v_Start (Reexecução)	App fecha inesperadamente ao rodar múltiplos testes	Falha na configuração da ferramenta

Tabela 10: Causas e soluções de falhas do Notepad

ID	Causa	Solução
01	Incompatibilidade entre a versão do GraphWalker e a versão do Java	Downgrade do JDK para a versão 21
02	Driver mal configurado	Ajuste no timeout do driver e refatoração do código para que os métodos reaproveitem o driver já iniciado
03	Lógica para verificação se a lista de notas está vazia não foi implementada após a criação da primeira nota	Separação da lista de notas em dois Page Objects e criação de variável de controle booleana indicando se a lista está vazia
04	Uso alto de memória	Reiniciar o servidor Appium no início de um novo teste

A falha de ID 01 foi a que levou maior tempo para ser resolvida, devido aos logs não serem explícitos quanto à causa. Já as falhas de ID 02, 03 e 04 foram resolvidas mais rapidamente, facilitadas pelos logs adicionados no código de testes.

Gallery: Na Tabela 11 são apresentadas as falhas apontadas após a primeira compilação dos testes do Gallery. Em sequência, na Tabela 12 são documentadas as causas e soluções analisadas para cada falha.

Tabela 11: Falhas encontradas nos testes do Gallery

ID	Ocorrência	Descrição da Falha	Classificação
01	Início	Testes não executam com comando GraphWalker	Falha na configuração da ferramenta

Tabela 12: Causas e soluções de falhas do Gallery

ID	Causa	Solução
01	GraphWalker não encontra o contexto de teste dos modelos atribuídos no código	Não foi solucionado

Similar ao que ocorreu com o Notepad, a primeira tentativa de execução dos testes do Gallery causaram erro apontado pelo GraphWalker. O erro apontado foi mais explícito quanto à causa, indicando que o GraphWalker não conseguiu identificar o contexto dos testes dos modelos atrubuídos, porém após despender um tempo considerável implementando sugestões propostas através de busca na Internet, a falha prosseguiu sem ser solucionada. Uma observação referente à busca é que a maioria dos códigos encontrados são antigos, ou seja, têm uma implementação que não é suportada pela versão mais atual da ferramenta.

File Manager: Na Tabela 13 são apresentadas as falhas apontadas após a primeira compilação dos testes do File Manager. Em sequência, na Tabela 14 são documentadas as causas e soluções analisadas para cada falha.

Tabela 13: Falhas encontradas nos testes do File Manager

ID	Ocorrência	Descrição da Falha	Classificação
01	v_NewFolder e_DeleteFolder	App não exibe opção de deletar a pasta quando está dentro da pasta	Falha no código de teste
02	v_NewFolder e_RenameFolder	A ação de voltar -> pesquisar -> pressionar e segurar a primeira opção não funciona para pasta, apenas para arquivo	Falha no código de teste
03	v_NewFile e_RenameFile	Automação não está buscando pelo nome do arquivo	Falha no código de teste
04	v_Start (Reexecução)	App fecha inesperadamente ao rodar múltiplos testes	Falha no Appium (Uso de memória)

Tabela 14: Causas e soluções de falhas do File Manager

ID	Causa	Solução
01	Elemento inspecionado equivocadamente	Ajuste no método de deletar pasta no Page Object para retornar à página anterior e adicionar a ação de pressionar e segurar
02	Elemento inspecionado equivocadamente	Remover a lógica de pesquisar e clicar no primeiro item e ajustar para buscar diretamente o elemento que contém o nome da pasta
03	Elemento com nome do arquivo não foi inspecionado	Adicionar inspeção de elemento com o nome do arquivo e remover a lógica de busca pelo primeiro arquivo
04	Uso alto de memória	Reiniciar o servidor Appium no início de um novo teste

Com uma implementação mais simplificada, a execução dos testes do File Manager ocorreu sem problemas de integração, ao seguir os passos da metodologia. Todas as falhas encontradas na primeira execução foram referentes à lógica codificada, que não foram difíceis de serem solucionadas. O único problema que afetou a execução foi o uso alto de memória relatado pela falha de ID 04, causando a interrupção da conexão com o servidor Appium em alguns momentos, o que consequentemente desabilita o *driver* que mantém o aplicativo aberto.

4.2.5 Métricas de execução

A métrica de execução coletada foi o tempo de execução dos testes após a resolução das falhas apontadas na etapa de depuração. Foram registrados os tempos de 10 execuções do conjunto de teste, e então foi calculada a média simples do tempo de execução para cada aplicativo. A Tabela 15 apresenta esta métrica após a resolução das falhas de código apontados pelas Tabelas 9 e 13 dos aplicativos Notepad e File Manager, respectivamente. As métricas do aplicativo Gallery não foram coletadas, já que as falhas deste não foram solucionadas, impedindo a execução dos testes automatizados.

Tabela 15: Tempo de execução dos testes

ID	Notepad	File Manager
01	01 min 24 seg	01 min 13 seg
02	01 min 17 seg	01 min 38 seg
03	00 min 57 seg	01 min 00 seg
04	02 min 13 seg	01 min 13 seg
05	01 min 15 seg	01 min 44 seg
06	02 min 12 seg	01 min 01 seg
07	00 min 46 seg	01 min 26 seg
08	01 min 11 seg	00 min 55 seg
09	02 min 11 seg	01 min 31 seg
10	01 min 04 seg	01 min 56 seg
Média	01 min 27 seg	01 min 21 seg

A partir dos dados de tempo de execução apresentados na Tabela 15 é possível notar a variação de tempo entre os testes, conforme esperado do GraphWalker, que foi configurado para percorrer caminhos aleatórios até uma cobertura de 100% de arestas. Para comparação, o aplicativo Notepad levou 46 segundos de execução do teste mais rápido e 02 minutos e 13 segundos no teste mais longo, apresentando variação percentual de 189,13%. No caso do aplicativo File Manager, o teste mais rápido levou 55 segundos e o mais longo, 01 minuto e 56 segundos, apresentando variação percentual de 110,91%.

4.2.6 Resposta à QP2

A partir das métricas coletadas, foi desenvolvida uma tabela que relaciona o esforço com o nível de conhecimento em TBM, a complexidade do modelo, a complexidade da inspeção de elementos e a complexidade da codificação da lógica dos aplicativos.

Tabela 16: Análise do esforço na implementação dos testes

Critério	Notepad	Gallery	File Manager
Conhecimento em TBM	Baixo	Baixo	Baixo
Complexidade da Modelagem	Baixa	Alta	Baixa
Complexidade da Inspeção	Alta	Baixa	Baixa
Complexidade da Lógica	Baixa	Baixa	Intermediária
Complexidade da Depuração	Alta	Muito Alta	Baixa
Esforço	Intermediário	Alto	Baixo

Com a análise indicada pela Tabela 16, conclui-se que o esforço na implementação dos testes automatizados com TBM em aplicativos móveis é mais alto quanto menor a experiência prévia com a abordagem e maior a complexidade dos modelos e da inspeção dos elementos dos aplicativos. Como não foi possível a análise com um maior conhecimento em TBM, é possível afirmar que a redução do escopo de testes, que proporciona modelos menores, aliada com a estratégia de priorizar as complexidades de ações e validações para

o nível de código ao invés de explicitar no modelo, pode diminuir o esforço na utilização de TBM, tornando mais viável a implementação dos testes, como reportado nas métricas do aplicativo File Manager.

4.3 QP3 - Quais os desafios observados no uso do TBM?

Para responder a esta questão de pesquisa, foram registradas avaliações gerais na experiência do estudo como um todo. Estas avaliações são descritas elencando-se os tópicos principais de vantagens, desvantagens e lições aprendidas.

4.3.1 Vantagens

Nesta seção são discorridas as vantagens identificadas no decorrer do trabalho, nas etapas de modelagem, codificação, depuração e execução, respectivamente.

Modelagem: Uma grande vantagem da modelagem foi a abstração dos casos de teste. Desta forma, com a adoção de CUJ, os fluxos principais de uso dos aplicativos foram modelados de forma rápida e simplificaram consideravelmente a codificação dos testes automatizados.

Codificação: Além da abstração de casos de teste provida pelo TBM, o uso de *Page Objects* também demonstrou uma grande vantagem, pois além de permitir uma melhor organização do código, também permitiu mais simplificação na lógica desenvolvida e transparência na integração entre os recursos do Appium e do GraphWalker, já que a implementação dos métodos de cada ferramenta ficaram segregados.

Depuração: Uma vantagem do GraphWalker foi a disponibilidade de comandos Maven que permitem gerar arquivos detalhados de logs de execução e falhas, apesar dos arquivos não terem sido particularmente úteis na depuração dos testes implementados do aplicativo Gallery devido à falta de documentação disponível com causas e soluções de problemas. No caso do Appium, a depuração, principalmente para os testes do aplicativo Notepad, ocorreu com maior facilidade, pois além da documentação disponível, muitos dos erros enfrentados já haviam sido identificados e solucionados por outros usuários, que compartilharam seus relatos e soluções na Internet. Isso evidencia como a presença de documentação aliada à existência de uma comunidade ativa traz vantagens significativas na resolução de problemas e no uso eficiente da ferramenta.

Execução: A maior vantagem na execução dos testes automatizados por TBM são as configurações de caminho possíveis pelos *generators* do GraphWalker. Com eles é possível alterar completamente a estratégia para percorrer os caminhos do modelo sem que seja necessário alterar a lógica da implementação, apenas um parâmetro na anotação GraphWalker, o que é um grande facilitador que pode ser adaptável dependendo do contexto de testes.

4.3.2 Dificuldades

Nesta seção são discorridas dificuldades identificadas no decorrer do trabalho, nas etapas de modelagem, codificação, depuração e execução, respectivamente.

Modelagem: O uso de TBM implica em algumas tomadas de decisão na etapa de modelagem, por exemplo na granularidade de casos cobertos pelo modelo e número de modelos utilizados para representar o sistema. Quanto maior a granularidade e quantidade de modelos, mais complexa é a implementação, como aconteceu com o aplicativo Gallery. Apesar da estratégia de dividir os modelos com estados compartilhados parecer ser um facilitador, não se demonstrou algo simples de ser implementado quando há pouca experiência no uso do GraphWalker.

Codificação: Aplicativos desenvolvidos de forma mais completa e com elementos próprios tendem a ter um melhor comportamento em testes automatizados devido a inspeção dos elementos na tela. Os aplicativos Gallery e File Manager tiveram uma integração bastante satisfatória pois a maioria dos elementos utilizados no escopo definido tinham ID próprio e recursos de acessibilidade, apenas alguns poucos elementos eram nativos Android e apresentaram caminhos relativos que dependem do nome, o que tende a apresentar problemas na replicação dos testes conforme o idioma do aparelho, que pode causar falhas nos testes. Já o Aplicativo Notepad teve uma integração bastante difícil, pois é um aplicativo muito simples e todos os elementos da tela são genéricos. Alguns elementos tem o mesmo caminho relativo, mesmo sendo completamente diferentes entre uma tela e outra, e este mapeamento pode inclusive gerar falsos positivos nos testes, ou gerar ramificações de caminhos não mapeados pelo modelo, o que aumenta a dificuldade de análise em casos de falha.

Depuração: A maior dificuldade na depuração foi encontrada no uso do GraphWalker e esteve relacionada à escassez de documentação e recursos atualizados na Internet. Como havia pouca experiência prévia no uso da ferramenta, mesmo os logs detalhados não ajudaram a identificar soluções para os erros encontrados especificamente para o aplicativo Gallery, que possuía modelos mais complexos. Além disso, também houve o problema relacionado à compatibilidade entre as versões do GraphWalker e as versões do JDK, que não foram encontradas de forma direta, o que acentuou o esforço empregado na etapa de depuração do aplicativo Notepad, que foi a primeira experiência com a ferramenta.

Execução: A limitação de recursos computacionais da máquina causou o *reset* do servidor Appium diversas vezes durante a execução dos testes por conta do uso de memória. Desta forma, não foi possível executar múltiplos testes em sequência. Somente foi possível fazer múltiplas execuções forçando o reinício do servidor Appium após cada execução dos testes, o que mitigou as vantagens do teste automatizado. A mesma limitação também impactou na execução de testes por meio de emulador, portanto todos os tempos de execução relatados foram referentes à execução quando conectado a um aparelho físico.

4.3.3 Lições Aprendidas

Nesta seção são discorridas as lições aprendidas no estudo:

- Para a primeira implementação, um escopo reduzido com apenas um modelo se mostrou mais adequado, pois facilita a resolução de possíveis problemas de implementação e integração. Após uma primeira execução com sucesso é possível incrementar o modelo e a lógica dos testes conforme necessário.
- A estratégia de deixar o modelo o mais enxuto possível e aplicar as lógicas de validação no código facilita a manutenção, principalmente quando há pouca experiência prévia com TBM.
- O mapeamento dos elementos da tela dependem diretamente de como o SUT foi implementado, tornando a automação mais fácil caso os elementos estejam desenvolvidos com IDs e campos acessíveis, e mais complexas caso existam apenas elementos genéricos ou nativos Android que são atribuídos por texto de exibição.
- Execuções dos testes automatizados em dispositivo físico utilizam menos recursos computacionais, porém podem apresentar complicações na conexão com o servidor Appium, além de apresentar maiores chances de falhas por conta da comunicação.
- Limitações de recursos computacionais impactam negativamente a execução dos testes, pois os *drivers* Android exigem um uso de memória considerável.
- Assim como em todo teste automatizado, com o TBM também é necessário avaliar criticamente as falhas, pois nem sempre é um *bug* do SUT, especialmente em casos de *crash* (aplicativo fecha inesperadamente). Este problema ocorreu diversas vezes durante o trabalho, e apenas na execução dos testes automatizados. Isso acentuou a importância da análise de uso dos recursos computacionais pelas ferramentas utilizadas.
- O uso de um tipo de log que imprime o estado atual e variáveis de controle facilita muito a análise em casos de falha, especialmente quando a falha é causada pelo código de testes.
- Em uma das tentativas de resolver a falha dos testes do aplicativo Gallery, foram coletados logs extensivos dos erros apresentados pelo GraphWalker, e carregados em agentes de inteligência artificial generativa, como ChatGPT, Gemini e Claude. Mesmo com a quantidade de dados coletada e o contexto do projeto, nenhum dos agentes foi capaz de propor uma solução assertiva para o problema.

5 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo a avaliação das ferramentas GraphWalker e Appium na implementação de Teste Baseado em Modelo na automação de testes *end-to-end* em aplicativos móveis em nível inicial de aprendizado da abordagem de TBM. Como principais resultados, a avaliação de uso do GraphWalker se mostrou balanceada. A experiência na instalação, integração da ferramenta com o projeto Java e o uso da interface de modelagem foram positivas, enquanto que a depuração se mostrou difícil. Já a avaliação de uso do Appium se mostrou mais positiva no contexto geral. A etapa de instalação mostrou-se mais complexa, mas teve a dificuldade mitigada pela quantidade de conteúdo sobre a ferramenta na Web. Além disso, o uso do software de inspeção e a codificação tiveram avaliação muito boa, sendo o ponto de maior desvantagem o uso de recursos computacionais na integração do servidor Appium com os *drivers* Android na execução dos testes automatizados.

Portanto, para uma melhor experiência no uso destas ferramentas, é necessário levar em conta a curva de aprendizado elevada, que é citada em estudos sobre TBM na literatura. Considerando a pouca experiência com a abordagem, os melhores resultados foram obtidos com a criação de modelos enxutos e escopo reduzido de testes, conforme resultados analisados com o aplicativo File Manager.

Como possíveis trabalhos futuros, pode ser interessante avaliar a escalabilidade da metodologia com o incremento do modelo, avaliando como se daria o esforço da implementação de testes em mais funcionalidades dado um aplicativo que já possui automação com TBM. Além disso, pode ser interessante a avaliação de um processo de execução em pares das etapas de navegação e modelagem, de forma a validar a confiabilidade do escopo de teste escolhido e dos modelos produzidos. Outras propostas interessantes seriam a avaliação de diferentes ferramentas de TBM e o uso de aplicativos de outros escopos como SUT.

Referências

1 GSMA. *The Mobile Economy 2025*. 2025. Acessado em: 06/09/2025. Disponível em: <https://www.gsma.com/solutions-and-impact/connectivity-for-good/mobile-economy/>.

2 SILVA, H. N. et al. A mapping study on mutation testing for mobile applications. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 32, n. 8, p. e1801, 2022.

3 PRESSMAN, R.; MAXIM, B. *Engenharia de software - 9.ed.* [S.l.]: McGraw Hill Brasil, 2021. ISBN 9786558040118.

4 ORSO, A.; ROTHERMEL, G. Software testing: a research travelogue (2000–2014). In: *Future of Software Engineering Proceedings*. [S.l.]: Association for Computing Machinery, 2014. p. 117–132.

5 KONG, P. et al. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, v. 68, n. 1, p. 45–66, 2019.

6 NETO, A. D. et al. Improving evidence about software technologies: A look at model-based testing. *IEEE Software*, v. 25, n. 3, p. 10–13, 2008.

7 UTTING, M.; LEGEARD, B. *Practical model-based testing: a tools approach*. [S.l.]: Elsevier, 2010. ISBN 9780123725011.

8 KRAMER, A.; LEGEARD, B. *Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level*. [S.l.]: John Wiley & Sons, 2016. ISBN 9781119130017.

9 SILVA, H. N. da et al. Evaluating the impact of different testers on model-based testing. In: *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2018. p. 57–66.

10 GRAPHWALKER. *Website do projeto GraphWalker*. 2025. Acessado em 06/09/2025. Disponível em: <https://graphwalker.github.io/>.

11 GUDMUNDSSON, V. et al. Model-based testing of mobile systems – an empirical study on quizup android app. *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, v. 208, p. 16–30, maio 2016. ISSN 2075-2180. Disponível em: <http://dx.doi.org/10.4204/EPTCS.208.2>.

12 MATTA, S.; GAROUSI, V. Mbtmodelgenerator: A software tool for reverse engineering of model-based testing (mbt) models from clickstream data of web applications. *arXiv preprint arXiv:2506.08179*, 2025.

13 FARTO, G. d. C. *Uma contribuição ao teste baseado em modelo no contexto de aplicações móveis*. Dissertação (Mestrado) — Universidade Tecnológica Federal do Paraná, 2016.

14 STUDIO, A. *Website do portal de desenvolvimento Android*. 2025. Acessado em 06/09/2025. Disponível em: <https://developer.android.com/studio>.

15 FARTO, G. de C.; ENDO, A. T. Reuse of model-based tests in mobile apps. In: *Proceedings of the XXXI Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2017. p. 184–193.

16 APPIUM. *Website do Appium*. 2025. Acessado em 06/09/2025. Disponível em: <https://appium.io/>.

17 SINGH, S.; GADGIL, R.; CHUDGOR, A. Automated testing of mobile applications using scripting technique: A study on appium. *International Journal of Current Engineering and Technology (IJCET)*, v. 4, n. 5, p. 3627–3630, 2014.

18 FLORA, H. K.; CHANDE, S. V. A review and anaysis on mobile application development processes using agile methodlogies. *International Journal of Research in Computer Science*, White Globe Publications, v. 3, n. 4, p. 9, 2013.

19 COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Addison-Wesley Professional, 2009. ISBN 9780321579362.

20 LABUSCHAGNE, A.; INOZEMTSEVA, L.; HOLMES, R. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 821–830. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3106288>.

21 KARHU, K. et al. Empirical observations on software testing automation. In: IEEE. *2009 International Conference on Software Testing Verification and Validation*. [S.l.], 2009. p. 201–209.

22 LINARES-VáSQUEZ, M.; MORAN, K.; POSHYVANYK, D. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 399–410.

23 NÉMETH, G. Á. Teaching model-based testing. *Teaching Mathematics and Computer Science*, v. 18, n. 1, p. 1–17, 2020.

24 BERNARDINO, M. et al. Systematic mapping study on mbt: tools and models. *IET Software*, Wiley Online Library, v. 11, n. 4, p. 141–155, 2017.

25 GAROUSI, V. et al. Model-based testing in practice: An experience report from the web applications domain. *Journal of Systems and Software*, Elsevier, v. 180, p. 111032, 2021.