CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS

Recredenciado pela Portaria Ministerial nº 1.162, de 13/10/16, D.O.U. nº 198, de 14/10/2016
AELBRA EDUCAÇÃO SUPERIOR - GRADUAÇÃO E PÓS-GRADUAÇÃO S.A.

RIDNEY GOMES BARBOSA DA SILVA QUEIROZ

ADOÇÃO DE AUTOMAÇÃO DE TESTES NO DESENVOLVIDO MÓVEL EM FLUTTER: UM PROCESSO PARA GARANTIR QUALIDADE E INTEGRIDADE DO CÓDIGO

Palmas – TO

2022

RIDNEY GOMES BARBOSA DA SILVA QUEIROZ

ADOÇÃO DE AUTOMAÇÃO DE TESTES NO DESENVOLVIDO MÓVEL EM FLUTTER: UM PROCESSO PARA GARANTIR QUALIDADE E INTEGRIDADE DO CÓDIGO

Projeto de Pesquisa elaborado e apresentado como requisito parcial para aprovação na disciplina de Laboratório de Criação do curso de bacharel em Sistemas de Informação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Esp. Fábio Castro Araújo.

Palmas – TO

2022

RIDNEY GOMES BARBOSA DA SILVA QUEIROZ

ADOÇÃO DE AUTOMAÇÃO DE TESTES NO DESENVOLVIDO MÓVEL EM FLUTTER: UM PROCESSO PARA GARANTIR QUALIDADE E INTEGRIDADE DO CÓDIGO

Projeto de Pesquisa elaborado e apresentado como requisito parcial para aprovação na disciplina de Laboratório de Criação do curso de bacharel em Sistemas de Informação pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Esp. Fábio Castro Araújo.

Aprovado em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. Esp. Fábio Castro Araújo

Orientador

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. Me. Jackson Gomes de Souza

Centro Universitário Luterano de Palmas – CEULP

_____

Profª. Esp. Fernanda Pereira Gomes

Centro Universitário Luterano de Palmas – CEULP

Palmas – TO

2022

RESUMO

Queiroz, Ridney Gomes Barbosa da Silva. **Adoção de automação de testes no desenvolvimento móvel em *Flutter*: Um processo para garantir qualidade e integridade do código**. 2022. 00 f. Projeto Tecnológico (Graduação) – Curso de Sistemas de Informação, Centro Universitário Luterano de Palmas, Palmas/TO, 2022.

A crescente demanda por novos aplicativos aumenta a cada dia pelos consumidores. O conceito 80/20 nas soluções digitais aliada ao *mobile first* tem despertado o interesse em melhor interface e experiência do usuário para este segmento de mercado. O presente trabalho tem como objetivo realizar testes de renderização de *Widgets* para verificar o funcionamento do código de componentes, sua resposta a interações, e seu comportamento visual em um aplicativo desenvolvido em Flutter. Levando em consideração os objetivos e relevância de teste aplicado ao contexto do cenário pretendido, visto que correspondam ou não à resposta para qual foram aplicados. Capturando todas exceções registradas com feedback ágil assegurando reduzir esforços de teste manuais em ambientes de constante evolução ou mudança. Este trabalho propôs o uso de elementos essenciais de acordo com os requisitos do negócio, software e compõem as etapas de planejamento, implementação, execução do projeto e avaliação dos critérios de saída dos processos anteriores. Como resultados, as variáveis do objeto de estudo consistiram em simular interação, comportamento e dados no teste, sem alterar o modelo ou interface do aplicativo, a fim de se obter a garantia do funcionamento de comportamento nos cenários possíveis de aceitação e falha de testes e análise das saídas nos resultados encontrados. Assim é válido ressaltar o potencial uso e extensão desta pesquisa como ferramenta de trabalho para outros projetos, este não discutiu problemas da simulação, análise de performance e desempenho em ambiente de testes local, trabalhos futuros podem avaliar e diagnosticar.

Palavras-chave: Automação de Teste; *Widget*; Integridade; Flutter; *Mobile*. Aplicativo.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TDD	Desenvolvimento Guiado por Testes (do inglês, *Test-Driven Development*)
XP	Programação Extrema (do inglês, *Extreme Programming*)
PRU	Princípio da Responsabilidade Única (do inglês, *Single Responsibility Principle*)
OO	Orientação a Objetos

SUMÁRIO

1 INTRODUÇÃO

O planejamento e construção do *software* para solução de um problema é cada vez mais indispensável à sua necessidade de mercado. Erros ocorrem, pois, o software requer manutenção, evolução e planejamento no ciclo de vida do processo. Monitorar e evitar risco é necessário para evolução e planejamento do *software*. No mercado competitivo a gestão assertiva atende estes requisitos e garante satisfação. Sobre o planejamento evolução e manutenção Sommerville (2011, p. 21) afirma que:

> Durante o estágio final do ciclo de vida (operação e manutenção), o software é colocado em uso. Erros e omissões nos requisitos originais do software são descobertos. Os erros de programa e projeto aparecem e são identificadas novas necessidades funcionais. O sistema deve evoluir para permanecer útil. Fazer essas alterações (manutenção do software) pode implicar repetição de estágios anteriores do processo.

A arquitetura do projeto construída e pensada para diferentes contextos promovem a evolução e a flexibilidade na reutilização de código, utilizar padrões arquiteturais úteis garantem manutenção, escalabilidade e integridade estrutural do software compõem um conjunto de atividades de grande impacto a longo prazo no sucesso do projeto.

Elementos do software não inspecionados que atestam o comportamento e funcionalidades impactam em deixar de prover novas estratégias de solução e gera custo de manutenção que é mais oneroso que o custo de desenvolvimento devido a refatoração do código.

Omitir etapas, processos e auditorias na qual garantem que os artefatos cumpram com seu objetivo desde a iniciação do planejamento até o ciclo de entrega, gera custo de manutenção oneroso maior que o do desenvolvimento devido a reescrita de código.

Processo não envolve o desenvolvimento de nova funcionalidade do sistema, mas correção de código errado e não funcional. Com isso o desperdício de energia física, mental, financeira impede a inovação e prioriza o foco da sobrevivência da empresa. Após estar inserida no mercado mantendo o *software* legado com funcionalidade essencial fica passível de falhas. Software de qualidade prevê inconsistências e defeitos, assegura satisfação do cliente, custo benefício e competitividade. Sobre funcionalidade Sommerville (2011, p. 22) afirma que:

> Cada incremento ou versão do sistema incorpora alguma funcionalidade necessária para o cliente. Frequentemente, os incrementos iniciais incluem a funcionalidade mais importante ou mais urgente. Isso significa que o cliente pode avaliar o sistema em um estágio relativamente inicial do desenvolvimento para ver se ele oferece o que foi requisitado.

A automação de testes tem como objetivo reduzir a quantidade de testes manuais, entretanto é necessário avaliar quais casos serão suscetíveis a testes automáticos. Segundo Pfleeger, (Shari Lawrence Pfleeger. 2004, v.2, p. 360), diz que: “Durante os testes, você pode tomar cuidado ao considerar todos os casos de teste, automatizar os testes quando for adequado e assegurar que o seu projeto considere todos os perigos possíveis.”.

Desempenho de automação e resultado qualitativo aplicado a metodologia ao tipo de teste servirá para documentar e validar as entregas no prazo previsto na fase final de entrega do produto. A reutilização de componentes é essencial para menor uso de recursos e ganho de escalabilidade. Gestão eficiente consiste na automação de testes cobrindo todas etapas de entrega no ciclo de vida do produto.

No desenvolvimento em Flutter todo Widget é o componente que contém as configurações, instruções e estado que definirá a interface para o usuário, Google (2022). O Flutter possui a biblioteca de renderização gráfica Skia em linguagem Dart. Bibliotecas são coleções de subprogramas utilizados para programação. A renderização é feita em 2D sem comprometer qualidade e desempenho com o processo de mobile first.

Há três árvores de *Widgets,* a árvore *Widget,* a árvore *Element* e a árvore *RenderObjects*–. *C*ada árvore em particular tem sua função e a combinação resulta na renderização de uma interface para o usuário no *Flutter*. Um ou vários componentes podem não realizar sua função, com isso há a necessidade de garantir que os fluxos e componentes estejam funcionando no cenário de contexto do aplicativo. Para validar a integridade da UI (User Interface) é necessário o teste de Widget.

Testar *Widget* envolve o teste de várias classes, o *Widget* deve responder as interações do usuário e a aparência do aplicativo. O framework não precisa de nenhuma configuração adicional explicitamente, o pacote flutter_test vem adicionado nas dependências do projeto contida no arquivo pubspec.yaml. Porém é necessário prover uma classe para o ciclo de vida *Widget.* Vantajoso pois testa um componente e seus filho diferente do teste de unidade, e o ambiente é mais simples que a configuração de um teste de integração contínua.

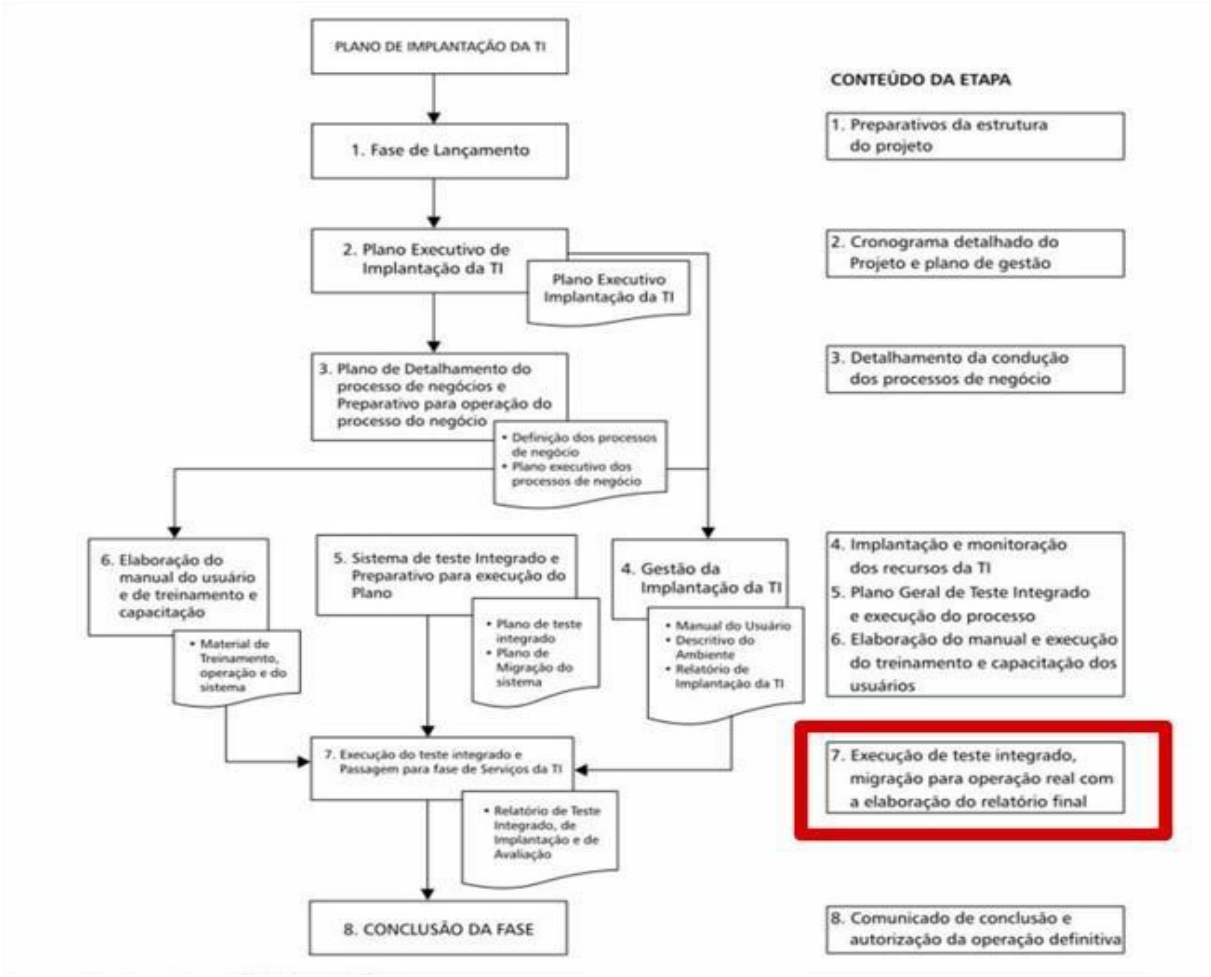
O teste de *Widget* prevê a detecção de cenários onde o comportamento do aplicativo agirá em resposta ao tipo de simulação de interação do usuário. Um *Widget* em UI pode ser testado sem necessidade que todo aplicativo esteja pronto. O desenvolvimento em Flutter é multiplataforma, então é esperado que determinadas bibliotecas de terceiros possuam falhas de comportamento e renderização. O comportamento nativo padrão esperado tem o propósito de cobrir todas as falhas dos recursos oferecidos pelo framework para as versões de

aplicativo Mobile, Web, Desktop. O propósito deste trabalho é realizar testes de renderização, integridade de comportamento de Widgets em um aplicativo *mobile* desenvolvido em *Flutter*.

2 REFERENCIAL TEÓRICO

A visão geral sobre testes no processo e ciclo de vida de desenvolvimento de software consiste em diversas etapas, desde a engenharia de requisitos, modelagem, arquitetura e implementação e testes. Conforme Paula Filho (2009), entrega incremental e iterativa é o resultado do processo do ciclo de vida que está de acordo em todas as instâncias do que foi requisitado pelo cliente.

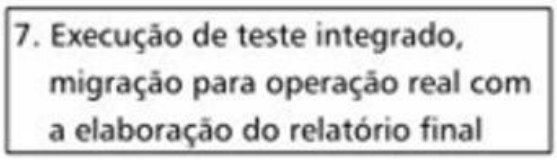
Figura 1. Estrutura geral da fase de implantação da TI.



Fonte: (AKABANE, 2011 p.151 apud INOUE, 2008 p.83)

A Figura 1 detalha as atividades da implantação e ajustes dos recursos de TI, com objetivo de produzir artefatos, documentos com conteúdo contendo informações inerentes ao modelo de negócio.

Figura 2. Execução de teste integrado.



Fonte: (AKABANE, 2011 p.151 apud INOUE, 2008 p.83)

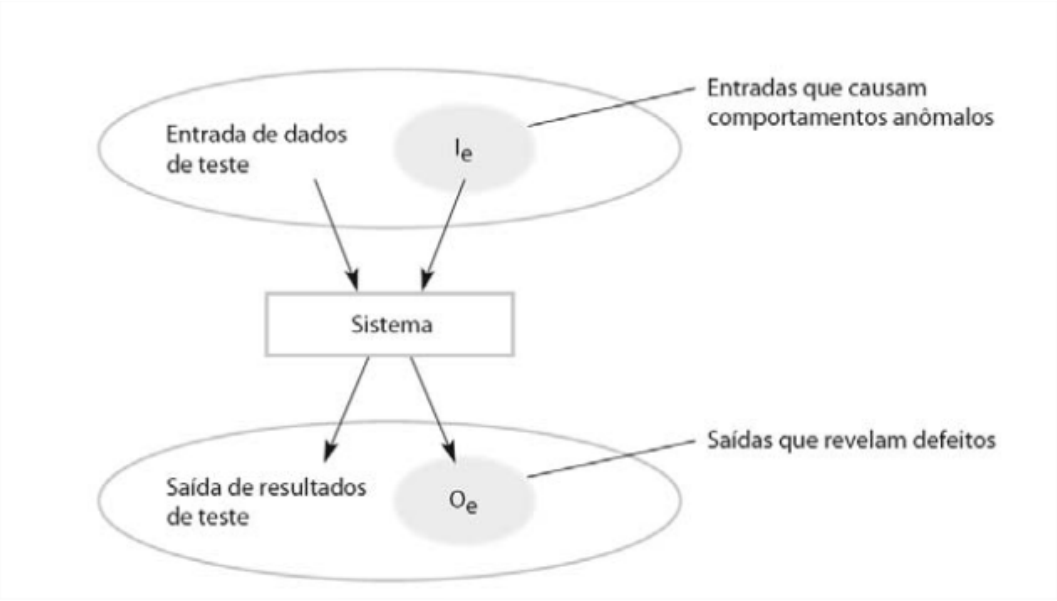
A Figura 2 ilustra a atividade da área de testes e funcionalidades do software, a responsabilidade do processo é garantir que estão atendendo as expectativas do usuário, antecipando problemas que podem acontecer na entrega para o cliente.

2.1 TESTES

O processo de desenvolvimento de software compõe a área de teste como parte do ciclo de vida do software. O código, programa ou sistema necessita ser testado para que garanta que o produto tenha qualidade. O objetivo do teste não é provar que o software está correto mesmo que não detecte mais defeitos. Há a ilusão de que não há mais defeitos logo o software está livre deles. Contudo só não é possível detectar os defeitos se não foram feitos os testes corretos. Testes de software podem mostrar apenas a presença de defeitos, nunca sua ausência (DIJKSTRA, et al., 1972). Se o teste é realizado e não se encontra falhas é porque o teste não está sendo bom o suficiente. O objetivo do teste é encontrar a existência de falhas.

No início é fácil o encontro de falhas, em dado momento é deixado de encontrar as falhas mais fáceis e então é necessário ser mais criterioso com técnicas mais avançadas para encontrar defeitos escondidos.

Figura 3. Um modelo de entrada e saída de teste de programa.



Fonte: Sommerville (2011)

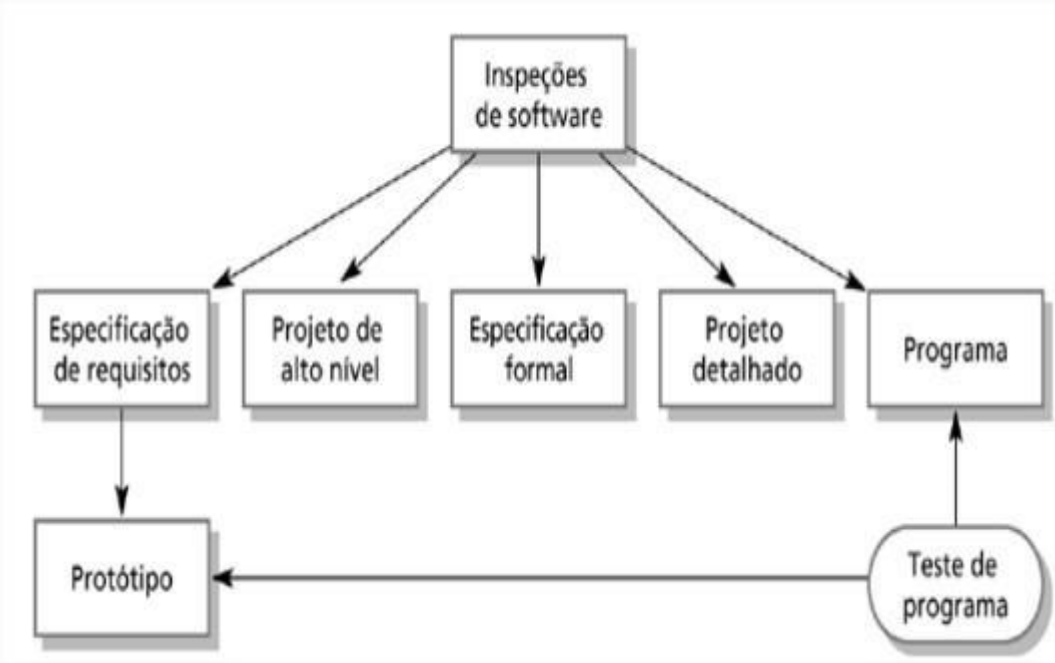
A Figura 3 ilustra a ideia do objetivo do teste de tentar encontrar possíveis entradas de dados que ao serem processadas pelo sistema gera saídas que mostram a falha do sistema.

Testar todas as entradas de dados possíveis se torna inviável. Todavia um bom teste é aquele que se propõe observar quais as entradas são potenciais prováveis para mostrar que o software possui falhas. A uma característica dentro da ideia de teste é a atividade dinâmica de (V & V). Validação e verificação de software, ambas verificam se o software produzido possui o mínimo nível possível de qualidade. Há uma diferença entre elas. A validação checa a inexistência de erro e se está atendendo as expectativas do cliente. Assegura a validação das necessidades propostas nesta atividade.

A atividade de verificação inclui testes e averigua o cumprimento dos requisitos funcionais e não para construção do software. Aplicada em qualquer artefato ou atividade do projeto, código fonte, diagramas, etc.

Atividades estáticas de V&V são inspeções e revisões. Um caso de uso pode ser revisto, aberto e lido, assim como determinado trecho de código ao inspecionar à procura de defeitos, não é uma atividade dinâmica de execução.

Figura 4. V & V estática e dinâmica.



Fonte: Engenharia de Software, 8ª. edição. Capítulo 22

A Figura 4 demonstra o comportamento estático de inspeções e dinâmico no teste do programa. Conjunto e princípios com foco em levar a otimização de processos e boas práticas de programação que vão gerar qualidade no produto final no software.

O uso da metodologia traz benefícios como autonomia, análise e integra o código desenvolvido à área de testes, revisa por meio de técnicas externa e interna a entrega para partes interessadas permitindo criar código mais tolerante a falhas.

2.2 DESENVOLVIMENTO MULTIPLATAFORMA COM FLUTTER

O *Flutter* é uma tecnologia sdk lançada pela equipe da Google, foi lançada em 2017 com intenção de trazer para o desenvolvedor de aplicativos, o ambiente de desenvolvimento multiplataforma.

O *framework* compila de forma nativa, buscando resolver problemas para atender as necessidades de plataformas devido a tendência de grandes quantias de usuários destes, no mercado. A oferta desta tecnologia traz a possibilidade do desenvolvimento de um único código de forma nativa, ganhando performance. O framework compila de forma nativa e com código fonte singular. Atende a resolução de problemas com versatilidade, com fácil aprendizagem e agilidade.

Testes Caixa Preta (*Black Box*)	Examina a funcionalidade e a aderência aos requisitos, em uma visão externa ou do usuário, sem conhecimento da lógica, código do componente testado.
Testes Caixa Branca *(White Box*)	Classifica cláusulas de código, a lógica interna do componente codificado, as configurações e outros elementos técnicos.
Testes Unitários	Teste aplicado há partes menores, componentes de código criado.
Testes de Sobrevivência	Avaliam a capacidade do *software* em continuar operando mesmo quando software ou hardware fica inoperante para funcionar.
Testes de Regressão	Visa garantir que o *software* permaneça intacto depois de alterações e testes realizados.
Testes de Usabilidade	Visa verificar o nível de facilidade de uso do *software* pelos usuários.
Testes de Interoperabilidade	Avaliam as condições de integração com outros *softwares* e /ou ambientes.
Testes de Carga	Avaliam a resposta de um *software* sob uma pesada carga de dados, repetição de certas ações de entrada de dados, entrada de valores numéricos grandes, consultas complexas a base de dados.
Testes de Segurança	Validar a capacidade de proteção do *software* contra ameaças de acessos internos ou externos não autorizados.
Teste de Compatibilidade	Visa verificar a capacidade do *software* executar em determinado ambiente, plataforma, *hardware*, sistema operacional ou rede.
Teste de Renderização	Testa a renderização de uma interface para o usuário no *software*.
Teste de Integração	Teste de integração, permite testes de vários dispositivos, utilizando firebase test lab, parecido com escrita de testes de Widgets.
Teste de Comportamento	Visa testar e garantir o comportamento correto e esperado, em Flutter se responde a interação do usuário.

Testes Alfa:	Visa ser feito entre o término do *software* e a sua entrega.
Testes Beta	Teste feito quando o *software* está concluído.

A linguagem de programação também criada pela Google é chamada de Dart, sua sintaxe se assemelha às linguagens Java e C Sharp, para quem possui familiaridade com estas linguagens populares.

A proposta visa o desenvolvimento *Web, Desktop e Mobile*, *Flutter* 3 é a mais nova versão do kit de desenvolvimento de *apps*, lançada dia 11 de Maio de 2022 com a possibilidade de em um único código para seis plataformas, oferecendo produtividade.

2.2.1 TIPOS DE TESTES EM FLUTTER

A caracterização dos testes de software é dividida em vários tipos. Diante de vários tipos, cada teste possui uma característica específica necessária para uma finalidade. O teste de funcionalidade verifica a acurácia, a funcionalidade, ou seja, o teste está indo direto ao ponto está fazendo exatamente o que é proposto.

Há uma série de outros testes que não são somente funcionais. O teste de tempo de resposta, checa se o software está fazendo o que é necessário fazer em um tempo de resposta razoável esperado para o usuário.

Além deste há o teste de processamento que pode estar relacionado ao desempenho, processando vários números de pedidos em determinado tempo. A definição da construção do *software* e estratégia de teste podem ser desenvolvidas através de diferentes abordagens, segundo (Hetzel, 1988), (Beizer,1995), (Myers, 1979) os tipos de testes são:

Tabela 1:Tipos de testes segundo, (Hetzel, 1988), (Beizer,1995), (Myers, 1979)

2.2.1.1 TESTES NA PLATAFORMA MOBILE

O uso de dispositivos móveis está presente em diversas situações e aplicações no cotidiano dos usuários. Eles utilizam seus aplicativos no dispositivo que mais preferem, portanto, para que os aplicativos ofereçam uma experiência livre de erros.

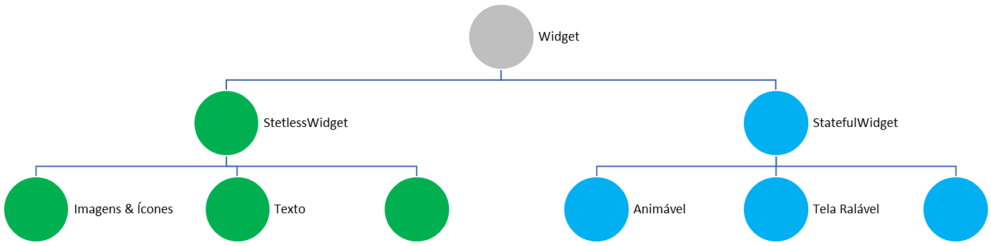
Há os Mobile Web Apps, páginas web que adaptam o site para dispositivos móveis. Independentemente do tamanho de tela, versão do sistema operacional e recursos disponíveis no aplicativo testar é necessário.

2.2.1.2 TESTE DE WIDGETS E RENDERIZAÇÃO

Os *Widgets* possuem uma relação de herança de pais e filhos na árvore de componentes contendo elementos estruturais como botões, menus e objetos de renderização.

Widgets são divididos em duas categorias de estados, Stateless (Sem estado) e Stateful (Com estado). O estado de um Widget refere-se aos valores que podem ser mutáveis ou não, em outras palavras a condição das variáveis a esse Widget.

Figura 5. Árvore de *Widgets*.



Fonte: Flutter (2022, tradução nossa).

A árvore de componente na Figura 5 ilustra a hierarquia mencionada, a documentação diz sobre a mudança de estado durante o ciclo de vida do aplicativo. Um Widget com estado pode estar incluso em um sem estado e reciprocamente o inverso.

> "Quando o estado de um widget muda, o widget reconstrói sua descrição, que a estrutura diferencia em relação à descrição anterior para determinar as alterações mínimas necessárias na árvore de renderização subjacente para a transição de um estado para o próximo." (Google, 2022).

Se uma variável for declarada dentro de uma classe em um *Widget* codificado como Stateless, esse campo pertencerá a uma classe imutável, aplicado a telas com conteúdo estático, pois seu estado não é alterado. Ou seja, se algum atributo é modificado em tempo de execução, esta é uma forma equivocada de se usar um componente Stateless violando o componente que não tem todos os valores finais.Então, se há uma variável que é necessária alteração usa-se Stateful o Widget com estado é redesenhado a alteração é dinâmica, consequentemente todos filhos na árvore deste serão também.

Então, se há uma variável que é necessária alteração usa-se Stateful o Widget com estado é redesenhado a alteração dinâmica, consequentemente todos filhos na árvore deste serão também.

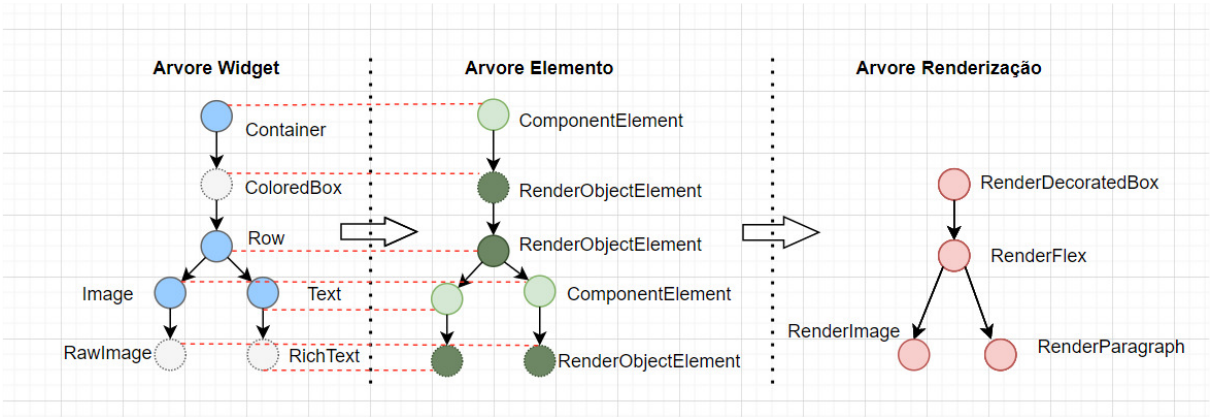
Figura 6. Estilo declarativo de programação de interface.

Fonte: Flutter (2022, tradução nossa).

A Figura 6 demonstra como o Flutter renderiza o aplicativo de forma eficiente cada Widget. O framework governa através do conceito de três árvores de objetos, a árvore de Widget é a primeira usada para configurações da interface do usuário (IU).

No contexto tudo é um componente sejam eles um texto, botões ou imagens são Widgets com isso é possível modificar as propriedades de cada componente para optar como será exibido na tela.

Figura 7. Layout e renderização.



Fonte: Flutter (2022, tradução nossa).

Ao centro da imagem a árvore de elemento (*Element Tree)* Ilustrada na Figura 7, administra e remodela todas as árvores e instância específica de um *Widget*. Os elementos desta árvore têm uma referência ao *Widget* da primeira árvore. É o que gerencia a atualização

e alteração da interface do usuário, o elemento controla tudo gerenciando o ciclo de vida dos *Widgets.*

Os elementos são responsáveis por comparar as diferenças dos *Widgets* entre a primeira e segunda árvore permitindo a composição sobre a herança de classes. O *framework* consiste na combinação de *Widgets* menores e até personalizado para reuso máximo de recurso para o quanto menos possível for criar objetos.

A árvore de renderização (*Render Object*). É composta pelo processo de construir e reconstruir *Widgets,* que é um objeto na árvore de renderização.

Este é o modo como o *Flutter* gerencia o estado e está fazendo por meio de uma *API* (*Application Programming Interface)* Interface de Programação de Aplicações.

Assim é como representa a mudança combinando os conceitos das árvores anteriores para tornar a renderização da interface do usuário.

A árvore de renderização e de elemento permanece a mesma, otimizando a quantidade de trabalho e está definindo declarativamente como será sua interface de usuário.

2.2.1.3 TESTES DE WIDGETS E INTEGRIDADE

Teste de unidade e *Widgets* feitos em classe e funções é um ideal a fim de se obter informações de como cada *widget* funciona isoladamente, contudo essas partes devem ser testadas como um todo.

A documentação oficial aborda com limitados detalhes sobre os testes, com tópicos breves embora sejam ótimos a documentação cita vários sistemas *Flutter* de integração contínua para automatizar a distribuição dos aplicativos e executar testes e compilações.

Dentre os sistemas de integração contínua citados na documentação o *Codemagic* e *Bitrise* são maduros em torno do ecossistema *Flutter*, a empresa responsável pelo *Codemagic* é a empresa Nevercode.

Possibilita especificar o tipo de dispositivo na qual deseja executar os testes, pode ser um emulador, simulador ou dispositivo físico integrado com a plataforma (*AWS Device Farm*) ou *Firebase Test Lab*. Garantir a integridade da escrita e execução dos testes de integração são suficientes para suprir e complementar os testes de unidade e *widgets*.

Alinhar este tipo de configuração de automação de teste, localmente e em nível de produção utilizando as plataformas mencionadas anteriormente garante que o código passará nos testes em vários dispositivos, tamanhos de tela, localidade e versões de sistemas operacionais.

O processo de teste de recursos executado durante o processo de desenvolvimento é útil pois lidar com testes manuais está sujeito e propenso a lidar com erros manuais e várias jornadas do usuário.

Embora haja valor em teste de *widgets*, integração há alguns riscos à medida que o código se torne muito extenso dado a lógica de modelo de negócio, poluindo o ambiente de produção, pode ser pouco confiável e lento na melhor hipótese.

2.2.1.4 TESTES DE WIDGETS E COMPORTAMENTO

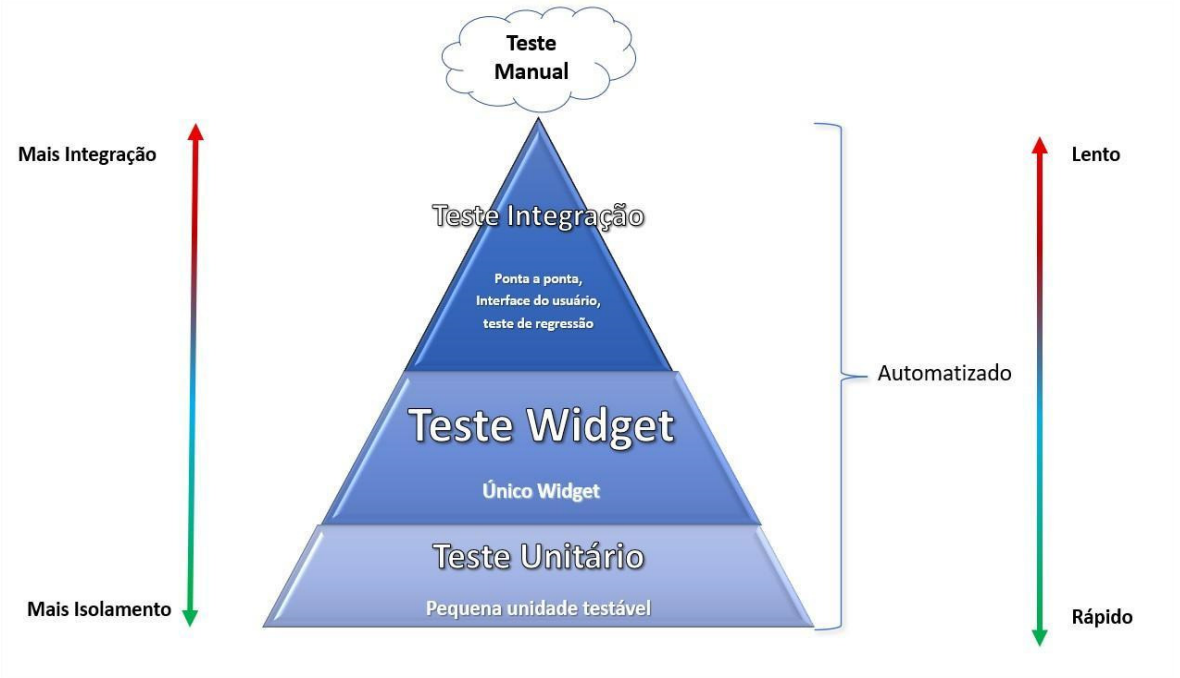
Etapas envolvidas no teste de comportamento compõem o processo e definição do objetivo do teste, se o teste for escrito corretamente, para entrada de dados esperada, produzirá a saída esperada tornando o processo previsível e sem dúvidas.

Há sinais que indicam um teste ruim. O indicador de um teste mal produzido é quando o objetivo do teste não é claro ou possui vários objetivos.

Os testes devem concentrar-se em um comportamento prudente, é importante como os testes apresentam os resultados, pois a ordem do teste afetará se ele passará. Segundo Martin Fowler (1999) afirma que “A refatoração é o processo de modificar um sistema de software de modo que não altere o comportamento externo do código, embora melhore a sua estrutura interna. É uma maneira disciplinada de reorganizar o código, minimizando as chances de introduzir bugs.” (p.14).

O objetivo do teste de *Widget* é verificar se a interface do usuário de cada *Widget* se parece e se comporta da forma esperada. O primeiro passo é executar em um ambiente de teste, então é necessário executar algumas ações que mudará seus estados. Por fim, é preciso confirmar a nova estrutura de visão da interface, com base nas alterações feitas no estado.

Figura 8. Pirâmide de automação de testes. Adaptada, tradução nossa.



Fonte: Crispin e Gregory (2009).

A Figura 8 ilustra a hierarquia da pirâmide de testes adaptada ao *framework Flutter* contendo os tipos de testes que um aplicativo deve ter. Em sua base há o teste unitário no qual contém a menor unidade testável representa a parte mais larga. O teste de *Widget* acima na escala com cobertura maior e por seguinte teste de integração na ordem, todos automatizados.

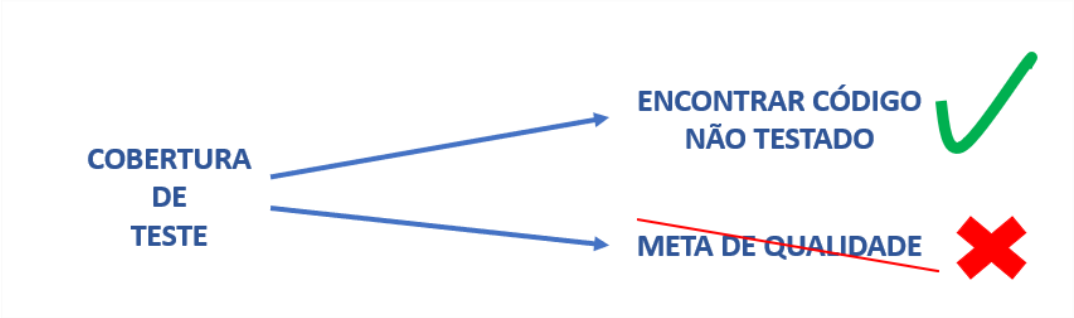
Ao subir cada nível da pirâmide os testes tornam-se menos isolado e mais integrado e o custo de tempo torna-se maior, o teste manual é útil na aplicação quando os automatizados não forem possíveis de aplicar, excluí-lo do conjunto de testes significa omitir a necessidade da cobertura deste tipo de teste.

2.2.2 METRICAS E COBERTURAS

O objetivo da cobertura de testes é identificar onde o código desenvolvido não foi testado, a fim de se obter a cobertura máxima da lógica de negócio desenvolvida e recursos do produto, cabe à equipe de desenvolvimento definir a cobertura dos recursos, requisitos e riscos do projeto.

Identificar áreas de testes que são inúteis faz parte do mecanismo de cobertura de testes, estes casos podem ser eliminados e produzir menos código para ambiente de testes, assim, é possível extrair métricas de valor do código que por sua vez também do produto através de relatórios de cobertura.

Figura 9. Cobertura de testes.



Fonte: Adaptado de : https://www.martinfowler.com/bliki/TestCoverage.html
Acesso em 25 Abr.2022, tradução nossa.

O teste de unitário possui um alcance mais restrito, pois testa uma única função ou classe, o teste de *Widget* possui uma cobertura maior que o teste de unidade, pois no *Flutter* tudo são *Widgets.*

Há hierarquia da árvore de *Widgets* onde alguns dependem de outros contendo diversos níveis de *Widgets* relacionados entre si. A cobertura do teste deve ser capaz de verificar todas as instâncias dos componentes filhos.

Widgets são a composição de uma tela ou representação do leiaute de um botão, ou textos. O alcance e cobertura de teste de integração tende a ser mais complexo, pois teste o sistema por completo ou grande parte deste.

	Unidade	**Widget**	**Integração**
Confiança	Baixo	Alto	Mais Alto
Custo de manutenção	Baixo	Alto	Mais Alto
Dependências	Pouca	Mais	Maior
Velocidade de Execução	Rápido	Rápido	Lento

Tabela 2. Cobertura de testes. Fonte: Adaptado de: https://docs.flutter.dev/testing . Acesso em 23 Abr. 2022, tradução nossa.

A tabela 2 ilustra as propriedades de testes com especificidade e cobertura de cada um destes.

Os critérios de *Widgets* são maiores, pois há uma dependência e maior integração com demais componentes, conservando sua velocidade de execução de um único *Widget* ou fluxo de *Widgets*.

O teste de *Widget* entre os demais torna-se mais viável e rápido, fazendo o uso de AVD (*Android Device Manager*). Para criar e configurar ambientes virtuais fornecendo *feedback* em tempo real sobre renderização, comportamento e integridade.

A documentação oficial do *framework Flutter* deixa claro a estratégia de cada tipo de teste e cobertura, porém não informam sobre seu desempenho sob carga.

2.2.2.1 TESTES COM FIREBASE TEST LAB

O *Firebase Test Lab* é uma plataforma para testar os aplicativos em dispositivos reais em um ambiente em nuvem, garantindo que o aplicativo funcione para todos os usuários. Há uma infinidade de dispositivos e versões de sistemas operacionais a serem testados em diversas situações. Além desta também há a *AWS Device Farms* no mercado.

É possível executar o teste de instrumentação conhecido como teste de integração no qual mostra a maior quantidade de automação e fornece percepções sobre como o aplicativo funciona.

O teste *Game Loop* para desenvolvedores de jogos em diferentes plataformas e dispositivos, voltado para renderização em 2D.

Teste de unidade e *Widgets,* ambos são de níveis diferentes e mais baixos que o teste de integração, eles podem testar e lidar com complexidades apenas para desenvolvedores e testadores. Também são menos complexos de executar e possuem menor custo, não há aplicativo real sendo construído.

O teste de Integração testa o fluxo de interação da interface do usuário no aplicativo, com perspectiva de um real usuário pois imita seu comportamento.

Embora seja efetivamente eficiente na cobertura de descoberta de erros que afetam os usuários, fornece pouca informação sobre quais erros realmente aconteceram. Em razão disso, porque este tipo de teste por si só não é suficiente, mas um complemento para os de unidade e *Widget.*

Para realizar o teste de integração é preciso configurar o *Google Cloud Project* que é associado ao Firebase, mesmo apesar de estar trabalhando apenas com *Firebase Test Lab*, a empresa possui integração de seus consoles e plataformas.

3 METODOLOGIA

Nesta seção são apresentados os materiais e métodos da pesquisa aplicada de natureza quanti-qualitativa com objetivo realizar testes de *software* em um aplicativo desenvolvido em *Flutter*. Esta seção está organizada da seguinte maneira: Na seção 3.1 são descritos os materiais que são utilizados no desenvolvimento do trabalho; na seção 3.2 são descritas as atividades propostas para execução do trabalho.

3.1 MATERIAIS

Os materiais e tecnologias utilizadas para desenvolvimento do trabalho fornecem a comunicação e viabilizam a visão macro e micro da cobertura de testes, além de agregar valor. Os materiais são:

Visual Studio Code: É uma IDE, editor de código fonte que é executado no Windows, macOS e Linux, extensível e personalizável com opção de mudar idioma, *plugins,* temas e com suporte a diversas linguagens de programação segundo Microsoft (2022).A ferramenta foi usada no projeto para possibilitar integração à plataforma de versionamento, navegação e codificação do aplicativo. Auxiliou com funcionalidades de refatoração, suporte à depuração, definição de pontos de interrupção para inspecionamento de objetos que foram desenvolvidos.

Android Studio: É uma IDE da empresa JetBrains com foco em desenvolvimento de programas e sistemas na linguagem Dart. Esta ferramenta possui duas versões: uma paga e uma comunitária segundo a Google (2022). o que tange a configuração de ambiente forneceu o recurso de emulador do SDK Android, a ferramenta dispôs da função para criar um dispositivo virtual definindo para qual plataforma utilizar, móvel ou web, personalizando o tamanho da tela entre vários aparelhos *Mobile,* escolha da versão do *Android*.

Linguagem Dart: É uma linguagem de programação criada pela Google, fortemente tipada orientada a objetos, multiparadigma, versátil e utilizada no desenvolvimento de aplicativos *Mobile, Desktop* na criação de *Scripts.* A linguagem necessitou ser avaliada pelo comitê técnico TC52 da associação da indústria dedicada à padronização de sistemas de informação e comunicação segundo Ecma International (2019). A associação certificou que a linguagem Dart usa todos os padrões e é aceita em todos navegadores segundo consta em seu portal. O uso da linguagem no projeto foi com a finalidade de escrita dos *Widgets* testados.

Para isso, a linguagem utiliza mais de uma plataforma, as quais são: *Mobile* e *Web*, cada uma possui recursos necessários para desenvolver em cada um dos diferentes ambientes. O ambiente do projeto é feito na plataforma *Mobile* onde o aplicativo foi desenvolvido.

Flutter: É um conjunto de ferramentas *User Interface* portáteis, criado pela Google. É um *framework* apresentado ao público em 2015 e que passou alguns anos se aperfeiçoando nas versões *Release Preview,* sendo a primeira versão instável o Flutter 1.0, tendo sido lançado em 2018. *framework* é desenvolvido com as linguagens C, C++, Dart e *Skia Graphics Engine* que é uma biblioteca gráfica também desenvolvida pela Google, com ele é possível criar aplicações híbridas, multiplataformas e manter a performance nativa. *Flutter Web Support* segundo Google (2022). *Flutter* foi escolhido para atender o projeto pela praticidade em ser multiplataforma e unir os conceitos de criar aplicações de forma rápida com o conceito *Stateful Hot Reload.* O conceito contribui com o princípio da atualização automática do aplicativo, quando o arquivo de projeto é salvo. Houve a possibilidade de customização de *Widgets,* além disso foi integrado às diferentes IDEs, editores.

Biblioteca Mock: É uma biblioteca que simula classes com dados falsos através de funções de testes, principalmente onde o uso de recursos externos é necessário, como o consumo de uma API externa. Um dos princípios do teste de *Widget* e unidade é a velocidade de execução, portanto, o uso da biblioteca *Mock* decorreu para simular o retorno de dados falsos locais de maneira ágil e eficiente. Escrita de teste de *Widget* na qual o consumo de uma *API Web* com conexão direta para um determinado servidor demanda custo de tempo, ferindo o princípio destes tipos de testes. O uso da biblioteca *Mock* no projeto descreveram sua necessidade, pois o arquivo de teste permitiu escrever um teste para cada condição ou cenário. Posteriormente foi feita a execução destes testes, onde estas funções ou classes que dependem do serviço *Web* ou de uma base de dados segundo Google (2022).

Firebase: É uma plataforma que pode ser classificada como BaaS (*Backend As A Service*). É um serviço de computação em nuvem que está ligada ao desenvolvimento e soluções de aplicativos para dispositivos *Mobile* em todo o mundo. ornece aos desenvolvedores a solução de conectar as aplicações *Mobile* e *Web* à serviços na nuvem a partir de *APIs* e *SDKs*, permitindo que os programadores concentrem o desenvolvimento com a experiência do usuário *UX* ao invés de lidar com a infraestrutura e o *backend* e faz parte da infraestrutura no pacote de produtos GCP (*Google Cloud Plataform*), segundo Google (2022).

Dentre os produtos disponíveis na plataforma há o *realtime database*, onde contém opção de escolha para hospedagem do aplicativo, permitindo a criação de *snippets* especificando no script o produto do *database*, após isto é possível criar várias regras para

proteger seus dados. O uso do Firebase no projeto proveu uma maneira diferente de tipo de teste, pois a plataforma possui bibliotecas que implementam a *API* de dada biblioteca do Firebase e simula seu comportamento ao invés de testes locais com *Mock*.

3.2 MÉTODOS

Nesta seção é apresentado os métodos e organização de cada etapa do fluxograma do processo de pesquisa para desenvolvimento e teste de um módulo do aplicativo desenvolvido.

Figura 10. Representação do fluxo de etapas.

A pesquisa foi realizada conforme ilustra detalhes na imagem da Figura 10. Na imagem cada etapa é sequenciada e discriminada.

A primeira etapa demonstrada na Figura 10 abrange a conformidade do mapeamento dos objetivos do trabalho. O desenvolvimento inicial deu-se início com a reunião com os professores especialistas de domínio para levantamento de requisitos para entendimento do contexto. Os principais assuntos deste trabalho consistiram na análise e definição do escopo sobre a plataforma e as tecnologias existentes para aplicação. Foi realizada a busca em sites, livros, artigos, monografias, documentação oficial que orientou e serviu como referência bibliográfica para identificação e definição dos artefatos, materiais no qual resultaram na compreensão e aplicabilidade de práticas adequadas para fundamentar a base desta pesquisa.

Posteriormente a segunda etapa apresentou os principais conceitos para determinar os testes e desenvolvimento, requisitos implementados no projeto. Para isso foi feito o diagnóstico acerca do problema que tem como objetivo, como realizar testes de integridade e comportamento de *Widgets* em um aplicativo desenvolvido em Flutter. Após a atividade de elicitação dos dados para definição dos requisitos para desenvolvimento de algumas telas do aplicativo móvel, na qual possibilita união de universidades, empresas e alunos egressos que desejam ingressar no mercado.

Para o estágio de desenvolvimento se fez necessária a opção de cadastro para o usuário inserir informações pessoais, como nome do egresso, nome da instituição, ano de conclusão entre outros dados utilizados facilitando o planejamento para próximas telas. Eventos de ações necessárias para funcionamento e usabilidade são usadas na tela de perfil e formulário, permitindo o cadastro, alteração do mesmo, exclusão, navegação e mensagens de validação. Tais funcionalidades são essenciais para que atenda os objetivos específicos que envolve a próxima etapa.

Por consequência sucedeu o início da terceira etapa de escrita de casos de testes para atender a necessidade do fluxo do usuário, onde foi feita a análise do cenário de execução de testes para verificar se os *Widgets* cumpriram o objetivo de agir da forma esperada e simularam a construção da interface do usuário.

A partir destes métodos tornou-se evidente o relatório de testes gerados mostrando que traz vantagem para descobrir a cobertura de testes sem a perda da qualidade, encerra com dados obtidos de todo o processo de trabalho.

4 RESULTADOS E DISCUSSÃO

O trabalho tem como resultado a criação do app e os *Widget's* utilizados para o ambiente *mobile* e os cenários de teste onde o primeiro é "O teste não deve rolar formulário", o segundo "O teste ecrã menor de tela" e terceiro "Erro de lista não fornecida". Como foi descrito nos capítulos anteriores, buscou-se aplicar boas práticas no **desenvolvimento** do aplicativo. Os resultados deste tipo de teste específico norteiam o desenvolvedor sobre o entendimento em diferentes casos e condições de teste. A diversidade de combinação oferecida garante e atendem o funcionamento de um ou alguns comportamentos dentro do aplicativo.

O Grupo de Testes tem mais exatidão em encontrar desvios e erros para validar as regras de negócio. A compatibilidade de teste dentro de um grupo de testes torna o teste mais

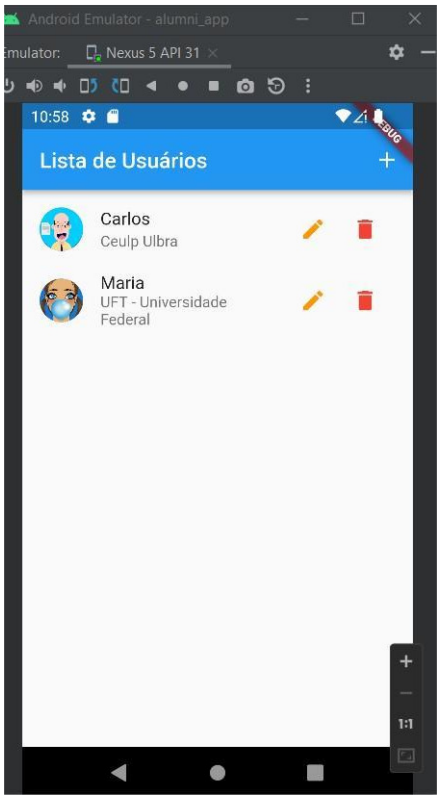
conciso e legível, melhora a abstração e validação da lógica de negócio para o qual foi proposto. Aperfeiçoa o conhecimento de administrar a arquitetura, leiaute, e desvios no projeto.

A escrita de teste poupa tempo e tem sua importância. A complexidade determina como os componentes serão organizados na tela. A partir do retorno obtido através das mensagens padrões do *Flutter* possibilita melhor experiência da escrita do teste. Em cenários e contextos onde a validação é complexa é vantajoso o uso deste benefício. É importante criar e manter testes escritos com qualidade demonstrando interesse na experiência do usuário.

Durante o processo, a integridade do aplicativo em qualquer situação de mudança será testada, feito o rastreamento dos defeitos e existência de falha ao simular as funções de exibir os elementos mudança de estado. Os elementos explorados, os componentes revelam as condições de testes a qual foram aplicados, respondem aos estímulos impostos e obtêm-se como resultados os cenários esperados. Conforme o avançar do desenvolvimento do aplicativo, testar garante a integridade das regras de negócio conforme as mudanças e cenários esperados.

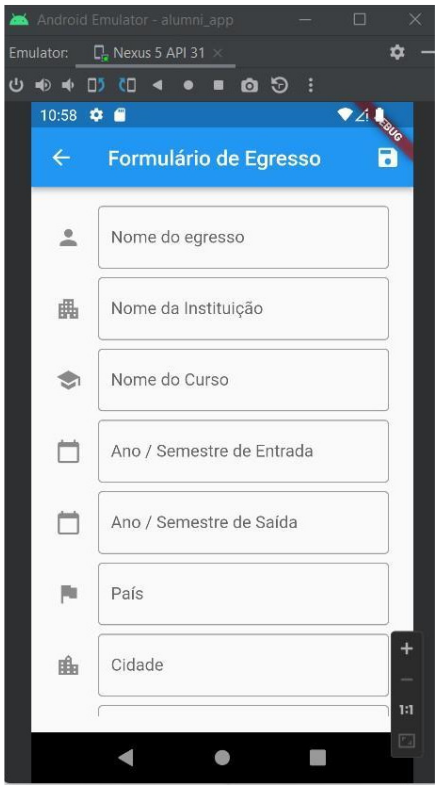
O aplicativo foi desenvolvido para smartphones para promover atividades entre instituições de ensino superior e egressos. O contexto escolhido do aplicativo foi o desenvolvimento híbrido onde se encaixa a interface de plataforma mobile para execução nos sistemas operacionais iOS e Android. Com a necessidade de desenvolvimento centralizado na ideia do projeto Alumni foi levantado os requisitos para implementação das telas.

Figura 11. Tela lista de usuários.



Na Figura 11 é apresentado a tela principal do aplicativo contendo lista de alunos egressos, identificação com nome e instituição de graduação. A tela contém a listagem de usuários pré-cadastrados exibindo os campos: nome do egresso, nome do curso, e imagem do avatar. Contém os botões cadastrar com o ícone: +, editar com o ícone: lápis, e excluir com o ícone: lixeira.

Figura 12. Tela formulário de egresso.



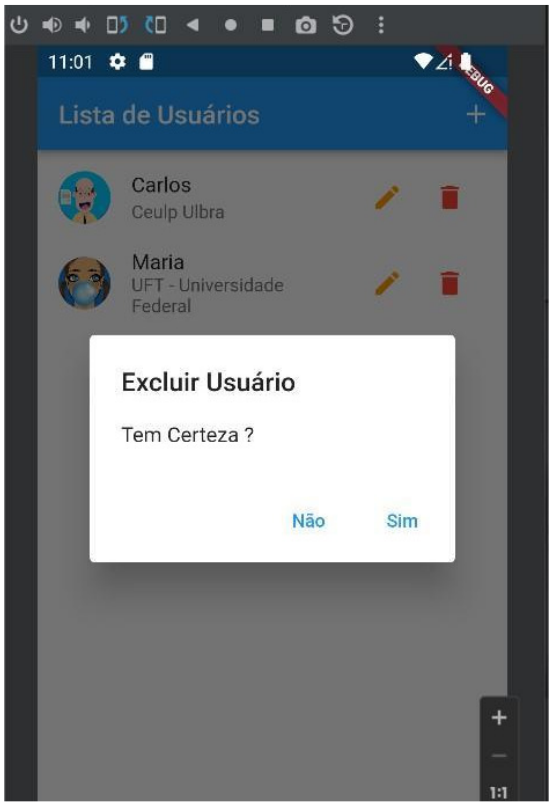
Na Figura 12 é apresentado tela de formulário para cadastro do egresso com objetivo coletar perfis para pesquisa e divulgação no mercado de trabalho. A entrada de cada campo de texto está associada a um ícone. O formulário contém em seu menu o botão com ícone disquete com a ação de salvar o formulário.

Figura 13. Tela Edição de egresso.

Na Figura 13 é apresentado os dados previamente preenchidos simulando interação ao acesso a um banco de dados exibido na tela formulário de egresso os dados carregados, contendo os campos preenchidos com os dados do primeiro usuário pré-cadastrado exibido na Figura.11. A tela é exibida após a ação de clique no botão de edição do usuário.

O objetivo desta funcionalidade é permitir que altere os dados de egresso do projeto, esta funcionalidade será permitida apenas para os usuários egressos, a pré-condição para que altere os dados é ter o formulário completamente preenchido, este fluxo faz parte da regra de negócio definida no projeto.

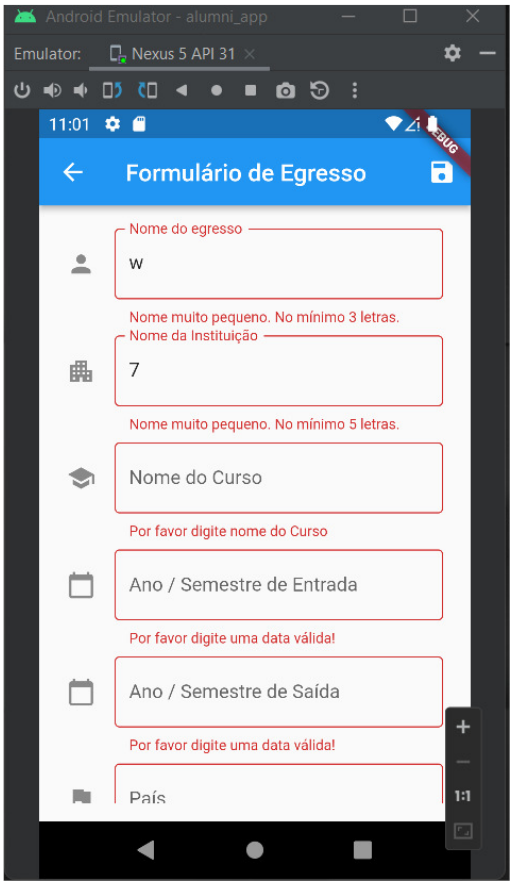
Figura 14. Tela exclusão de egresso.



Na Figura 14 é o aplicativo apresenta uma janela com uma mensagem de confirmação para o interesse em apagar o registro no banco de dados. O *modal* de confirmação após clique no botão excluir na tela lista de usuário com apresentação de uma janela solicitando a confirmação do usuário para concluir ou cancelar a ação.

A ação de excluir o egresso dispara o evento no aplicativo na qual é possível observar a mudança de estado do botão de não pressionado para pressionado. Excluir o usuário da lista ao clicar no ícone dispara um evento para o *Widget* responsável onde está contido o método com a função para remover.

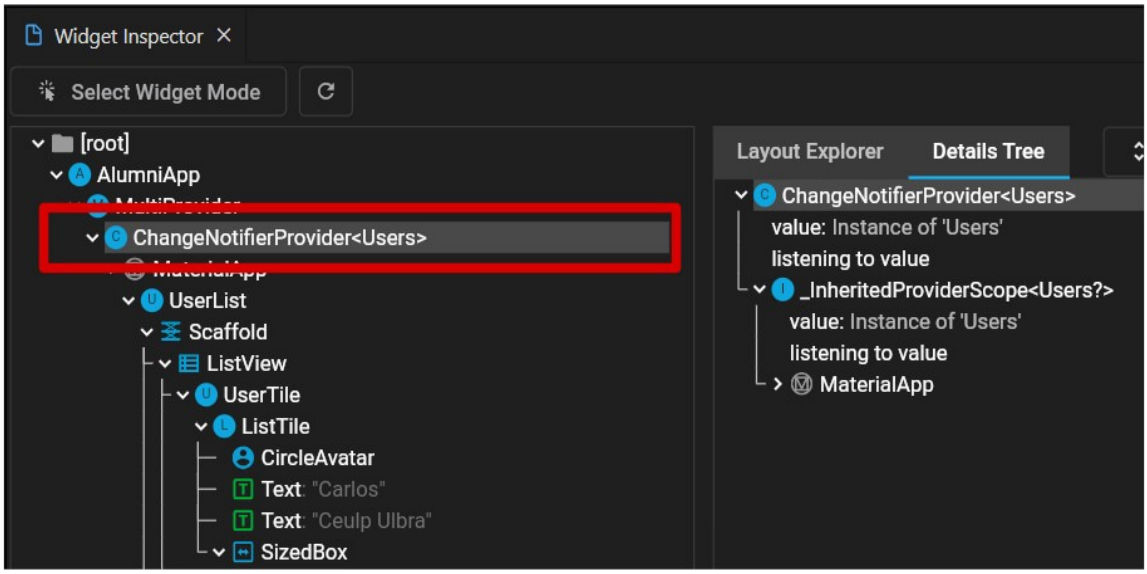
Figura 15. Tela formulário de egresso com validação.



Na Figura15 é apresentado o formulário com validação exigente das informações válidas informadas pelo usuário. Retorna a mensagem de erro amigável induzindo a correção para prosseguir e salvar. O retorno da mensagem é dado conforme validação de cada campo e informações fornecidas pelo usuário. Os erros dos campos são tratados pelo *Widgets* cuja propriedade valida se a condição é atendida com base na regra de negócio antes de enviá-los para salvar.

Os tópicos seguintes abordam sobre os Widgets desenvolvidos no aplicativo para este trabalho. É abordado a profundidade da árvore de Widgets e sua reconstrução, custo de desempenho e impacto de problema de performance em aplicativos. As telas desenvolvidas em relação aos requisitos funcionais e de desempenho foram feitas para os cenários de testes, os requisitos de configuração como a preparação do ambiente do projeto se deu por meio de teste manual, como um todo o projeto abrange os riscos, falhas a tecnologia utilizada e organização e controle das mudanças ao longo do seu desenvolvimento.

Figura 16. Inspecionando *ChangeNotifierProvider*.



Na Figura 16 é apresentada a árvore de *Widgets*, coluna à esquerda, onde estão localizados os Widgets visuais e comportamentais do app. Em hierarquia, a combinação destes compõem elementos de interface desenhando a aparência na tela de botões, ícones e listas.

O *Widget ChangeNotifierProvider* selecionado, demarcado com a linha vermelha na (Figura 16) gerencia o estado e aparência inicial do aplicativo. Ele é um provedor dos dados existentes mostrando na tela a visão geral da lista de usuários.

A visualização é possível por meio da ferramenta *DevTools* contida no Visual Studio Code o editor de texto desenvolvido pela Microsoft, a ferramenta possui o recurso de monitorar o código fonte em Dart e Flutter em tempo de execução permitindo examinar o estado do aplicativo, diagnosticar problemas de desempenho, consumo de memória, informações de rede e diagnósticos.

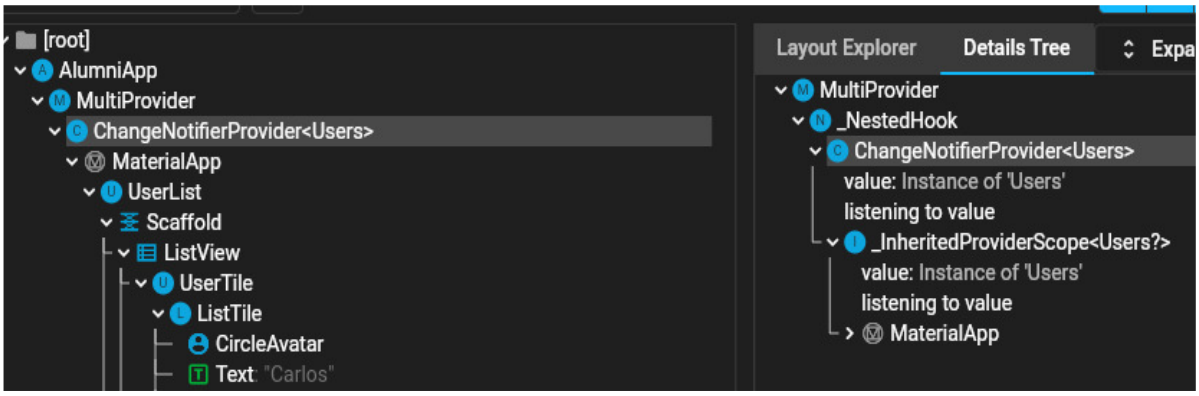
O uso em conjunto desta ferramenta auxilia a entender o processo de leiaute dos *Widgets* baseados em *Flex* classe do framework com a propriedade utilizada para organizar os espaços dos componentes e de seus *Widgets* filhos horizontal e vertical. Se o *Widget Container* filho do *Widget Column* em um formulário usado em conjunto com o *Widget Expanded* o auto preenchimento dos espaços vago dentro do pai disponíveis na tela é feito pelo filho automaticamente em seu posicionamento na tela, para indicar uma proporção diferente para um dado filho basta atribuir o valor numérico na propriedade flex.

O aplicativo foi implementado para fornecer interações e permitir que o usuário execute algumas ações para testes, estes eventos permitem a visão de alguns recursos usados para simular erro de estouro de leiaute na tela. A finalidade desta ferramenta consiste em

corrigir leiautes do *Flutter* que usam linha ou coluna, no contexto do trabalho para gerenciar e prover os dados no aplicativo independentemente do tamanho de tela. Na aba *Layout Explore* contida na Figura16 é possível visualizar os eixos cruzados renderizados na tela.

Figura 17. Detalhe da árvore *ChangeNotifierProvider*.

Na Figura 17 a coluna à direita, aba *Details Tree* contém a sub-árvore pertencente ao

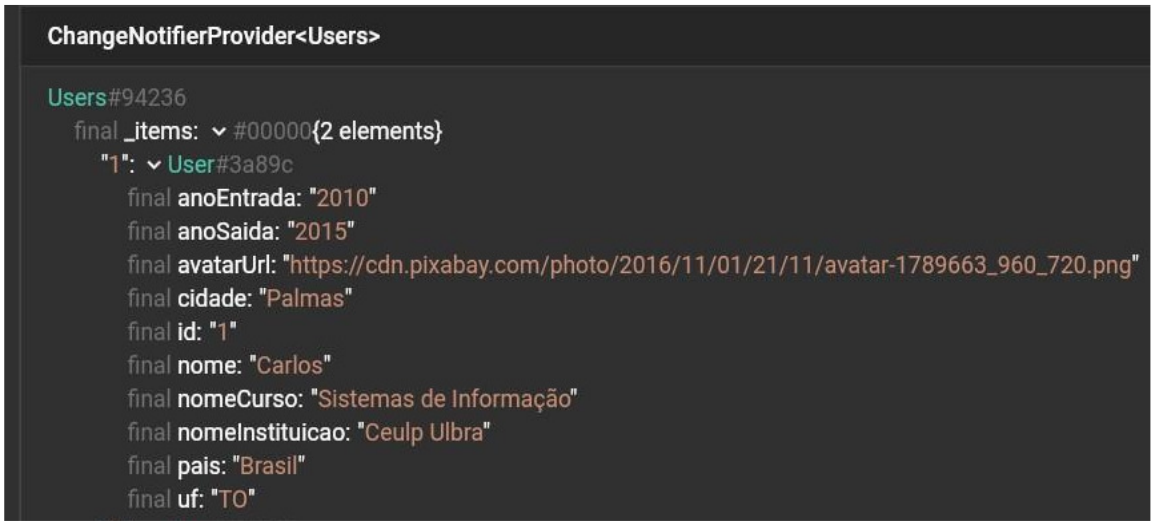


Widget selecionado. O *Widget* dispõe o valor e o passa propagando a diante na subárvore descendente de *Widget* a interface do usuário é reconstruída quando há mudanças do estado.

Isto é importante pois representa a maneira como os dados de usuário podem ser passados para o *Widget* filho. Os dados estão no topo, então o *Widget* do aplicativo está no meio, e a página inicial na parte inferior, o contexto de estado é usado como fluxo que fará que o aplicativo tenha acesso à referência dos dados do usuário e exiba-os na tela.

O *InheritedProviderScope* é um *Widget* herdado de *ChangeNotifierProvider*, com isso é fornece um método que escuta e notifica as mudanças sempre que um valor for alterado, neste contexto de usuários a reconstrução acontecerá com os valores atualizados.

Figura 18. *Provider*.



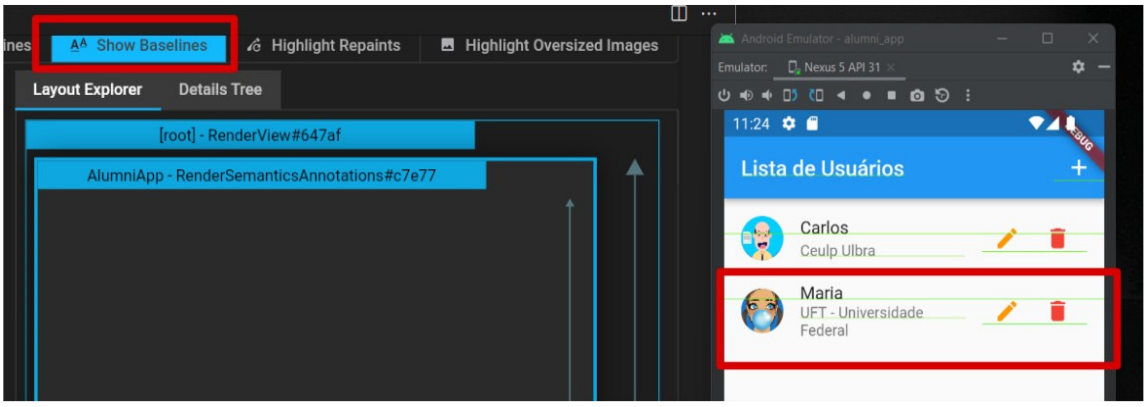
Na Figura 18 o *Widget Provider* propaga as informações para qualquer *Widget* acima ou abaixo da árvore de componentes fornecendo os dados distribuídos percorrendo a árvore de maneira eficiente e escalável em um aplicativo.

O *Provider* fornece uma maneira fácil e simplificada de separa a lógica de negócio dos *Widget's.* A informação é propagada ao envolver qualquer *Widget* em um *Provider* com uma variável declarada, qualquer *Widget* adicionado a este *Provider* acessará a variável declarada. Portanto o estado do *Provider* está acima dos *Widget's* provendo mudança de estado de um o *Widget* para outro.

No contexto deste trabalho se o *Widget* formulário sofrer qualquer alteração em um usuário específico o *Widget* lista externo em outra tela, contendo lista de usuários precisará de acesso ao estado que pertence a outro componente, então o *Provider* acionará toda mudança no aplicativo para qualquer *Widget* que esteja sendo gerenciado por ele,

O *ChangeNotifierProvider Widget* expõe os valores para os descendentes na árvore através de um atributo neste caso a lista com dados fictícios, os valores usados representam a lista de usuários egressos. O *Widget* armazena e representa os valores de cada item no padrão chave-valor, a chave representada pelo id e o valor com dados do usuário para cada item do formulário.

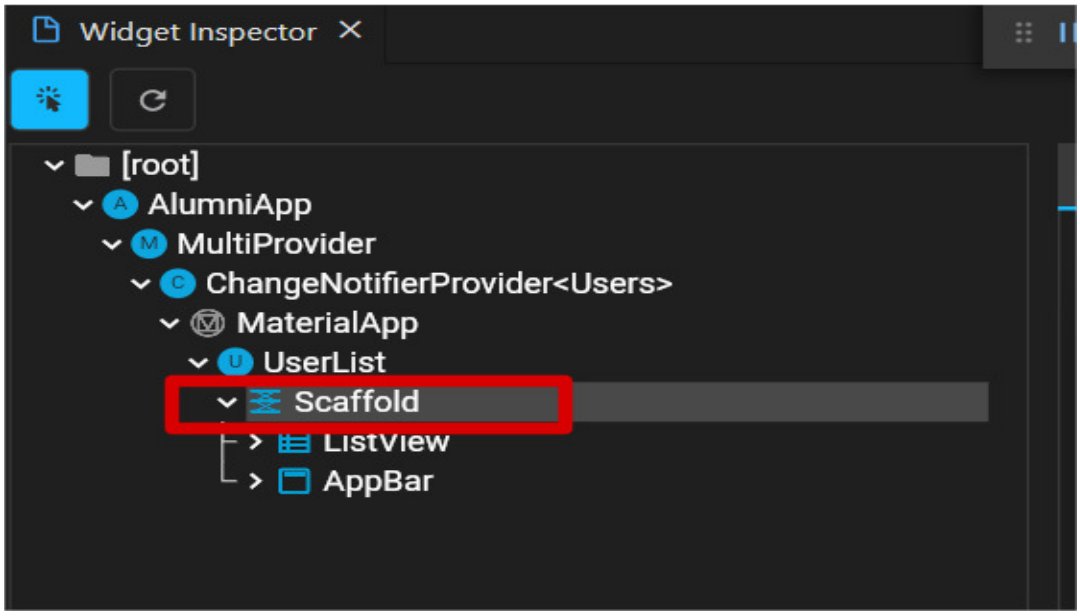
Figura 19.Tela lista de usuários *Widget Inspector*.



Na Figura 19 após selecionar opções: show baseline do *Widget Inspector*. Mostra as linhas de base para posicionar o texto, útil para verificar se há desalinhamento no texto.

As bordas em azul das caixas selecionadas mudam de cor sempre que a caixa é redesenhada, ou seja, se o *Widget* for modificado será redesenhado, e se for com muita frequência visualmente é perceptível notar que prejudica o desempenho com lentidão. As setas em verde representam a visualização de rolagem e o alinhamento com linhas amarelas.

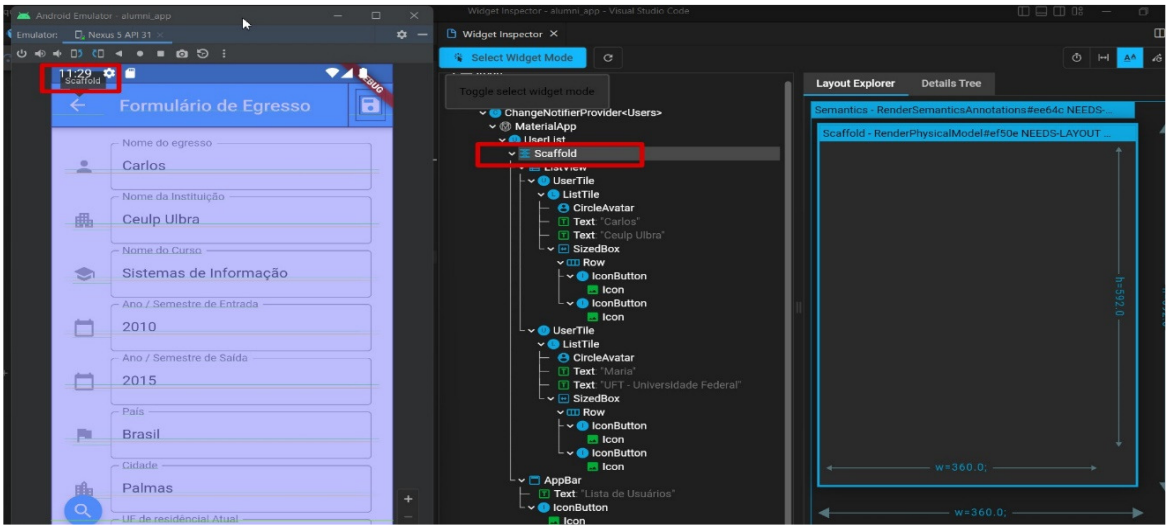
Figura 20.*Widget Scaffold*.



Na Figura 20 o *Widget Scaffold* selecionado constrói a estrutura do leiaute na qual permite demais componentes combinações internas desenhando na tela o formulário funcional e responsivo.

Para cadastro de um novo usuário é exibido o formulário alinhado ao centro da tela para preenchimento dos dados contendo os campos de texto para as validações de entrada de dado sem cada campo com o *Widget Form*, e não o *Widget TextField* que é um tipo para entrada de informações básicas sem validação.

Figura 21.*Widget Scaffold Slow Animations*.



Na Figura 21 após selecionado propriedade *Slow Animations* é apresentado o componente após ação de desacelerar uma animação para inspeção visual, a fim de se observar uma animação.

A animação de transição de tela é vista ao selecionar usuário no campo a ser editado. É possível ver a transição ao realizar as ações salvar e excluir também.

A apresentação dos dados simulados se fez necessária porque uma visualização vazia em um *Widget* não pode retornar nenhum valor indicando para visualizar, é possível contornar e evitar isto sem o *Widget Scaffold* porém resulta em uma tela com plano de fundo na cor preto.

Figura 22.*Widget AppBar*.

A Figura 22 contém o componente *AppBar*, barra de ferramentas composta com texto e botão para navegar para outra tela e voltar.

A navegação é um outro recurso utilizado no projeto. A navegação em um *Widget AppBar* contém a função de transição das telas exibidas, através de uma pilha de histórico para navegar para uma nova tela, responsável para adicionar ou remover tela da pilha.

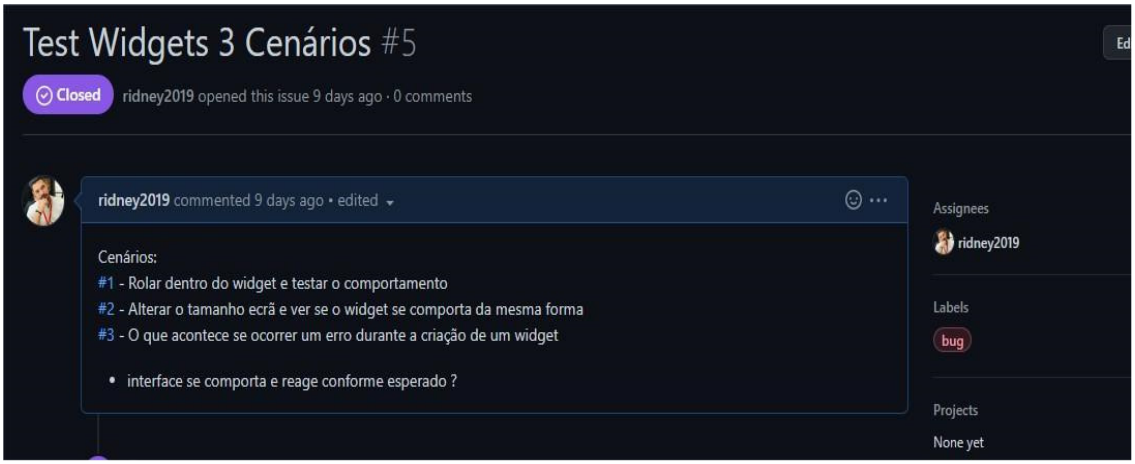
4.1 IMPLEMENTAÇÃO DOS TESTES

Apenas um arquivo de teste foi utilizado no projeto o nome do arquivo é “widget_test.dart”, este arquivo testa se a camada de controle na qual possui a responsabilidade de gerenciar a comunicação e o processamento os dados está sendo executada conforme esperado. Obrigatoriamente é necessário que o arquivo esteja dentro do diretório *test* deve possuir nome e o sufixo _test.dart por padrão.

Os métodos de testes são intuitivos e construídos com os elementos *WidgetTester* na qual permite interagir com os *Widgets* e construí-los no ambiente de teste. O *Finder* pesquisa *Widgets,* útil para usa-los para encontrar um *Widget* na qual é necessário realizar uma ação específica ou garantir que o *Widget* seja exibido, é possível pesquisar componentes com os tipos texto, botão, ícone, identificador chave entre outros. O Matcher auxilia e compara ao verificar se um dado *Finder* localiza um ou mais *Widgets.*

A implementação das telas utiliza o mínimo número de *Widgets* para desenho e posicionamento dos componentes. As classes criadas do modelo e lógica de negócio com baixo acoplamento de código para auxiliar na performance, manutenção e organização do entendimento do funcionamento da aplicação e desenvolvimento dos cenários de testes.

Figura 23.Cenários de testes.



A Figura 23 é apresentada a lista de cenários cadastrada e concluída no repositório do projeto contendo os casos para o plano de testes definidos em reunião. Dentro das estimativas estão a quantidade de cenários para teste das funcionalidades e operações dentro do aplicativo desenvolvido.

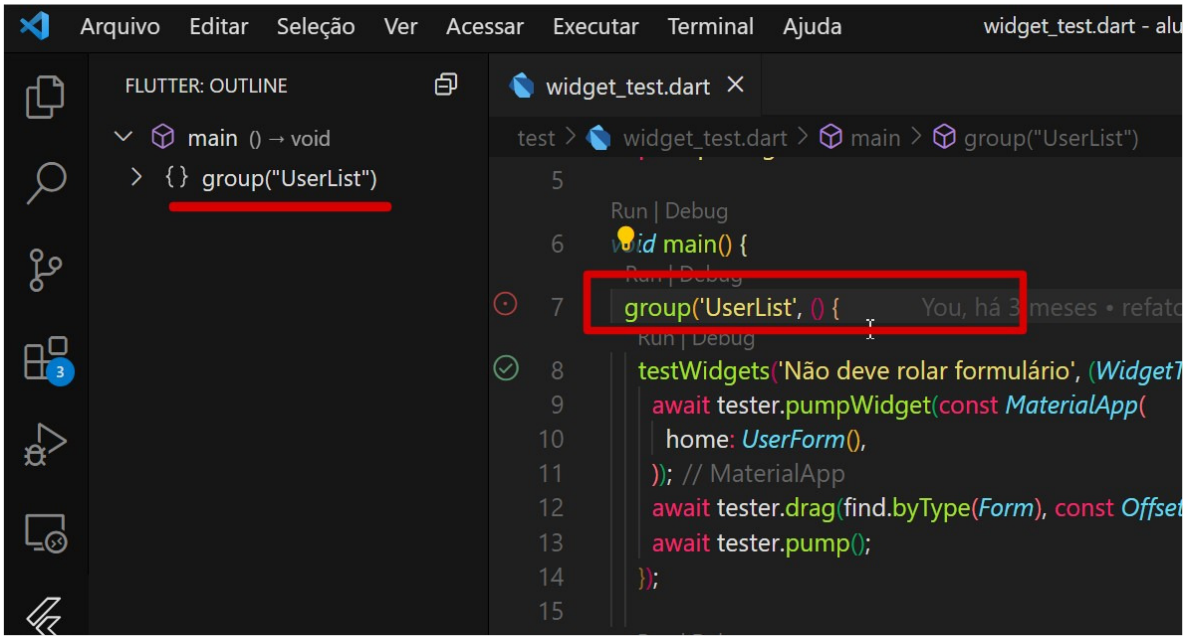
Em resumo a descrição breve de cada cenário é listada, com especificação clara e sucinta sobre operação a ser realizada no primeiro caso de teste de rolar dentro do *Widget* e testar o comportamento, o segundo caso de alterar o tamanho do ecrã da tela e validar se o mesmo se comporta da mesma forma, e o terceiro o que acontece se ocorrer um erro durante a criação de um *Widget* no caso de teste.

O Flutter dispõe de elementos “ferramentas” da classe de teste que representam pré-condições para escrita dos cenários e para o próximo passo de execução destes, de forma ordenada é exibido e executado os cenários em um grupo de teste que simulam para o contexto que foram aplicados. Observa-se os resultados e falhas esperadas para o fluxo sob as condições desejadas a fim de demonstrar que o requisito foi atendido além de identificar defeitos rastreados por meio de dados errôneos ou anomalia de desempenho do software.

Figura 24. Visão geral de testes.

A figura 24 apresenta uma visão geral do código implementado em todos cenários citados, estão contidos no grupo de testes com o título *UserList.*

Figura 25. Grupo de testes *UserList*.



A Figura 25exibe o grupo de teste foi criado cuja missão é organizar os testes agrupando-os com um determinado nome de sua escolha. É possível ver que o código está aninhado em um nível para executar todos os cenários em uma única vez, outro propósito de uso do grupo de teste é ter como alvo o mesmo objeto ou operação.

Um grande problema para os desenvolvedores é a adição de vários cenários de testes e agrupa-los resolve-o. Uma boa forma por exemplo de agrupar pode ser pelo tipo de operação que está realizando, a função de grupo não tem nenhum impacto no resultado dos testes, mas ajudará com cada nome prefixado pelo nome do grupo ao qual pertencem tornando mais fácil de ler e manter.

1º Cenário	2º Cenário	3º Cenário
Identificação do *Widget* no aplicativo	Aplicar teste no *Widget* definido	Observar o resultado

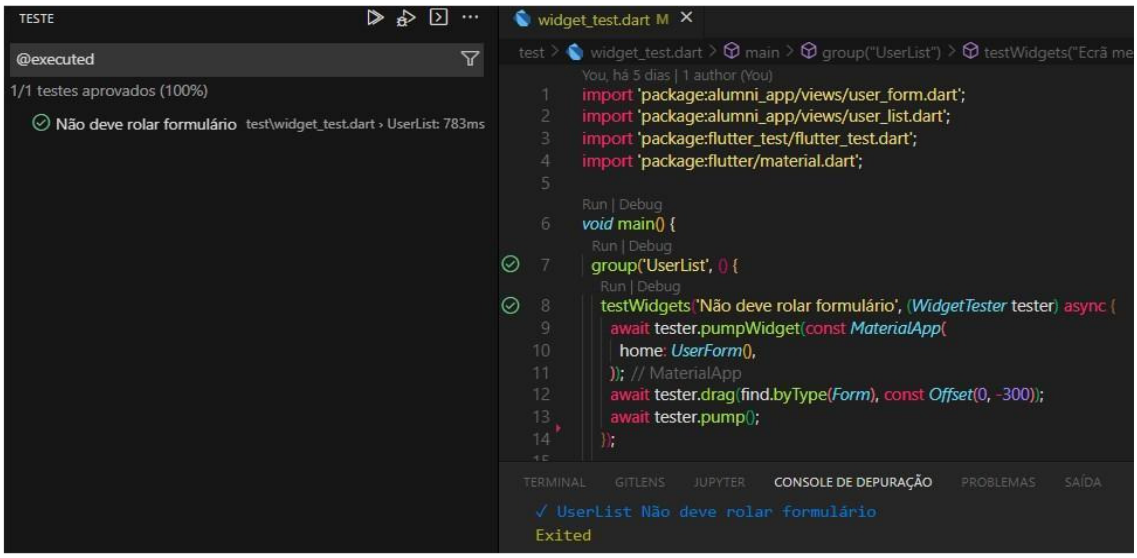
Tabela 3. Levantamento e etapas dos testes.

A tabela 3 apresenta o levantamento do processo na qual foi dividido em três etapas para execução das atividades de testes.

4.1.1 PRIMEIRO CENÁRIO

O cenário de teste é definido previamente com uma ação que o usuário pode realizar ou uma situação que o usuário pode se encontrar ao utilizar o *software.* Com ênfase em criar testes mais precisos com comportamento conforme esperado permite abranger fluxos e a cobertura de lógica do modelo de negócio, no contexto a seguir o *Widget* ou a composição de sua hierarquia considerando a perspectiva do usuário.

Figura 26.O teste não deve rolar formulário.



A Figura 26 ilustra a implementação que tem como finalidade testar o comportamento específico definido para este cenário.

A linha 8 do código contém o teste com a descrição *"Não deve rolar formulário"*, encarregada de usar o *WidgetTester* identificado dentro dos parênteses. A palavra-chave *async* que com função assíncrona, permite que a função possa ser executada em paralelo com outras enquanto o processamento desta não é finalizado.

A linha 9 iniciada com a palavra-chave *await* e. *pumpWidget(),* determinando esperar que o *Widget* simula e responda gerando o leiaute padrão de tela do *Flutter* criando a sua página inicial contendo o *Widget UserForm,* exibindo a lista de usuários.

A linha 12 é esperado novamente a palavra-chave *await tester* possibilitando arrastar algum elemento da tela do formulário*,* é feito a pesquisa do *Widget* formulário, ao encontrar o elemento é simulado o deslocamento. Para representar este comportamento é definido no teste os valores numéricos (0, -300) na classe de descolamento chamada Offset, esta propriedade requer que um *Widget* deve rolar a tela de uma posição inicial até determinada posição, porém só é executado a instrução na linha seguinte.

A linha 13 contém o *WidgetTester* para renderizar novamente a página e os componentes, pois o objetivo do teste é simular a interação do usuário arrastando a tela até a área delimitada.

O *Widget UserForm* declarada no teste aceita e realiza mudanças de estados com base nas interações do usuário, os elementos com estado alterados neste cenário estão em no *Widget UserForm(),* na qual é modificado em tempo de execução do aplicativo após ação de mover.

Na coluna à esquerda é possível ver o resultado com a descrição do nome do teste e que o teste foi concluído com êxito e atendeu o comportamento esperado. O teste foi capaz de obter exatamente um resultado que é verdadeiro, é possível ver que o resultado apresenta a cor verde. Se os valores forem alterados e alguma linha ou coluna com largura ou altura não delimitado dos *Widget's* filhos pertencentes ao *Widget Form* ocorrerá o erro de visualização por estouro de pixels na posição atribuída ao valor dado.

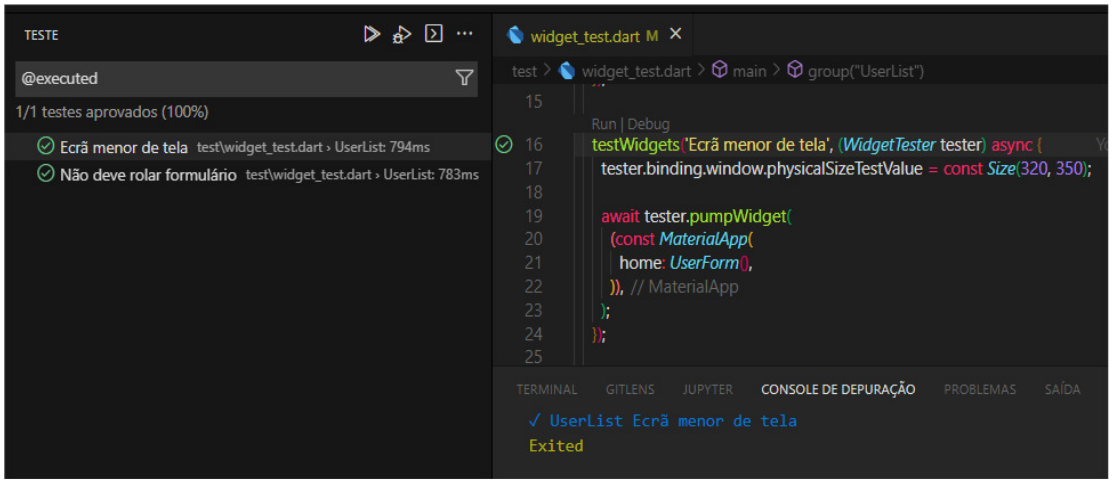
O método *find.byType()*, localizador por tipo procura *Widget's* com um tipo específico relevante para este cenário para atender o propósito do cenário esperado. O impacto do resultado positivo deste teste no cenário proposto tem suma importância, entrega a confiabilidade que o escopo do requisito inicial foi atendido considerando as boas práticas de código limpo após passar no teste. O teste validou também a hipótese de alteração de valores e aplicação de rolagem na hierarquia dos componentes filhos do *Widget Form* testado.

4.1.2 SEGUNDO CENÁRIO

A criação do cenário iniciou a partir da documentação dos requisitos levantados e o estudo e isolamento de cada ação possível do usuário. Associado ao teste proposto foi considerado problemas técnicos enquadrados ao cenário, considerando o que acontece com o ambiente diferente de resolução para qual foi produzido o aplicativo.

Portanto a dimensão de texto dos *Widget 's* na árvore podem ser redimensionados automaticamente de acordo com o tamanho, altura ou largura tela no *Flutter,* porém é necessário definir o *Widget* auto_size_text no arquivo de dependências do projeto. O foco deste cenário tem como objetivo e proposta final dimensionar e verificar o comportamento de como é exibida a tela antes da conclusão do aplicativo e testar a possibilidade em um ou mais dispositivos.

Figura 27.O teste ecrã menor de tela.



A Figura 27 ilustra o código implementado com objetivo de testar diferentes tamanhos de tela para um ou múltiplos dispositivos móveis.

A linha 16 testa o cenário se uma pequena tela mostrará o *Widget UserForm,* formulário de usuário em conjunto com *WidgetTester*, o teste contém a descrição "Ecrã menor de tela".

A linha 17 é passado o *WidgetTester* executa a propriedade *physicalSizeTestValue,* para altera o valor de tamanho físico da janela para um tamanho específico reduzido para este dispositivo, simulando execução em outro dispositivo.

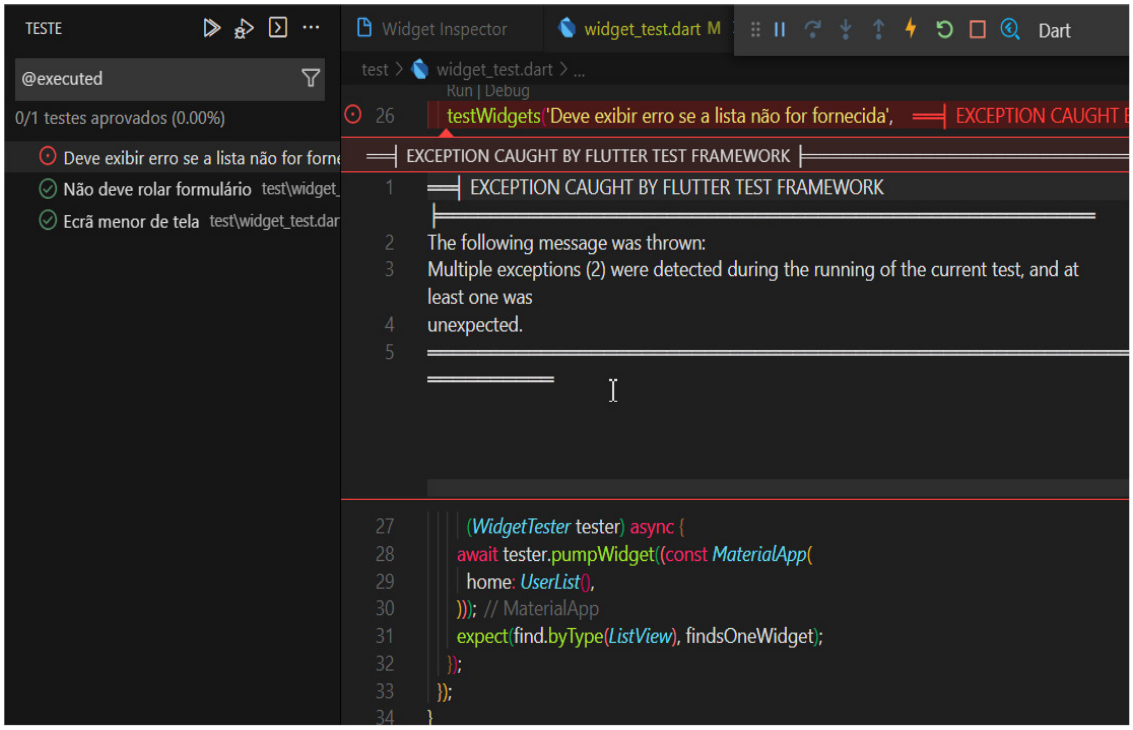
A linha 19 inicializa o aplicativo renderizando a página e os *Widgets.* Na coluna à esquerda é possível ver o resultado com a descrição do nome do teste e que o teste foi concluído e funcionando atendendo o comportamento esperado.

É possível ver o tempo de apenas 794 milissegundos que este teste levou para ser executado, se for preciso testar isso manualmente, seria necessário iniciar um emulador ou usar um dispositivo com este tamanho de tela, levando mais tempo para testar este comportamento.

4.1.3 TERCEIRO CENÁRIO

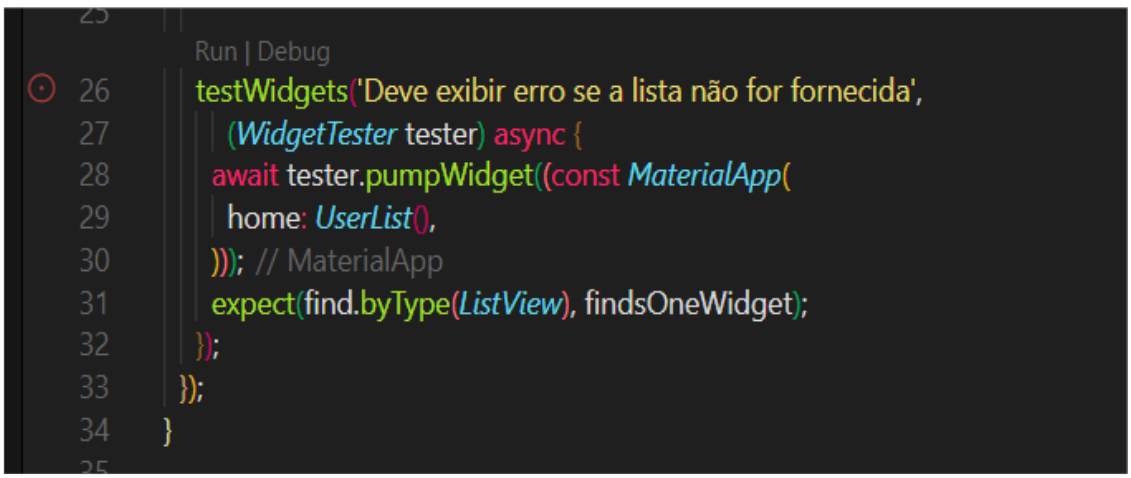
A proposta do cenário com possível erro foi levada em questão para interpretar a captura de causas que ocorrem durante a execução. O que deve ser feito após uma entrada incorreta de dados do usuário ou erro cometido pelo programador, em algum momento o aplicativo falhará, o tratamento de exceções a melhor forma de lidar com erros de forma mais explícita.

Figura 28. Visão geral teste erro lista não fornecida.



A Figura 28 contém uma visão geral do último cenário contido no grupo de testes, verifica o que acontece se o *Widget* retornar um erro.

Figura 29. Implementação do teste erro não fornecida.



A linha 28 é declarado o teste com o cenário para dispara um erro se um *Widget ListView* não for fornecido, o teste contém a descrição “Deve exibir erro se a lista não for fornecida”, executando o *WidgetTester* declarado dentro dos parênteses.

A linha 28 iniciada com a palavra-chave *await* e. *pumpWidget(),* determinando esperar que o *Widget* simula e responda gerando o leiaute padrão de tela do *Flutter* criando a sua página inicial contendo o *Widget UserForm,* exibindo a lista de usuários.

A linha 31 recebe o resultado esperado após a buscar para encontrar o elemento em sequência pesquisado pelo tipo, no contexto o *Widget ListView,* porém para que o teste falhe é esperado que receba uma exceção pois não é correspondido a busca do *Widget ListView* para validar este cenário.

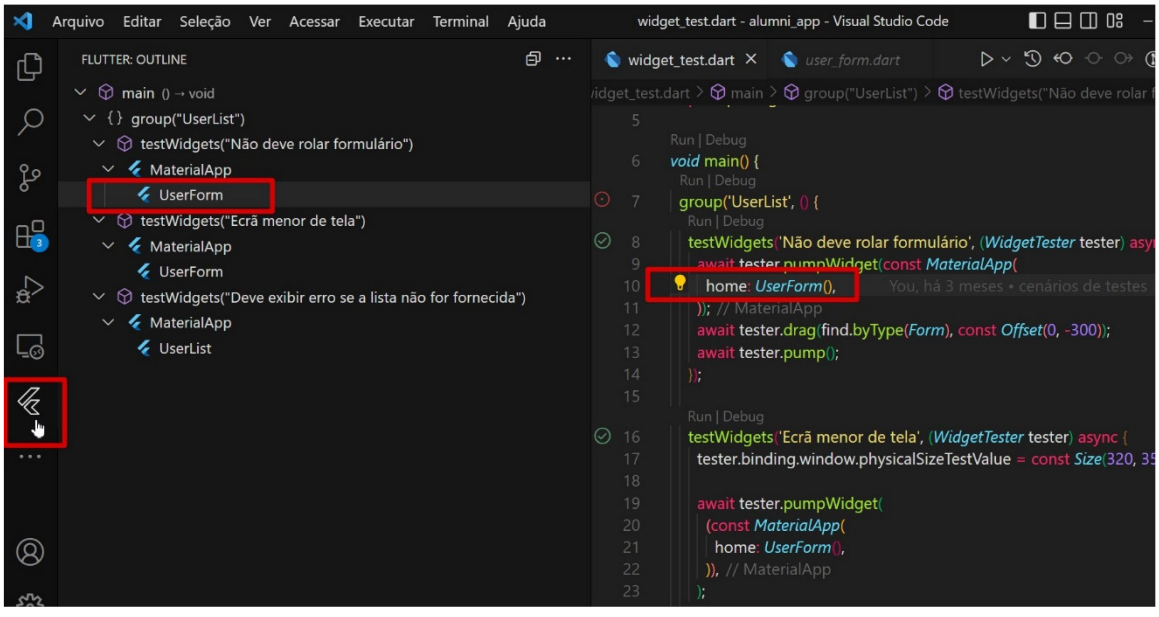
Se o valor esperado e o encontrado correspondem a busca, o teste passará e o código não agirá conforme o esperado não é este o caso proposto deste cenário.

4.2 RESULTADO

As informações para levantamento dos requisitos se deram por meio de reuniões presenciais e online, nestas foram definidas as funcionalidades que norteiam o aplicativo desta pesquisa e os cenários de teste. Uma pessoa apenas era responsável e especialista de domínio

A visualização da hierarquia dos *Widgets* utilizados no teste bem como suas associações entre componentes pais e filhos é possível através da janela de ferramentas embutido como recurso do plugin Dart / Flutter.

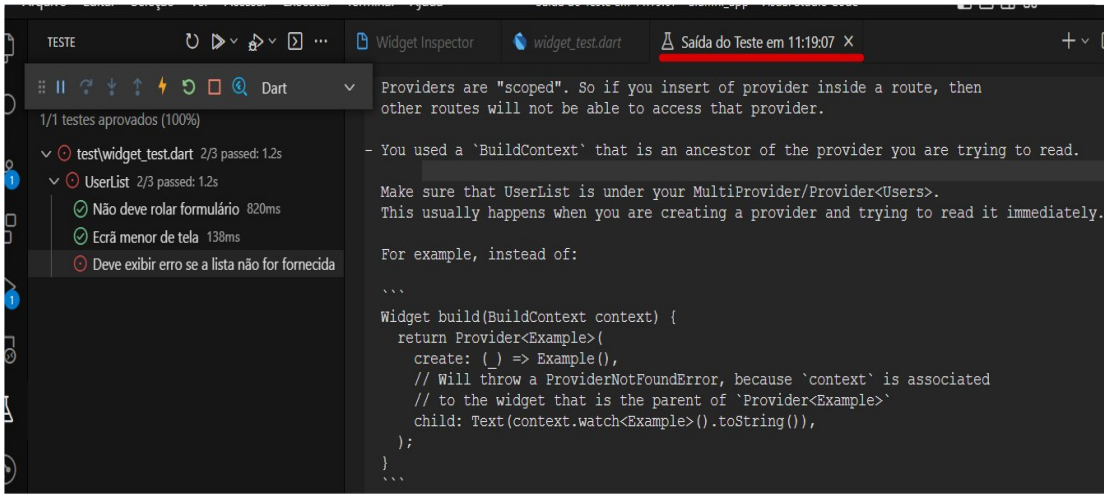
Figura 30. *Flutter Outline*.



A Figura 30 fornece a visualização dos *Widgets* utilizados nos cenários de teste, é possível ver a área delimitada na cor vermelha em quadrado com a logomarca do framework ao centro no canto inferior esquerdo da Figura, No contexto é possível ver a classe principal main() do arquivo de teste *Widget_test.dart* o campo UserForm delimitado com um retângulo ao ser selecionado.

No desenvolvimento de grupo de testes a visualização e organização destes arquivos enquanto são codificados podem ser um problema mapeá-los em um projeto grande, para que o desenvolvedor tenha uma visão geral e possa associar a estrutura visual e o código o uso deste recurso se torna útil e prático para análise dos relatórios.

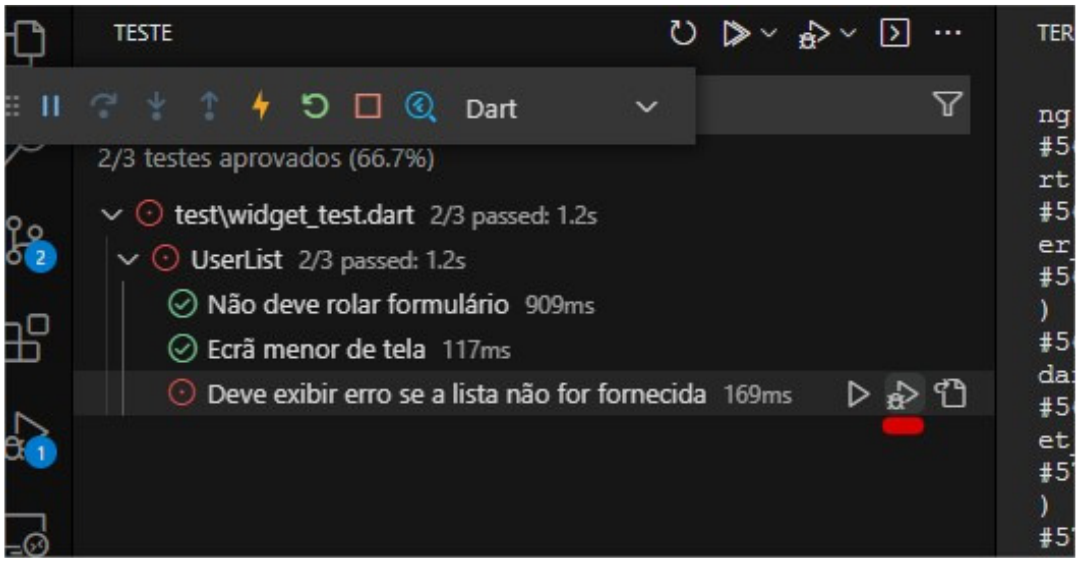
Figura 31. Saída do Teste.



A Figura 31 apresenta o resultado de saída impresso via terminal, a visualização é possível após a ação de clicar no ícone em triângulo ao lado da seta circular no menu de testes, selecionar a opção executar teste padrão, após execução o resultado é apresentado na coluna esquerda com o nome dos testes executados, o tempo específico que cada cenário levou e o indicativo na cor verde sinalizando que o teste passou e na cor vermelha que houve falha.

Para exibição do relatório de saída do teste é necessário selecionar o último ícone, mostrar saída, situado na barra de menu na coluna a esquerda ou via atalho do teclado pressionando simultaneamente as teclas “Ctrl + ç + o”.

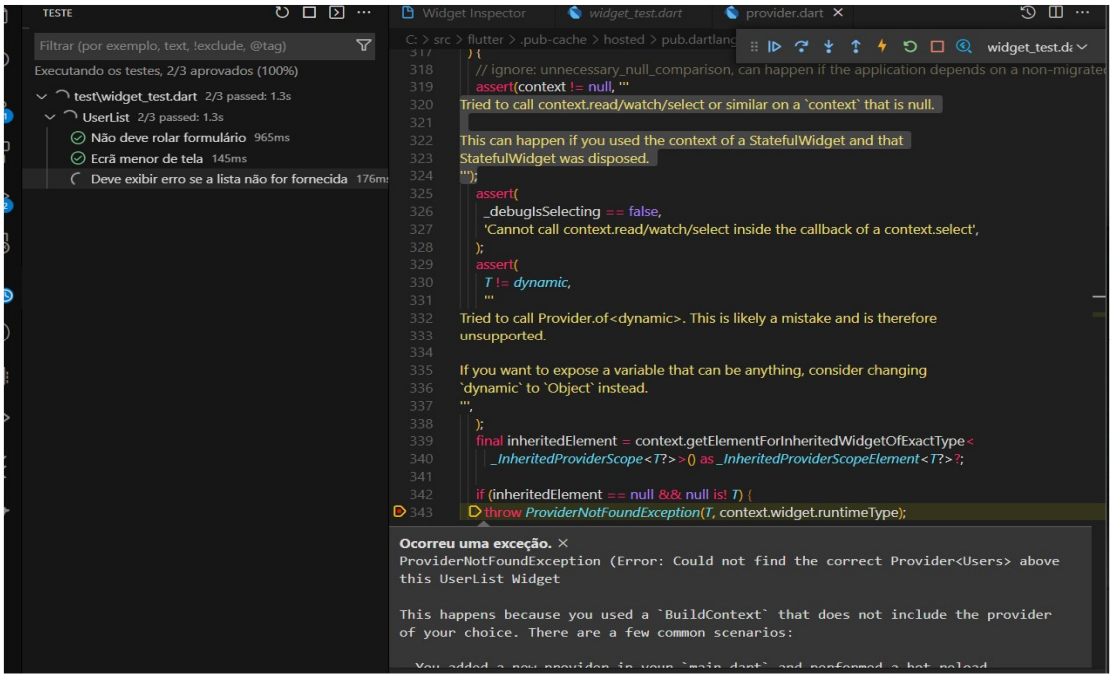
Figura 32. Depurar testes.



A Figura 32 exibi a coluna de teste executada em modo de depuração, na área demarcada em vermelho contém o ícone para depurar apenas o teste individual, na barra de menu da coluna de teste contém o mesmo ícone, porém com a opção de executar todos os testes em modo de depuração.

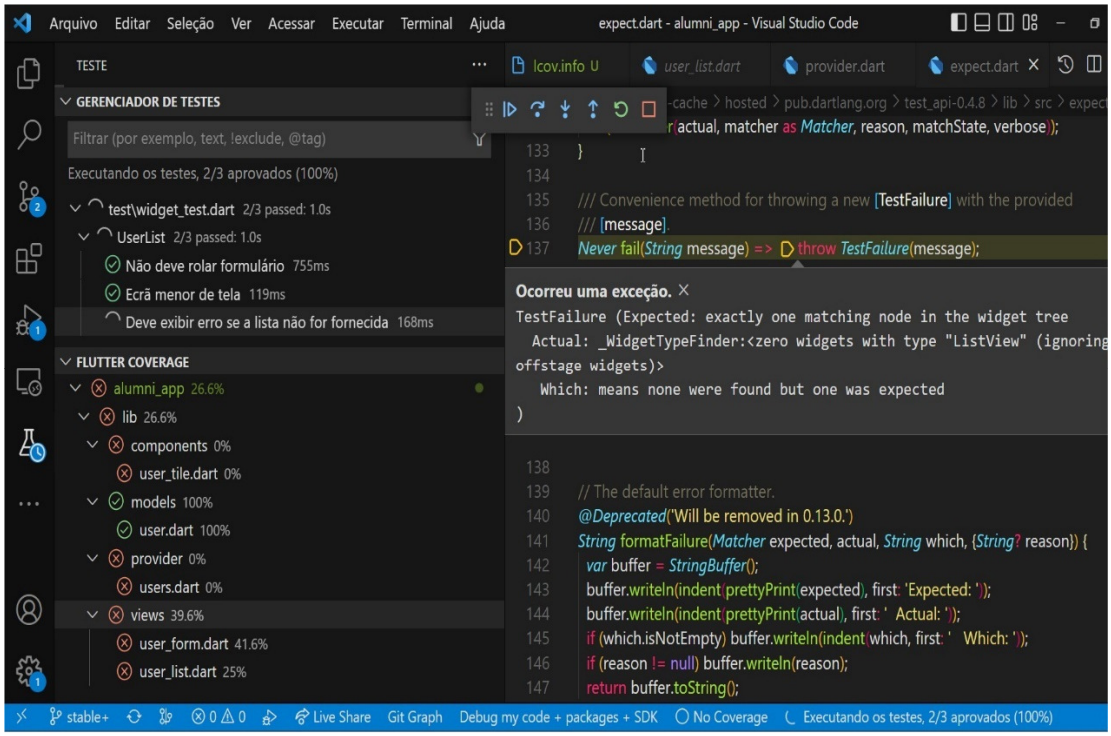
É importante o uso do modo de depuração do teste para o processo pois se algum teste vir a falhar a execução geral deste cenário específico que foi apontado com falha é executado passo a passo até encontrar a causa. Com o foco em separa e isolar de maneira eficaz o teste que apresentou a falha como previsto no cenário acima após a indicação da falha no teste as informações são apresentadas para diagnóstico e auxiliar a entender o ponto de interrupção para ser corrigido o problema.

Figura 33. Detalhe da depuração.



A Figura 33 apresenta a mensagem e a indicando onde foi encontrada a falha no teste, depurando o fluxo de trabalho é possível entender o passo a passo que resultaram no erro. A importância da execução dos detalhes de depuração ajuda a identificar se o erro foi causado por construção de sintaxe, layout o desenho na tela para corrigi-los.

Figura 34. Continuação da depuração.

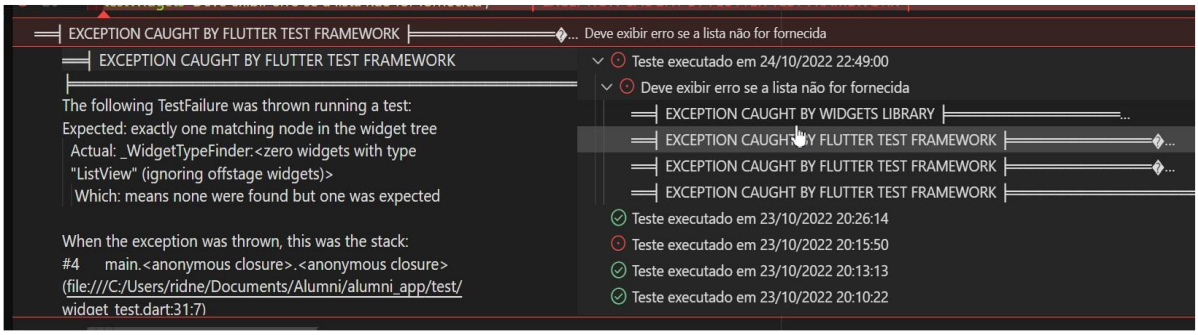


A Figura 34 apresenta a mensagem com maior detalhe que o Widget List esperado na busca no cenário de teste não está presente na criação do cenário simulado, portanto o teste só poderá encontrar Widgets renderizados, esta informação é útil pois na mensagem contém o tipo do Widget. Os localizadores método find.byType() apresentam mensagens de erro quando suas asserções falham, sendo mais rápido que os métodos find.ancestor() e find.descendant() percorrendo o conjunto da árvore de Widgets.

A coluna esquerda abaixo do gerenciador de testes contém a exibição em árvore relativa ao container de testes exibindo a cobertura do caminho arquivo/pasta, onde os arquivos de teste estão sendo localizados. O percentual da cobertura é indicado em cada arquivo da árvore o teste passou.

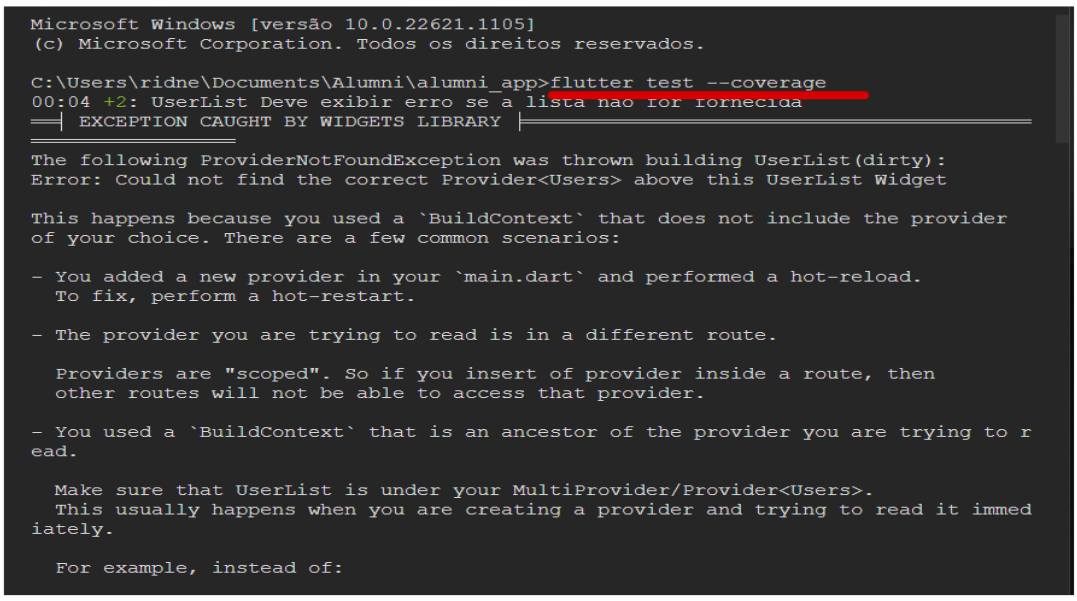
Os Plugins instalados no VSCode são Flutter Coverage e Coverage Gutters o primeiro responsável pela visualização da coluna Flutter coverage e o segundo para acionar a opção “watch” na parte inferior da IDE. Para tal ação basta clicar em “No coverage” que o status mudará para “watch”, então basta clicar em algum arquivo da arvore na coluna Flutter coverage que será possível visualizar quais linhas do arquivo selecionado o teste passou ou não.

Figura 35. Exceção *Flutter*.



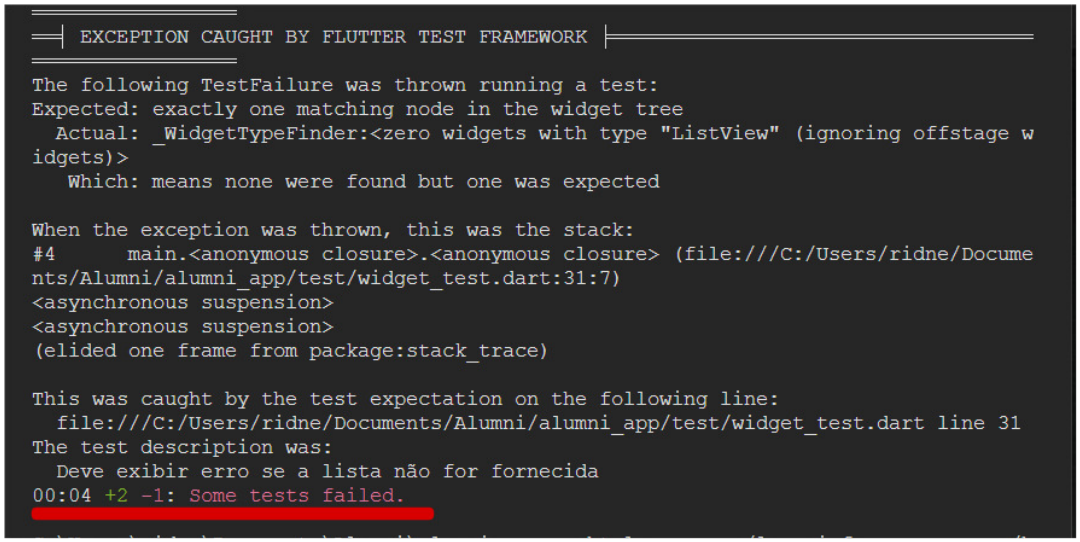
Conforme descrito no log de teste é possível visualizar a mensagem de exceção do erro capturada pelo *Flutter,* esperava exatamente um nó correspondente na árvore de *Widgets,* como resultado nenhum, zero Widgets do tipo *ListView* foi encontrado na árvore de busca. Os objetos de teste são para comparar o que é esperado que aconteça e o que realmente aconteceu, sendo assim o teste cumpriu seu objetivo passando pela exceção.

Figura 36. Relatório por comando.



A Figura 36 apresenta o comando Flutter test –coverage demarcado através da linha tracejada em vermelho, quando executado via terminal na ferramenta VSCode gerar a visualização do relatório de cobertura dos testes. Lista o registro com a mensagem de exceção encontrada durante a falha de um teste.

Figura 37. Relatório por comando informação



A Figura 37 com a linha demarcada em vermelho o resultado da execução dos testes, com a propriedade de tempo total indicando 00:04 segundos, seguido do indicativo com o sinal positivo que dois testes passaram “+2” e um teste com sinal negativo “-1” com a mensagem que algum teste falhou.

A exceção capturada pelo framework, ou seja, um caso excepcional o comportamento esperado pelo tipo de exceção a regra de funcionamento pela falta de envolver o Widget List para o Widget Provider ocorrida no código do arquivo de teste conforme aponta na Figura 37 indicando a linha 31.

5 CONSIDERAÇÕES FINAIS

O intuito deste trabalho foi mostrar o desenvolvimento de testes automatizados em um aplicativo desenvolvido em Flutter, com visão geral de suas estruturas e elementos evidenciando as melhores práticas para o desenvolvimento ágil, com intuito de ilustrar suas etapas, mitigar falhas e contribuir com a melhoria da qualidade.

Ao lançar uma próxima versão do aplicativo um dos piores cenários é descobrir que algo não está certo, que contém *bugs,* idealmente seria detectado automaticamente quaisquer problemas antes com testes de integração contínua, mas a execução de testes móveis pode ser difícil em diferentes plataformas, dispositivos, níveis, versões de API e configurações possíveis para detectar problemas antes do usuário.

As considerações acerca do que foi desenvolvido na proposta deste trabalho é ser referência e contribuir com o tema por meio de informações teórica e prática. Com intenção de assegurar melhor qualidade os testes são úteis para medir esforço, cobertura e

monitoramento do projeto, minimizando o tempo gasto auxiliando a compreensão dos requisitos de design e codificação.

As implicações deste se resumem ao uso do notebook com configurações de hardware que atendem a capacidade de simular o aplicativo tendo em vista que não foi utilizado nenhum *smartphone*. O impacto foi mitigar o número de erros possíveis, avaliar a qualidade do processo de cada ação dos cenários citados anteriormente e diminuir o tempo de resposta. As influências dos resultados possibilitam a rastreabilidade do comportamento se está de acordo com o requisito, ter visão do produto e qual release é possível utilizar as funcionalidades do aplicativo.

Como trabalhos futuros considera importante a ampliação para cobertura de outros tipos de testes. A utilização do recurso de integração com o *Firebase* como dependência externa podendo adicionar novas funcionalidades como por exemplo autenticação simulada de usuários, onde em um arquivo de teste valida os campos de e-mail e senha e se os dados foram persistidos com sucesso após período de tempo definido, se houver algum erro o teste deve validar o tratamento da mensagem no aplicativo.

O desenvolvimento de teste de *Widget* em aplicações *Offline First* (Primeiro Offline) aplicativo com funcionamento pensado para uso offline, mesmo que seja por um período de tempo curto onde a funcionalidade de mensagens ou perfil possa ser vista sem conexão. Onde a ação de alterar foto do perfil ou resposta de uma mensagem seja possível e quando retornar a conexão estas ações são disparadas para o banco de dados online.

Outro cenário possível para teste em trabalhos futuros é encontrar um *Widget* identificado com um valor chave único específico fornecido a ele. É útil em casos específicos em um *Widget ListView*, onde contém vários componentes do mesmo *Widget,* permite que o teste identifique um *Widget* de texto que possua a mesma informação de texto dos demais, facilitando sua localização no ambiente de teste e aplicativo.

O processo de teste precisa ser automatizado, minimizando o custo e o esforço de manutenção isso para que se possa ter uma acurácia melhor de quais tipos de conjunto de testes podem ser utilizados para melhor predição, desempenho do aplicativo mais rápidos e os usuários satisfeitos.

6 REFERÊNCIAS

COHN, M.Succeeding with Agile: software development using Scrum(Addison-Wesley Signature Series). Upper Saddle River, NJ: Addison-Wesley, 2009. 504p.

CRISPIN, L.; GREGORY, J. Agile testing: a practical guide for testers. [S.l.]: AddisonWesley Professional, 2009.

DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew. Continuous Integration: Improving Software Quality and Reducing Risk. Boston: Addisonwesley Professional, 2007. 336 p.

ECMA INTERNATIONAL (Eua) (org.). **Extension of the RF Ecma Patent Policy Option also to "Royalty Free TCs**. 2019. Disponível em: https://www.ecma-international.org/news/extension-of-the-rf-ecma-patent-policy-option-also-to-royalty-free-tcs/. Acesso em: 03 jun. 2022.

FLUTTER. Start thinking declaratively. Disponível em: https://docs.flutter.dev/development/data-and-backend/state-mgmt/declarative. Acesso em: 30 abr. 2022.

FLUTTER.Testing Flutter apps. Disponível em: https://docs.flutter.dev/testing. Acesso em: 23 abr. 2022.

GETULIO K. AKABANE. Gestão Estratégica Da Tecnologia Da Informação: Conceitos, Metodologias, Planejamento E Avaliações. 1ª edição ed. [*S. l.: s. n.*], 2012.

GOOGLE DEVELOPERS (Eua). Google (org.). **Conheça o Android Studio**. 2021. Disponível em: https://developer.android.com/studio/intro/. Acesso em: 03 jun. 2022.

GOOGLE (Eua) (org.). **Introduction to widgets**. 2022. Disponível em: https://docs.flutter.dev/development/ui/widgets-intro. Acesso em: 23 mar. 2022.

GOOGLE (Eua) (org.). **Dart overview**. 2022. Disponível em: https://dart.dev/overview. Acesso em: 03 jun. 2022.

GOOGLE (Eua) (org.). **Flutter architectural overview**. 2022. Disponível em: https://docs.flutter.dev/resources/architectural-overview. Acesso em: 05 jun. 2022.

GOOGLE (Eua) (org.). **Mock dependencies using Mockito**. 2022. Disponível em: https://docs.flutter.dev/cookbook/testing/unit/mocking. Acesso em: 08 jun. 2022.

GOOGLE DEVELOPERS (Eua) (org.). **Make your app the best it can be**. 2022. Disponível em: https://firebase.google.com. Acesso em: 03 jun. 2022.

Ian Sommerville. Engenharia de Software, 9ª Edição. Pearson Education, 2011.

Ian Sommerville. Engenharia de Software, 8ª Edição. Pearson Education, 2007.

MICROSOFT (Eua) (org.). **É assim que você faz o software**: conheça a visual studio família. Conheça a Visual Studio família. 2022. Disponível em: https://visualstudio.microsoft.com/pt-br/. Acesso em: 06 jun. 2022.

ORGANIZADOR RAFAEL FÉLIX. Teste de software. Pearson 139 ISBN 9788543020211.

STEVE FREEMAN;NAT PRYCE. **Desenvolvimento de software - orientado a objetos**. 1ª ed. [*S. l.: s. n.*], 2012.

UNIVERSITAT POLITÉCNICA DE VALÊNCIA. Desarrollo de una aplicación móvil multiplataforma para la creación y resolución de nonogramas. Disponível em: https://riunet.upv.es/handle/10251/172217. Acesso em 13 maio.2022.

WILSON DE PÁDUA PAULA FILHO. Engenharia de Software - Projetos e Processos. 4ª ed. [*S. l.*]: LTC, 2019.