Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Facilitando a Criação de Testes de UI Automatizados em Fluxos de Aplicações iOS

Jacqueline Alves Barbosa

Trabalho de Graduação

Recife - PE
Setembro/2023

Universidade Federal de Pernambuco

Centro de Informática

Jacqueline Alves Barbosa

Facilitando a Criação de Testes de UI Automatizados em Fluxos de Aplicações iOS

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Kiev Santos da Gama*

Recife - PE
Setembro/2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Barbosa, Jacqueline Alves.
Facilitando a criação de testes de UI automatizados em fluxos de aplicações iOS / Jacqueline Alves Barbosa. - Recife, 2023.
61 p. : il., tab.

Orientador(a): Kiev Santos da Gama
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2023.
Inclui referências, apêndices.

1. Automação de Testes. 2. iOS. 3. XCUITest. 4. Lua. I. Gama, Kiev Santos da. (Orientação). II. Título.

000 CDD (22.ed.)

Dedico este trabalho à Jacqueline de 17 anos, que por vezes duvidou que conseguiria chegar até aqui. E a todas as pessoas que em algum momento a ajudaram a não desistir.

Agradecimentos

Gostaria de agradecer aos meus pais, que para mudar de vida tiveram que deixar muito para trás, mas que também construíram muito juntos. Obrigada por me ensinarem a importância de ir atrás do que quero e por sempre apoiarem o que vi como melhor para mim. Quero agradecer também os meus irmãos, que tiveram um papel fundamental na construção de quem sou e me incentivaram a querer continuar.

Quero também agradecer aos meus amigos, especialmente aqueles que conheci no Centro de Informática e que até hoje tenho o privilégio de conviver. Biel, Ícaro, Igor, e Pedro, vocês tornaram esta jornada mais leve e divertida. Obrigada por me ajudar a persistir e seguir em frente, uma carona por vez. Um agradecimento especial para os meus amigos Renan e Alyne, que estiveram ao meu lado nos momentos de desânimo e me ajudaram a não deixar que ele tomasse conta de mim.

Minha gratidão se estende a todas as pessoas da Apple Developer Academy, que estiveram presentes ao longo de dois anos de muito aprendizado e crescimento, tanto profissional quanto pessoal. Minha trajetória tomou um rumo diferente graças a vocês. Entre todos, quero fazer um agradecimento especial a Kiev, que aceitou me orientar neste trabalho e me deu a confiança necessária para segui-lo. Por fim, quero expressar minha gratidão a todos os professores que fizeram parte da minha formação, desde o jardim de infância até a graduação. Cada um de vocês, que escolheu como missão de vida compartilhar conhecimento com tantas pessoas, desempenhou um papel crucial nesta conquista. Agradeço profundamente por terem ajudado a me tornar a pessoa que sou hoje.

Obrigada!

"A frase mais perigosa que existe em um idioma é: Sempre fizemos assim."
—GRACE HOPPER

Resumo

O processo de desenvolvimento de aplicativos móveis envolve o planejamento e design da interface gráfica, que geralmente é entregue à equipe de desenvolvimento por meio de *mockups*, que é uma representação visual e não funcional do produto demonstrando como os componentes estarão presentes na interface. Testes, como os de interface de usuário (UI), regressão e integração, são essenciais para garantir a qualidade do software. No entanto, os testes de UI são muitas vezes executados de forma manual, o que os torna demorados e vulneráveis a erros, uma vez que dependem inteiramente da atenção humana que, com o passar do tempo e após várias repetições durante a realização dos testes, pode perder a atenção e deixar que falhas passem despercebidas. A automação desses testes oferece diversas vantagens, mas muitas ferramentas têm barreiras de uso devido ao conhecimento técnico necessário.

No caso de aplicações iOS, um framework muito utilizado é o *XCUITest*, disponibilizado pela Apple. Apesar de sua fácil integração com o ambiente de desenvolvimento, contém algumas limitações como a necessidade de que os testes sejam escritos na linguagem Swift e a falta de uma maneira de realizar uma verificação visual da aplicação sendo testada.

Nesse trabalho é proposto um framework que simplifica a automação de testes de UI em aplicativos iOS por meio de scripts. Esse framework verifica a integração e a aparência da interface em diferentes dispositivos e configurações, tornando a automação de testes mais acessível, mesmo para desenvolvedores com pouca experiência. Os testes são executados através da integração de scripts, escritos em Lua, com o XCUITest, que simula as interações de um usuário na aplicação.

Foi possível implementar casos de teste com um número reduzido de linhas de código, em comparação com o uso somente da API do XCUITest, e abstraindo detalhes da implementação interna da aplicação. Além disso, também foi possível validar visualmente fluxos da aplicação testada a partir de imagens de referência previamente definifas.

Palavras-chave: Automação de Testes, iOS, XCUITest, Lua

Abstract

The mobile app development process involves planning and designing the graphical interface, which is typically delivered to the development team through *mockups*, which are a visual and non-functional representation of the product, demonstrating how the components will be present in the interface. Tests, such as user interface (UI), regression, and integration tests, are essential to ensure software quality. However, UI tests are often performed manually, making them time-consuming and error-prone, as they rely entirely on human attention, which, over time and after several repetitions during testing, may lose focus and allow defects to go unnoticed. Automating these tests offers several advantages, but many tools have usability barriers due to the technical knowledge required.

In the case of iOS applications, a widely used framework is *XCUITest*, provided by Apple. Despite its easy integration with the development environment, it has some limitations, such as the requirement that tests be written in the Swift language and the lack of a way to visually verify the application being tested.

This work proposes a framework that simplifies the automation of UI tests in iOS applications through scripts. This framework verifies the integration and appearance of the interface on different devices and configurations, making test automation more accessible, even for developers with limited experience. Tests are executed through the integration of scripts, written in Lua, with XCUITest, which simulates user interactions within the application.

It was possible to implement test cases with a reduced number of lines of code, compared to using only the XCUITest API, and abstracting details of the application's internal implementation. Additionally, it was also possible to visually validate flows within the tested application using pre-defined reference images.

Keywords: Test Automation, iOS, XCUITest, Lua

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Durante o processo de desenvolvimento de um aplicativo, a interface de cada tela nos fluxos planejados é projetada por uma equipe de design que define quais componentes serão usados, suas especificações e como devem ser apresentados ao usuário final. Esses elementos são então entregues à equipe de desenvolvimento por meio de *mockups*, uma representação visual e não funcional do produto demonstrando como os componentes estarão presentes na interface, criados com ferramentas como Figma, Adobe XD, Sketch, InDesign, entre outras. Para garantir que o aplicativo funcione corretamente e que os elementos de design sejam implementados conforme o planejado, as empresas de software realizam diversos tipos de testes, incluindo testes de interface de usuário (UI), de integração e de regressão.

Os testes de UI são frequentemente realizados manualmente, nos quais as pessoas testadoras seguem um roteiro de interações e verificam se a aparência da tela corresponde ao design original representado nos *mockups*. No entanto, esse processo manual consome muito tempo, já que exitem vários dispositivos e configurações onde uma aplicação pode ser executada, e está sujeito a falhas humanas. Portanto, surgiu a necessidade de automação desses testes. Os testes automatizados oferecem diversas vantagens, como redução de custos, testes mais frequentes, detecção precoce de defeitos e melhoria na qualidade do sistema em comparação com os testes manuais [5].

Diversos frameworks e ferramentas foram desenvolvidos para a realização de testes automatizados de UI, tais como o *XCUITest*, o *EarlGrey* e o *Appium* [4] [14]. Essas ferramentas ajudam a validar se as funcionalidades da aplicação estão funcionando conforme o esperado. No entanto, cada ferramenta possui métodos diferentes de verificação e geram resultados distintos para análise. Além da validação das funcionalidades, é importante verificar se a interface está sendo exibida corretamente de acordo com o design original, se os componentes estão na posição correta, se as cores são as mesmas presentes nos *mockups*, entre outros. Essa verificação específica não é comumente oferecida pelas ferramentas por padrão, exigindo que os usuários combinem mais de uma ferramenta para realizar essa validação. Além disso, muitas das ferramentas exigem conhecimento técnico específico, tanto para a integração no projeto quanto para escrever e manter os testes, o que pode ser uma barreira para pessoas sem conhecimento técnico, dificultando a participação delas no processo e aumentando o tempo necessário para que novos membros da equipe se familiarizem com elas.

Nesse contexto, este trabalho propõe o framework *BIUTest* (User Interface Babysitter), que permite a automatização de testes de UI em aplicativos iOS por meio de scripts de fácil compreensão. Esse framework não apenas verifica a correta integração da aplicação, mas também como a interface seria apresentada ao usuário final em diferentes dispositivos e configurações. Sua API simplificada foi projetada para permitir que os casos de teste sejam escritos em poucas

linhas e abstraindo detalhes da implementação interna da aplicação, com o objetivo de facilitar o entendimento por parte de pessoas com pouco conhecimento técnico. Ademais, a estrutura dos casos de teste foi planejada na intenção de ajudar a assegurar o funcionamento dos fluxos importantes da aplicação que forem estipulados.

CAPÍTULO 2

Fundamentação

2.1 Teste de Software

Teste de Software é o processo de avaliação de um sistema para garantir que este atenda aos critérios definidos pelo cliente. O objetivo é identificar discrepâncias entre os resultados reais e esperados, com foco principal na descoberta de bugs, erros ou requisitos ausentes no software ou sistema [24]. É uma etapa muito importante do processo de desenvolvimento do software para a garantia da qualidade do mesmo, e pode ser considerado um investimento.

2.1.1 Pirâmides de Testes

Em *Succeeding with Agile: Software Development Using Scrum* [3] o autor introduz o conceito da pirâmide de testes, uma representação visual sobre a organização lógica dos testes em software. A pirâmide é formada por três camadas: testes unitários, testes de serviço, ou de integração, e testes de UI. A forma de pirâmide é uma metáfora, horizontalmente para a quantidade de testes necessários de cada tipo, e verticalmente para a complexidade e o tempo necessários para a execução dos testes. Existem alguns tipos de pirâmides que podem definir o modelo de testes a ser utilizado e a proporção de cada tipo de teste a ser implementado, dependendo do contexto de cada projeto. A seguir são especificados cada um dos tipos de teste presentes nas pirâmides e uma explicação de dois dos modelos existentes.

2.1.1.1 Testes Unitários

Nessa fase os testes são feitos de maneira isolada em partes menores do sistema, como métodos, funções, classes, ou qualquer parte pequena testável do programa. O objetivo dessa fase é evitar que erros nessas partes menores sejam propagados para níveis mais altos dos testes.

Normalmente eles são escritos pelas próprias pessoas desenvolvedoras, e podem ser feitos durante o desenvolvimento. Os testes unitários constituem a maior parte da base de testes do sistema e são muito rápidos, podendo rodar milhares em poucos minutos.

2.1.1.2 Testes de Serviços

Os testes de serviço, também chamados de testes de integração, são responsáveis por testar a integração do sistemas com outras partes fora da aplicação, como bancos de dados, sistemas de arquivos e chamadas de redes para outras aplicações. Nesse caso é necessário rodar não só a aplicação, como também esses componentes que se comunicam com ela. Por esse motivo esse tipo de teste demanda um tempo maior para ser executado, em comparação com os testes

unitários. Assim como nos testes unitários, é necessário um conhecimento sobre a estrutura interna da aplicação para que os testes sejam escritos, por esse motivo eles também são feitos pelas pessoas desenvolvedoras.

2.1.1.3 Testes de Interface do Usuário (UI)

Os testes de UI são feitos para assegurar que a interface da aplicação está funcionando corretamente. Isso inclui, por exemplo, garantir que uma entrada do usuário acione alguma determinada ação, ou que os dados estão sendo mostrados da forma como foram desenhados para ser.

Atualmente o mais comum é que esse tipo de teste seja realizado de forma manual, onde as pessoas responsáveis pelos testes seguem um determinado roteiro de interações e verificam se o estado da aplicação após cada uma delas é o esperado. No entando, existem ferramentas que são utilizadas para automatizar parte desses processo. Elas simulam as interações de um usuário e verificam de diferentes maneiras se a interface está adequada, como por exemplo verificando a existência de elementos a patir de um identificador ou através de uma análise estática de uma captura de tela. Essas ferramentas diminuem a possibilidade de falhas humanas e aumentam a eficiência do processo de testes, reduzindo o tempo gasto nestas atividades [5].

Esse tipo de teste demanda mais tempo do que os unitários e os de serviço. Apesar disso, mesmo com uma quantidade menor, é possível testar uma boa parte da aplicação.

2.1.1.4 A Pirâmide Clássica

De acordo com a Pirâmide de Testes Clássica, a quantidade de testes unitários deve ser maior que a quantidade de testes de serviço, e este maior que a de testes de UI. O objetivo dessa divisão é manter um equilíbrio entre velocidade e eficácia, além de que as alterações feitas em um nível mais alto da aplicação, na interface e nas APIs, são mais frequentes do que na lógica das funções implementadas nos códigos, por esse motivo necessitam de um esforço maior para serem validadas [23]. Na figura 2.1 é possível ver uma representação da versão clássica da pirâmide.

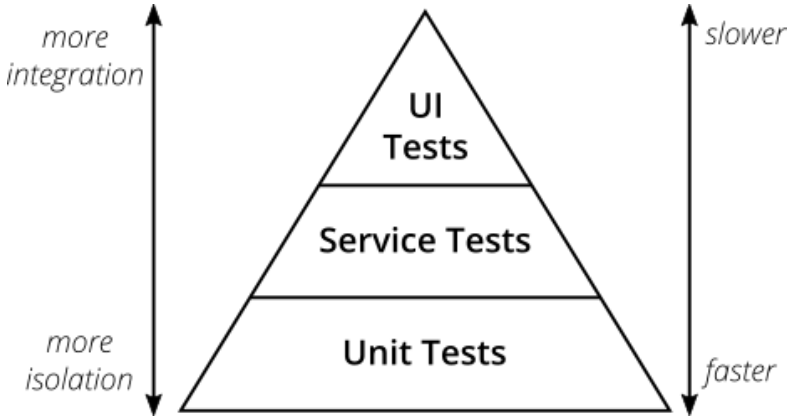

Figura 2.1 A Pirâmide de Testes Clássica. Fonte: The Practical Test Pyramid [3]

2.1.1.5 A Pirâmide Invertida

Outro tipo de modelo de testes é a Pirâmide Invertida, também chamada de Cone de Sorvete. Na Figura 2.2 é possível ver uma representação dela. Nesse modelo é aplicada uma grande cobertuda de testes de UI, com uma grande quantidade de testes manuais, e pouca ou nenhuma cobertura de testes unitários automatizados. Nesse caso os testes são mais focados no resultado de como a aplicação estará nas mãos do usuário final. Esse modelo não é muito recomendado nos dias de hoje, já que os testes de UI são mais demorados e, quando realizados manualmente, são mais suscetíveis a erros despercebidos. Por outro lado, em alguns casos pode fazer sentido o uso dele, como por exemplo em sistemas legados com poucos testes unitários implementados ou em casos onde a aplicação sendo desenvolvida é um protótipo que precisa ser rapidamente validado.

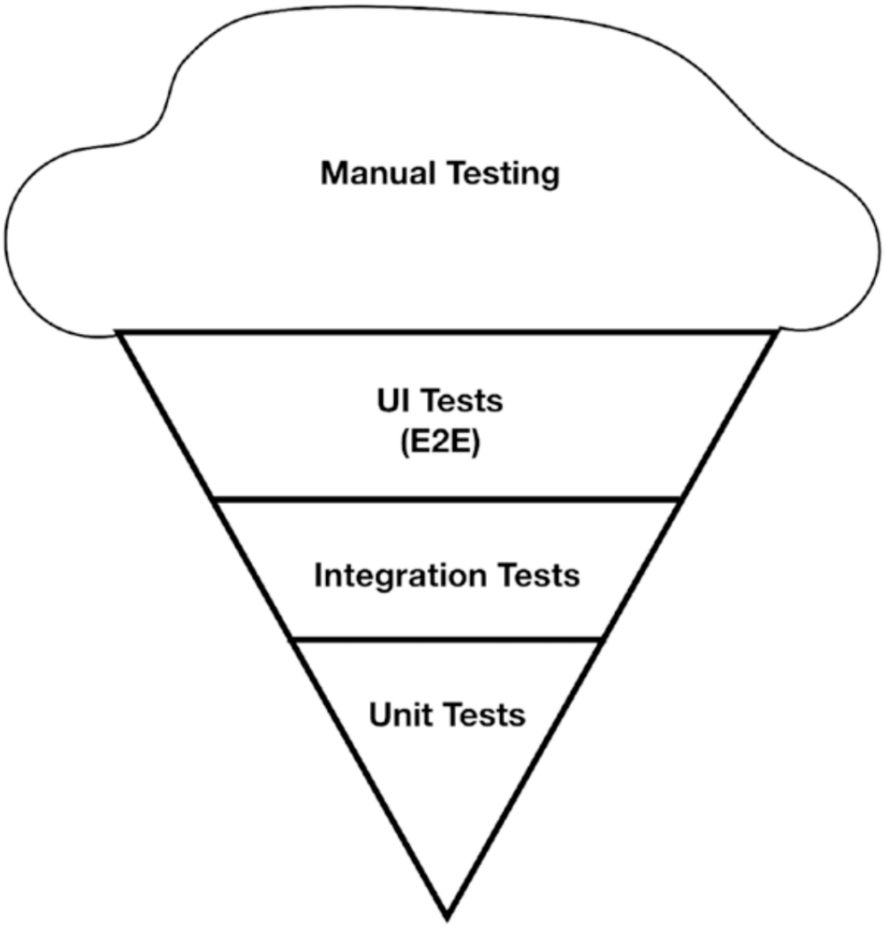

Figura 2.2 A Pirâmide de Testes Invertida. Fonte: Pro iOS Testing [23]

2.1.2 Métodos de Testes

Para a realização dos testes podem ser aplicados diferentes métodos levando em conta o tipo do sistema, os conhecimentos da pessoa realizando os testes, as ferramentas disponíveis e os objetivos da avaliação. Dois exemplos desses métodos são os testes estruturais e os funcionais.

2.1.2.1 Estrutural (Caixa-Branca)

O teste de caixa-branca é uma técnica que testa a utilidade do produto e também a estrutura interna da aplicação [20]. Por esse motivo, a pessoa realizando o teste necessita ter um conhecimento da implementação do programa sendo testado. Os testes unitários, por exemplo, se encaixam nessa categoria.

2.1.2.2 Funcional (Caixa-Preta)

Diferente dos testes caixa-branca, os testes caixa-preta não levam em conta a estrutura interna e como o sistema foi implementado. Nesse método são testadas apenas as funcionalidades do sistema, comparando os resultados obtidos a partir das entradas com os resultados esperados. Os testes de UI se encaixam nessa categoria, são analisadas as características gráficas do programa a partir de cada interação e não se leva em conta o código por trás. Por esse motivo, pessoas sem conhecimento sobre a implementação podem realizar esse tipo de teste.

2.1.3 Testes Automatizados

A automatização de testes é uma técnica que utiliza um software para realizar automaticamente o processo de validação das funcionalidades e dos critérios de um sistema. Testes automatizados podem levar a custos mais baixos, maior frequência de testes, identificação precoce de defeitos e maior qualidade do sistema [1], quando comparados com testes realizados manualmente.

A indústria de software vem se tornando um ambiente cada vez mais rápido e ágil, com ênfase em integração, desenvolvimento e entrega contínuos [17]. Esse ambiente coloca novos requisitos em relação à velocidade de testes e necessita de feedbacks mais rápidos e mais frequentes sobre a qualidade do software. A Integração Contínua (CI) é uma prática no desenvolvimento de software, que envolve as pessoas desenvolvedoras unindo suas alterações de código frequentemente em um repositório central. Posteriormente, são realizados builds e testes automatizados. Os principais objetivos da integração contínua são acelerar a detecção e resolução de bugs, aprimorar a qualidade do software e diminuir o tempo necessário para validar e liberar atualizações de software.

2.2 Testes de UI

Os testes de UI podem ser divididos em duas categorias: o teste das funções da aplicação por meio da interface do usuário e o teste para garantir que a interface do usuário funcione corretamente. Esse último pode ser subdividido em dois tipos: o teste para verificar se uma

integração de interface funciona corretamente e o teste para verificar se uma interface é exibida corretamente [18]. Uma das técnicas utilizadas para verificar se a interface está correta é através de testes de snapshot. Podemos fazer uma especificação da seguinte maneira: enquanto os testes de UI focam em "o que"aparece na interface, os testes de snapshot verificam "como"os elementos aparecem na tela.

É de extrema importância assegurar que aplicativos móveis e web sejam exibidos e funcionem adequadamente em uma variedade de dispositivos e navegadores, o que pode levar bastante tempo devido a diferentes características físicas e configurações de software que o dispositivo pode apresentar. Entre as características físicas importantes estão o tamanho da tela, a densidade de exibição (ou resolução da tela) e a orientação. Enquanto isso, as configurações de software relevantes incluem o idioma do telefone, o tamanho da fonte, o modo de cor/contraste, o nível de brilho, entre outros [7].

Levando isso em consideração, torna-se impraticável a realização de testes de UI exclusivamente de forma manual, uma vez que isso implicaria na necessidade de testar em diversos dispositivos distintos. Portanto, se fez essencial encontrar métodos para automatizar esse processo, possibilitando a realização de testes em uma variedade de dispositivos, cada um com suas próprias configurações, ao mesmo tempo em que se evita a ocorrência de falhas despercebidas. Em [9] os autores simplificam a diferenciação entre testes manuais e automatizados ao sugerir que os testes automatizados são principalmente usados para evitar a ocorrência de novos erros nos módulos funcionais já verificados. Por outro lado, os testes manuais são mais eficazes na descoberta de erros novos e inesperados. Portanto as duas estretégias são complementares e essenciais no processo de garantia de qualidade de uma aplicação.

2.3 Ferramentas de Testes de UI

No contexto das ferramentas de automação de testes de UI, existem várias opções voltadas para diferentes tipos de interfaces, como desktop, web e dispositivos móveis. Essas ferramentas proporcionam recursos para a criação e execução de casos de teste, além de permitirem a realização de uma suíte de testes na qual as pessoas testadoras podem definir um conjunto de testes a serem executados. Nessa seção serão citadas algumas das ferramentas consideradas estado da arte para diferentes plataformas.

2.3.1 Web

Existem diversas ferramentas para realização de testes de UI para aplicações web, em [13] e [19] algumas delas são comparadas identificando suas vantagens e desvantagens. A partir disso foram identificadas duas delas que propõem estratégias diferentes e são algumas das mais utilizadas para essa tarefa. Essas ferramentas são o *Selenium* e o *Katalon Studio*, que serão apresentados a seguir.

2.3.1.1 Selenium

O Selenium é um framework de automação amplamente utilizado no campo de testes de aplicativos da web. É um projeto de código aberto e é compatível com múltiplos sistemas operacionais e linguagens de programação, além de suportar testes em vários tipos de navegadores. No entanto existem algumas desvantagens, como a necessidade de construir projetos do zero, o que pode levar a processos de configuração demorados. Além disso, o Selenium pode não fornecer razões específicas para falhas nos testes, o que adiciona complexidade ao processo de depuração.

2.3.1.2 Katalon Studio

Katalon Studio é um software de código aberto reconhecido por sua capacidade de exportar com facilidade o código de scripts do Selenium, sendo uma possível alternativa à essa ferramenta. É conhecido como uma ferramenta de teste de automação sem a necessidade de conhecimento em programação, capturando as atividades do usuário, localizadores da web e gerando relatórios abrangentes. O Katalon oferece suporte a vários navegadores e sistemas operacionais, o que o torna uma escolha versátil. Ademais, ele se destaca por sua interface de usuário amigável, o que o diferencia de outras ferramentas comerciais e de código aberto.

2.3.2 Android

No site oficial para desenvolvedores Android[1] existe uma página dedicada à explicação de testes de UI automatizados, onde também são sugeridas algumas ferramentas para a realização dos mesmos. Duas delas também são citadas em trabalhos [14] e [21] onde são feitas comparações entre ferramentas de automatização de testes de UI em aplicações Android.

Por meio destes foi possível relacionar três ferramentas que são consideradas estado da arte para esse tipo de teste nessa plataforma: o *Espresso*, o *UI Automator* e o *Appium*, que serão descritos a seguir.

2.3.2.1 Espresso

O Espresso, desenvolvido pelo Google, é uma ferramenta de teste projetada para facilitar a criação de casos de teste destinados a avaliar a interface do usuário. Ele é amplamente reconhecido como uma das principais ferramentas de teste para aplicativos Android. O Espresso oferece flexibilidade para testes de caixa-preta, ao mesmo tempo em que permite a validação de componentes individuais ao longo do processo de desenvolvimento. Os testes no Espresso podem ser escritos em Java ou Kotlin, que são linguagens comumente utilizadas por desenvolvedores nessa plataforma.

2.3.2.2 UI Automator

O manual [26] do UI Automator o descreve como uma ferramenta de teste de interface do usuário adequada para testes funcionais de UI entre aplicativos, abrangendo tanto aplicativos

[1] `https://developer.android.com/`

do sistema quanto aplicativos instalados. Durante os testes, é possível interagir não apenas com a aplicação sendo testada, mas também com todas as outras no dispositivo. Isso se deve à capacidade da ferramenta de acessar as propriedades dos componentes da interface do dispositivo onde os testes estão sendo executados. Dessa forma, o UI Automator pode ser caracterizado como uma ferramenta de teste de caixa-preta, pois não depende da implementação interna da aplicação.

2.3.2.3 Appium

O Appium é uma ferramenta usada para testar aplicações nativas, híbridas, mobile web e para desktops. Ela tem suporte para simuladores e execução em aparelhos reais. A aplicação do Appium funciona através de uma arquitetura cliente-servidor, por isso o processo que está rodando a automação de testes (nesse caso, o servidor) não precisa estar no mesmo local que a aplicação sendo testada (nesse caso, o cliente). A ferramenta utiliza o WebDriver spec[2] como API e, através dos drivers de cada plataforma, converte o protocolo WebDriver para chamadas das bibliotecas específicas de cada uma delas. É necessário ter um certo conhecimento sobre as especificidades da plataforma e da API para escrever os testes, porém eles podem ser escritos em diversas linguagens de programação, como Ruby, Java, Node.js, PHP, C#, Clojure e Perl.

Como é um programa separado da IDE usada para desenvolver a aplicação, ocasionalmente pode ser necessário aguardar uma atualização da ferramenta quando a IDE é atualizada, a fim de evitar problemas nos testes existentes. Além disso, a configuração da aplicação no Appium pode ser um pouco complexa.

2.3.3 iOS

Em [4] o autor faz um estudo comparativo entre as principais ferramentas de automatização de testes de UI para aplicações iOS com o objetivo de demonstrar o estado da arte sobre o assunto. Além dessa, não foram encontradas outras pesquisas focadas especificamente na plataforma iOS, por esse motivo foram buscados em sites bastante utilizados pela comunidade para elencar as quais eram as mais recomendadas para esse tipo de teste. A partir disso, foi possível analisar três ferramentas que mais foram citadas: o *XCUITest*, o *EarlGrey* e o *Appium*, esse último já citado e explicado na seção onde foram trazidas as ferramentas para testes em aplicações Android. A seguir uma breve explicação das outras duas.

2.3.3.1 XCUITest

O XCUITest é um framework criado pela Apple em 2015 para a automatização de testes de UI. Foi construído em cima do XCTest, um framework de testes integrado no XCode, que é o ambiente de desenvolvimento integrado (IDE) da Apple. Os testes podem ser escritos utilizando as linguagens Swift e Objective-C para as aplicações iOS e macOS nativas.

Como o XCUITest vem integrado no XCode, não é necessário nenhum tipo de instalação ou configuração do ambiente para utilizá-lo. Os testes criados utilizando esse framework são rápidos e confiáveis. Além disso, o usuário pode executar os testes como parte do seu processo

[2] `https://w3c.github.io/webdriver`

de CI, e ter feedbacks contínuos nos dispositivos testados.

Os testes construídos no XCUITest são do tipo Caixa-Preta, ou seja, não se tem acesso à instrutura interna do código durante a execução, e nesse caso são rodados em um processo separado da aplicação em si.

2.3.3.2 EarlGrey

O EarlGrey é uma ferramenta desenvolvida pelo Google com o objetivo de melhorar a sincronização com a aplicação durante os testes, evitando comportamentos não-determinísticos e, por consequência, os chamados testes "flaky", onde um teste executado várias vezes sem mudanças de ambiente produz resultados diferentes. Em [22] é apontado que essas melhorias foram feitas baseadas em outra ferramenta de autoria do Google, o *Espresso*, que foi desenvolvido justamente para mitigar esses problemas nos testes de UI em aplicações Android.

Os testes construídos com o EarlGrey são do tipo Caixa-Cinza. Apesar de simularem interações de usuários reais, a pessoa desenvolvedora precisa ter acesso ao código para lidar com os elementos da tela. Os testes são executados no mesmo processo da aplicação sendo testada, por esse motivo não é possível interagir com telas de fora dela, como dialogs do sistema por exemplo.

2.4 Trabalhos Relacionados

À medida que as tecnologias empregadas no desenvolvimento de aplicativos móveis avançam, é fundamental que os frameworks de teste acompanhem esse progresso. Essa evolução é essencial para viabilizar a avaliação de novos componentes e funcionalidades, possibilitando a realização de validações nos produtos e contribuindo de maneira significativa para a qualidade do software. Uma categoria de validação de extrema importância é a validação visual, que visa garantir que o usuário interaja e visualize a aplicação da maneira prevista durante o desenvolvimento. O teste visual assegura o correto funcionamento da interface do usuário (UI) e engloba tanto a validação da integração adequada da interface quanto a verificação da exibição correta da mesma [18]. Neste trabalho, é proposta uma ferramenta que se enquadra em ambas categorias de validação visual, com foco em aplicações desenvolvidas para dispositivos iOS.

Em [4], foi realizado um estudo comparativo entre três das ferramentas mais utilizadas para a execução de testes de UI na plataforma iOS, com ênfase na validação do funcionamento da integração dos componentes na aplicação, sem abordar a conformidade visual. Da mesma forma, em [14], um trabalho semelhante foi conduzido, porém com foco em aplicações para a plataforma Android.

Por sua vez, [16] propôs uma ferramenta para a geração de casos de teste destinados a aplicações multiplataforma, porém os testes são gerados com base nos componentes visuais presentes na aplicação, sem considerar as regras de negócio ou os fluxos mais críticos, o que resulta em uma validação menos focada nos casos de uso da aplicação. Uma abordagem que leva bastante esse ponto em consideração é o Behavior Driven Development (BDD), em [11] são citadas diversas vantagens dessa estratégia, como a importância dada às necessidades e valores de negócio e a colaboração e visibilidade do time, mas também algumas desvantagens,

como a dificuldade de manutenção e uma sobrecarga em times onde somente as pessoas testadoras participam do processo de testes.

Já em [18], diversas aplicações da plataforma Android, com diferentes configurações, foram analisadas por meio de uma abordagem que combinou avaliação manual e análise estática de capturas de tela, utilizando uma ferramenta de automação de testes. O objetivo era classificar diferentes tipos de defeitos que podem ser encontrados em aplicações. Além disso, [6] propôs uma ferramenta de automação de testes de interface, que realiza uma análise em tempo de execução, mas com foco na verificação da acessibilidade de aplicações Android. Em adição, [12] faz a proposta de uma ferramenta para análise da interface de aplicações iOS através de uma representação formal dos elementos com suas propriedades, porém essa técnica não permite que o resultado das telas seja visto com o objetivo de ser validado visualmente.

Por fim, [15] apresentou um framework e uma ferramenta de automação de testes, semelhantes ao proposto neste trabalho, porém direcionados especificamente para aplicações na plataforma Android.

CAPÍTULO 3

Solução

Como mencionado anteriormente, os testes destinados a verificar o correto funcionamento da interface do usuário podem ser divididos em dois tipos: testes de integração de interface e testes de exibição da interface. A solução proposta neste trabalho se encaixa em uma combinação dessas duas categorias.

Nos testes de UI manuais, é comum utilizar técnicas baseadas em *checklists*, nos quais listas contendo vários casos de teste são criadas apontando erros comuns em cada um deles. As pessoas testadoras seguem os passos dos casos de uso, verificando se a interface funciona e é exibida corretamente. Essa mesma abordagem é aplicada na solução proposta, porém de forma automatizada. Casos de teste são definidos e, por meio de uma sequência de instruções em cada um deles, as interações do usuário são simuladas e a interface é comparada com uma imagem de referência previamente definida.

Nessa seção será proposto um framework chamado *BIUTest*, que tem o objetivo de assegurar o correto funcionamento e a consistência visual de aplicações iOS através de testes de UI automatizados utilizando scripts de fácil legibilidade e manutenção.

3.1 Seleção do Ferramental

O BIUTest é composto por duas partes que conversam entre si: o script, contendo os casos de teste a serem executaods, e o motor do framework. Os scripts são escritos em Lua e o framework que se comunica com o dispositivo é escrito em Swift através do XCUITest. A seguir são apresentados os motivos levados em consideração para a seleção dessas duas ferramentas.

3.1.1 Lua

Lua é uma linguagem de script interpretada de alto nível que foi concebida em 1993 no Tecgraf, um laboratório vinculado ao Departamento de Informática da PUC-Rio. Seu propósito inicial era a extensão de aplicações em geral, prototipagem e a incorporação em software complexos, como jogos. Atualmente, Lua é amplamente empregada em diversas aplicações industriais, incluindo softwares como o Photoshop e o Lightroom da Adobe, com foco especial em sistemas embarcados, como o middleware Ginga para TV digital, e jogos populares, como World of Warcraft e Angry Birds[1].

Lua é reconhecida por sua simplicidade e velocidade, sem sacrificar seu poder de funcionalidade. Apesar de oferecer poucas estruturas iniciais, a linguagem permite a implementação

[1] `https://www.lua.org/portugues.html`

de novas estruturas utilizando os recursos disponíveis. Além disso, Lua é altamente portátil, já que é distribuída em um pacote compacto - seu código-fonte tem menos de 1 MB - e pode ser compilada sem alterações em todas as plataformas com um compilador C padrão, abrangendo desde sistemas Unix e Windows até dispositivos móveis, microprocessadores e mainframes. Além disso, Lua é um software de código aberto, o que significa que pode ser utilizado e adaptado para qualquer finalidade.

Conforme evidenciado por [25], linguagens de script geralmente são mais acessíveis para aprender do que linguagens de programação convencionais, como C++ ou Java, devido à capacidade de abstrair complexidades desnecessárias durante o processo de aprendizado. Isso faz com que pessoas com menos conhecimento técnico possam usá-las com facilidade.

Por sua portabilidade, o uso de Lua permite que uma solução semelhante à proposta nesse trabalho seja replicada em outras plataformas, como em Android por exemplo. Dessa maneira uma mesma pasta correspondente a um caso de teste pode ser utilizada para a execução dos testes em ambas as plataformas. Essas características combinadas fizeram de Lua uma escolha ideal para ser a linguagem de script utilizada na criação dos testes no *BIUTest*.

3.1.2 XCUITest

Como citado na Seção 2.3 de Ferramentas de Testes de UI, o XCUITests e o EarlGrey são algumas das mais utilizadas para a realização dessa tarefa. A partir disso foi feita uma busca no site Github[2] com o objetivo de analisar projetos que se encaixam no alvo de utilização do framework proposto nesse trabalho. A busca foi feita filtrando os repositórios públicos que continham projetos para a plataforma iOS, feitos utilizando a linguagem Swift, que tinham mais de 200 *stars* ou *forks* - indicando que são utilizados de exemplo por diversas pessoas desenvolvedoras -, e que foram atualizados pelo menos após 2020. Além disso não foram levados em consideração projetos que continham tutoriais e projetos que não tinham nenhum tipo de teste implementado.

A partir dessa busca foram selecionados 50 repositórios que foram analisados para verificar os frameworks mais utilizados para a realização de testes. Desses 50, 23 deles (46%) contêm apenas testes unitários, sendo que 12 deles (52%) implementam algum tipo de teste de interface através de comparação visual de componentes por meio de testes de snapshot. Os outros 26 projetos restantes (52%) implementam testes de UI e todos eles utilizam o XCUITest como ferramenta para tal. Ademais, 2 deles também implementam o EarlGrey, podendo indicar que alguma dessas ferramentas pode apresentar algum tipo de restrição de uso que nesses casos tiveram que ser supridas por outra. Além disso, foi observado que 5 das ferramentas que possuem testes de UI (19%) fizeram algum tipo de implementação para garantia visual através de comparação com screenshots.

Levando em conta a grande presença do framework XCUITest nos projetos onde foram realizados testes de UI, ele foi escolhido para ser utilizado na simulação de interações com a aplicação no framework proposto nesse trabalho.

[2] `https://github.com`

3.2 Exemplo de Uso

A seguir é mostrado um exemplo de uso da ferramenta. Foi utilizada a aplicação de exemplo disponível no repositório da ferramenta. A aplicação tem um campo de texto, dois botões "sign in"e "sign up"e um texto que fica na parte de baixo. São feitos dois casos de teste, **sign-in** e **sign-up**.

- **Sign In:** Nesse caso de teste é apertado o botão "sign in", então uma mensagem aparece informando que o campo de texto está vazio. Um texto é escrito no campo de texto e o botão "sign in"apertado novamente, mostrando então uma mensagem de sucesso. O script utilizado para esse caso de teste pode ser visto na Figura 3.1, e as imagens de referência geradas podem ser vistas na Figura 3.2.

```
1  biu.compare_ref()
2  biu.tap_button("sign in")
3  biu.compare_ref()
4  biu.enter_text { text = "biu@email.com", field = "email" }
5  biu.tap_button("sign in")
6  biu.compare_ref()
```

Figura 3.1 Caso de Teste 1. Fonte: Elaborado pela autora

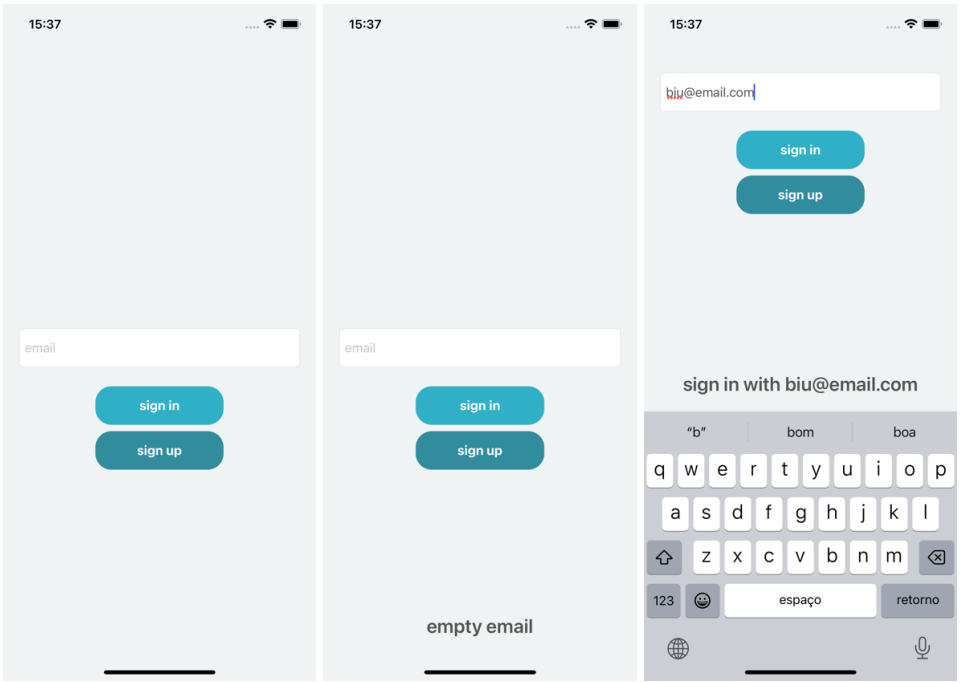

Figura 3.2 Referências Caso de Teste 1. Fonte: Elaborado pela autora

- **Sign Up:** Nesse caso de teste é apertado o botão "sign up", então uma mensagem aparece requisitando a entrada de um email no campo de texto e o botão "sign in"desaparece. O botão é apertado novamente e uma mensagem aparece informando que o campo de texto está vazio. Um texto é escrito no campo de texto e o botão "sign up"apertado novamente, mostrando então uma mensagem de sucesso. O script utilizado para esse caso de teste pode ser visto na Figura 3.3, e as imagens de referência geradas podem ser vistas na Figura 3.4.

```
1  biu.compare_ref()
2  biu.tap_button("sign up")
3  biu.compare_ref()
4  biu.tap_button("sign up")
5  biu.compare_ref()
6  biu.enter_text { text = "biu@email.com", field = "email" }
7  biu.tap_button("sign up")
8  biu.compare_ref()
```

Figura 3.3 Caso de Teste 2. Fonte: Elaborado pela autora

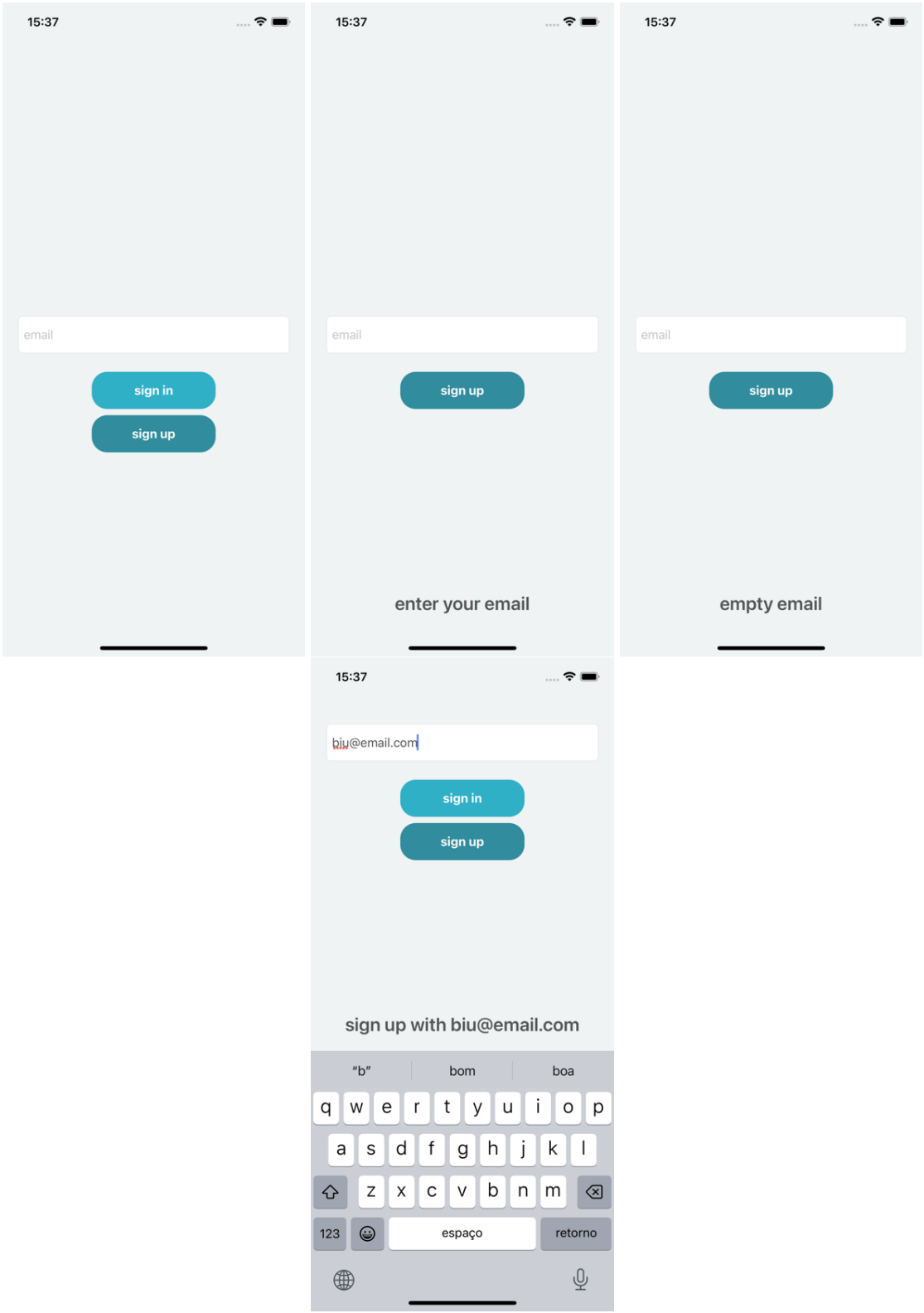

Figura 3.4 Referências Caso de Teste 2. Fonte: Elaborado pela autora

3.3 Arquitetura da Solução

O motor do framework foi escrito em Swift e tem duas responsabilidades: fazer a comunicação entre o script e o código nativo que invocará o XCUITest e preparar o ambiente para a execução dos testes.

A comunicação entre o script e o código nativo ocorre por meio de uma máquina virtual (VM) que executa todo o código Lua. Isso garante o isolamento, evitando que um erro crítico afete a aplicação. O código-fonte do Lua inclui uma API C que oferece um conjunto de funções disponíveis para que o programa se comunique com o Lua. Isso acontece através de uma pilha virtual que faz a transferência de valores entre as duas partes. Cada elemento na pilha representa um valor Lua, como nulo, número, string, etc. A VM implementada utiliza essa API para a comunicação entre o código nativo e o código Lua, acessando a pilha virtual.

Embora seja possível realizar conversões de cada tipo de valor do Lua para Swift e vice-versa, a estratégia adotada foi simplificar a implementação colocando apenas strings e ponteiros para funções nativas na pilha. Cada função Swift registrada é guardada na memória e, para cada uma delas, um ponteiro é gerado e adicionado à pilha. Ao chamar uma função no script Lua, os parâmetros enviados são inicialmente codificados para uma string contendo um objeto JSON e então o ponteiro para a função registrada é chamado. Com isso, a função é invocada no código nativo recebendo essa string, que é então decodificada para o objeto que ela espera receber como parâmetro.

No framework, as funções Swift registradas na VM através da API no Swift são responsáveis por chamar as funções da API do XCUITest, que irão de fato se comunicar com a aplicação sendo testada simulando as interações do usuário. Na Figura 3.5 é possível ver um esquema das partes do framework. Em verde estão as partes nativas, em Swift, em rosa estão as partes em Lua e em azul a comunicação entre as partes.

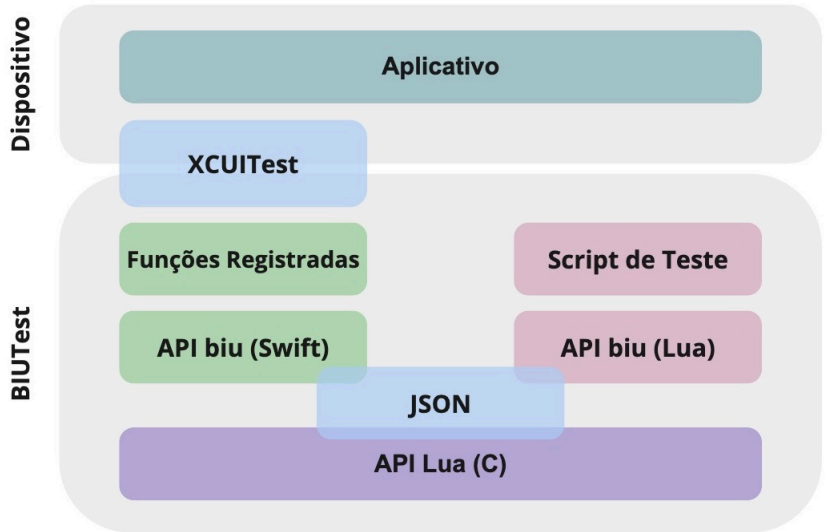

Figura 3.5 Arquitetura da Solução. Fonte: Elaborado pela autora

A seguir, nas Figuras 3.6 e 3.7, é mostrado um exemplo de como uma função pode ser registrada e, em seguida, chamada no script Lua. Foi registrada uma função chamada *enter_text*, que aceita como parâmetro uma *struct* do tipo *EnterTextData*, contendo dois campos: *field*, o identificador do campo de texto, e *text*, o texto a ser escrito no campo de texto. No exemplo de script dado, o texto "biu@email.com"é escrito no campo de texto com identificador "email".

```
1  struct EnterTextData: Codable {
2      let text: String
3      let field: String
4  }
5
6  BIUTest.register("enter_text") { (data: EnterTextData) in
7      let textField = self.app.textFields[data.field]
8      textField.typeText(data.text)
9  }
```

Figura 3.6 Registro de Função. Fonte: Elaborado pela autora

```
1  biu.enter_text { text = "biu@email.com", field = "email" }
```

Figura 3.7 Uso de Função. Fonte: Elaborado pela autora

Algumas funções mais comuns nos casos de teste são cadastradas por padrão no framework, como por exemplo **tap_button**, **enter_text**, **scroll_up**, **scroll_down**, **wait**, entre outras. Essas funções podem ser utilizadas nos scripts sem a necessidade de escrever o código em Swift demonstrado na Figura 3.6 de registro de função. O usuário pode tanto sobrescrever essas funções quanto cadastrar novas para serem chamadas através do script. Existem duas funções especiais que realizam comportamentos particulares: **compare_ref** e **save_ref**.

- **save_ref:** Tira um *screenshot* do estado atual da tela e salva na pasta de imagens de referências de acordo com o modelo e a versão do iOS do dispositivo onde o teste está sendo executado.

- **compare_ref:** Tira um *screenshot* do estado atual da tela e compara com a imagem de referência correspondente, seguindo a ordem da lista de imagens salvas.

A preparação do ambiente para a execução dos testes será demonstrada a seguir, na Seção 3.4.2.

3.4 Algoritmo da Solução

3.4.1 Estrutura de Pastas

Para a execução dos testes, a ferramenta acessa uma pasta chamada **biu-test** que deve ser referenciada no target de testes de UI no projeto Xcode da aplicação. A estrutura da pasta pode ser vista na Figura 3.8.

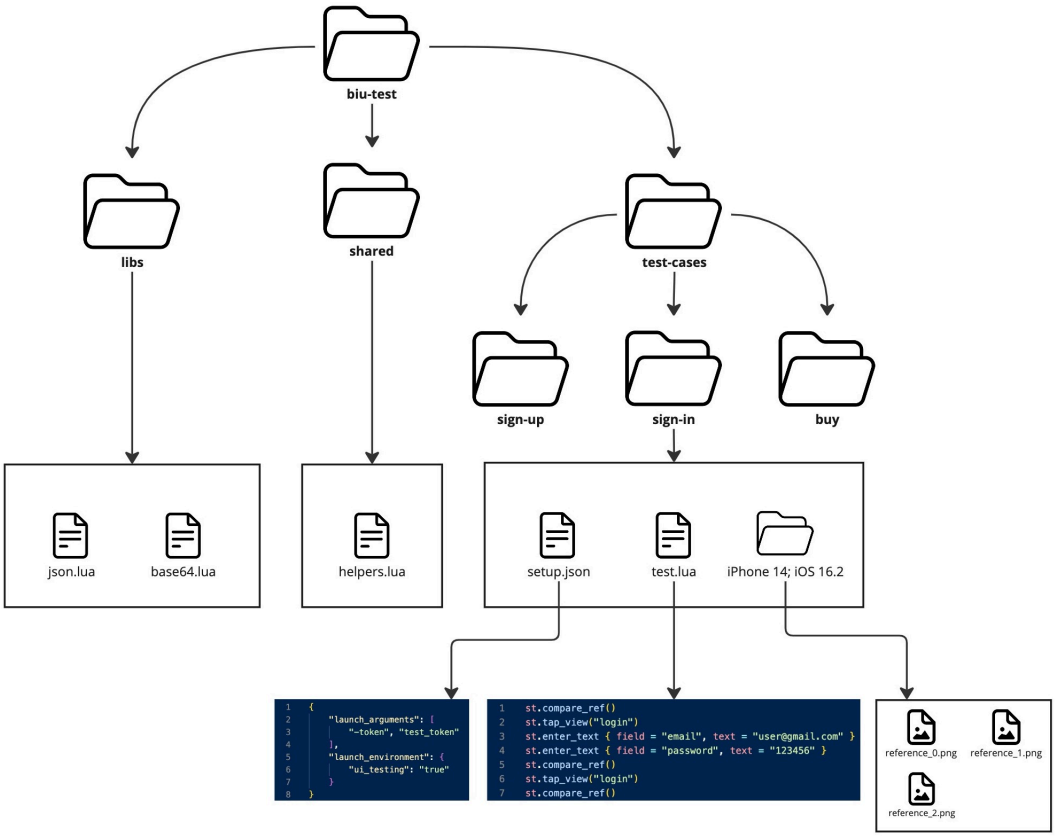

Figura 3.8 Estrutura de Pastas. Fonte: Elaborado pela autora

Na raiz existem três pastas:

- **libs:** Contém arquivos do tipo *.lua* com bibliotecas, que podem ser de terceiros. Os arquivos dessa pasta são carregados primeiro na VM do Lua, dessa maneira podem ser utilizadas pela pessoa desenvolvedora durante os testes. No exemplo, existem uma biblioteca para lidar com objetos JSON e outra para uso de base64;

- **shared:** Contém arquivos do tipo *.lua* contendo variáveis e funções que podem ser utilizadas por todos os casos de teste. Os arquivos dessa pasta são carregados na VM do Lua após os arquivos da pasta "libs". No exemplo foi adicionado um arquivo *helpers.lua*;

- **test-cases:** Possui n pastas contendo um caso de teste diferente em cada uma. Essa pasta inclui um arquivo obrigatório *test.lua*, contendo o script com a sequência de interações a serem executadas durante o teste, um arquivo opcional *setup.json* contendo argumentos e variáveis de ambiente que devem ser usadas durante a execução do caso de teste, e n pastas contendo as imagens de referência do caso de teste. Cada pasta é nomeada de acordo com o dispositivo em que as imagens de referência foram geradas, seguindo a convenção <*Modelo do Dispositivo*><*Versão do iOS*>. As imagens dentro das pastas devem ser nomeadas *reference_*<*number*>*.png*, onde <*number*> corresponde à ordem da imagem a ser comparada, ou seja, *reference_0.png* será a primeira a ser comparada, seguida de *reference_1.png*, e assim sucessivamente. Essas imagens são automaticamente geradas e salvas durante a execução dos testes, usando a função **save_ref**. Portanto, para atualizá-las, basta utilizar essa função nos pontos em que as comparações devem ser feitas e executar o caso de teste correspondente. Depois de geradas as imagens, a pessoa escrevendo os testes substitui **save_ref** por **compare_ref** para realizar as comparações.

3.4.2 Execução dos Testes

A execução da solução é feita em duas etapas que funcionam sequencialmente: a configuração do ambiente e a execução dos casos de teste. Ao iniciar a execução dos testes, os arquivos presentes na pasta "libs"são carregados na VM do Lua para que as bibliotecas fiquem disponíveis para serem utilizadas nos casos de teste, em seguida o mesmo acontece com os arquivos da pasta "shared". Logo após, as funções *Swift* padrão da ferramenta são registradas na VM do Lua para que seja possível chamá-las a partir do script. Em seguida são registradas as funções *Swift* definidas pela pessoa desenvolvedora, caso seja necessário.

Após a etapa de configuração, se inicia a execução dos casos de teste. A ferramenta entra em cada uma das pastas presentes em "test-cases", carrega os arquivos *setup.json* e as imagens de referência presentes na pasta e em seguida e executa o arquivo *test.lua* na VM configurada. Cada função previamente registrada é chamada de acordo com o script, e uma ação é refletida na aplicação. Em especial, a função *compare_ref* ativa a execução da comparação do estado atual da tela com a imagem de referência atual. Na Figura 3.9 é mostrado um esquema da execução dos testes.

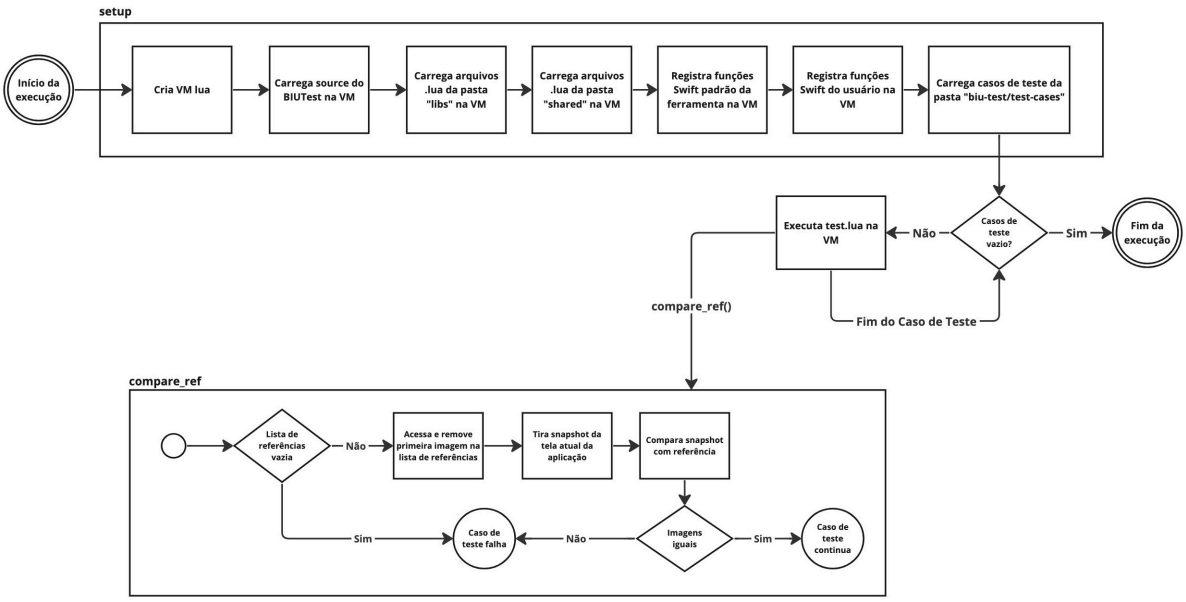

Figura 3.9 Execução dos Testes. Fonte: Elaborado pela autora

CAPÍTULO 4

Análise

4.1 Metodologia

Para a análise do framework proposto nesse trabalho foi escolhida uma aplicação seguindo um contexto em que ele seria usado na realidade, open source e que já incluía testes de UI na sua implementação. A partir disso foram selecionados casos de testes implementados na aplicação e esses foram reescritos utilizando o *BIUTest*. Com isso os códigos e resultados foram comparados levando em conta critérios selecionados previamente.

4.1.1 Seleção da Aplicação

Durante o processo de seleção de ferramentas mencionado na Seção 3.1, ao analisar os repositórios no GitHub, foram observados projetos com diversos níveis de maturidade e cobertura de testes. A aplicação a ser utilizada nas análises foi selecionada com base em vários critérios, como o número de *stars* e *forks* que o repositório recebeu, o status de ser mantido por uma empresa consolidada com uma base de usuários significativa, a presença de uma boa cobertura de testes de UI, bem como a disponibilidade de uma licença que permitisse a realização dos testes neste trabalho.

O projeto selecionado para conduzir as análises foi a aplicação de teste da SDK iOS do Stripe[1]. Essa SDK oferece um serviço de pagamento online e fornece a infraestrutura técnica necessária para prevenção de fraudes e operações bancárias. O repositório do projeto[2] pode ser encontrado no Github.

4.1.2 Seleção dos Casos de Teste

Após a seleção da aplicação foram explorados os casos de teste de UI já implementados no projeto. Uma suíte de testes em particular se tornou interessante para servir como objeto de análise por possuir 10 casos de testes, sendo 5 pares onde cada par consiste no mesmo caso de teste usando configurações de linguagem diferentes: um com a aplicação sendo executada em inglês e o outro em francês. Assim foi identificada uma oportunidade de aprimorar esses casos de teste utilizando o framework *BIUTest* proposto nesse trabalho. Os casos de teste verificam a integração da SDK em uma aplicação de teste, validando diversas funcionalidades através da simulação de interações com a aplicação e uma verificação do estado final após essas serem realizadas. A seguir são especificados os casos de testes que serão utilizados para comparação

[1] `https://stripe.com/br`
[2] `https://github.com/stripe/stripe-ios`

entre a ferramenta usada no projeto e a ferramenta aqui proposta:

- **Transação Simples:** Esse caso de teste simula uma compra simples com um cartão. Inicialmente são selecionados alguns produtos e em seguida o botão "Buy Now"é apertado. Na tela de checkout o botão "Pay from"é apertado direcionando o usuário para a tela de seleção de forma de pagamento. O cartão *Visa* de final "4242"é selecionado e o usuário volta para a tela de checkout. Ao apertar em "Buy"é mostrado um alerta confirmando o sucesso da compra. A Tabela 4.1 fornece uma descrição formal desse caso de uso e na Figura A.2 é possível ver como foi implementado originalmente na aplicação.

Caso de Teste 1	
Funcionalidade:	Transação Simples
Objetivo:	Realizar uma compra simples com um cartão
Etapas:	
1	Apertar o botão "Settings"
2	Apertar o botão "None"na seção "Require Shipping Address Fields"
3	Apertar o botão "Done"
4	Selecionar o vestido, sapato marrom e sapato vermelho
5	Apertar o botão "Buy Now"
6	Apertar o botão "Pay from"
7	Apertar o botão "Visa ending in 4242"
8	Verificar se foi redirecionado para a tela de checkout com os produtos e modo de pagamento corretos selecionados
9	Apertar o botão "Buy"
10	Verificar se aparece o alerta com mensagem de sucesso
11	Apertar o botão "OK"

Tabela 4.1 Caso de Teste 1

- **Checkout Seguro:** Esse caso de teste simula uma compra com um cartão que necessita de autenticação por parte do banco. Inicialmente são selecionados alguns produtos e em seguida o botão "Buy Now"é apertado. Na tela de checkout o botão "Pay from"é apertado direcionando o usuário para a tela de seleção de forma de pagamento. O cartão *Visa* de final "3220"é selecionado e o usuário volta para a tela de checkout. Ao apertar em "Buy"aparece a tela de autenticação, os botões "Learn more about authentication"e "Need help?"são apertados para mostrar uma ajuda sobre essa funcionalidade e então o botão "Complete Authentication"é selecionado, um alerta é mostrado confirmando o sucesso da compra. A Tabela 4.2 fornece uma descrição formal desse caso de uso e na Figura A.5 é possível ver como foi implementado originalmente na aplicação.

Caso de Teste 2	
Funcionalidade:	Checkout Seguro
Objetivo:	Realizar uma compra com um cartão que necessita de autenticação por parte do banco
Etapas:	
1	Apertar o botão "Settings"
2	Apertar o botão "None"na seção "Require Shipping Address Fields"
3	Apertar o botão "Done"
4	Selecionar o vestido, sapato marrom e sapato vermelho
5	Apertar o botão "Buy Now"
6	Apertar o botão "Pay from"
7	Apertar o botão "Visa ending in 3220"
8	Verificar se foi redirecionado para a tela de checkout com os produtos e modo de pagamento corretos selecionados
9	Apertar o botão "Buy"
10	Verificar se aparece tela de autenticação de cartão
11	Apertar o botão "Learn more about authentication"
12	Apertar o botão "Need help?"
13	Verificar se aparecm, os textos de explicação
14	Apertar o botão "Continue"
15	Apertar o botão "Buy"
16	Verificar se aparece o alerta com mensagem de sucesso
17	Apertar o botão "OK"

Tabela 4.2 Caso de Teste 2

- **Pagamento com Apple Pay:** Esse caso de teste simula uma compra utilizando o *Apple Pay*[3]. Inicialmente são selecionados alguns produtos e em seguida o botão "Buy Now"é apertado. Na tela de checkout o botão "Pay from"é apertado direcionando o usuário para a tela de seleção de forma de pagamento. A opção "Apple Pay"é selecionada e o usuário volta para a tela de checkout. Ao apertar em "Buy"aparece o modal de validação da forma de pagamento. A Tabela 4.3 fornece uma descrição formal desse caso de uso e na Figura A.8 é possível ver como foi implementado originalmente na aplicação.

Caso de Teste 3	
Funcionalidade:	Pagamento com Apple Pay
Objetivo:	Realizar uma compra utilizando o Apple Pay como forma de pagamento
Etapas:	
1	Apertar o botão "Settings"
2	Apertar o botão "None"na seção "Require Shipping Address Fields"
3	Apertar o botão "Done"
4	Selecionar o vestido, sapato marrom e sapato vermelho
5	Apertar o botão "Buy Now"
6	Apertar o botão "Pay from"
7	Apertar o botão "Apple Pay"
8	Verificar se aparece o modal de validação da forma de pagamento

Tabela 4.3 Caso de Teste 3

[3] `https://www.apple.com/br/apple-pay`

- **Adicionar Novo Cartão:** Esse caso de teste simula a adição de um novo cartão como forma de pagamento. Inicialmente são selecionados alguns produtos e em seguida o botão "Buy Now"é apertado. Na tela de checkout o botão "Pay from"é apertado direcionando o usuário para a tela de seleção de forma de pagamento. A opção "Add New Card"é selecionada e o usuário é levado para a tela de adição de cartão. São adicionados dados de um cartão e o botão "Done"é apertado. Aparece um alerta informando que o cartão está vencido e então os dados são atualizados. É selecionado o botão "Done"novamente e o usuário retorna para a tela de checkout. Então o botão "Buy"é apertado e aparece um alerta informando que houve um erro que o cartão foi recusado. A Tabela 4.4 fornece uma descrição formal desse caso de uso e na Figura A.11 é possível ver como foi implementado originalmente na aplicação.

Caso de Teste 4	
Funcionalidade:	Adicionar Novo Cartão
Objetivo:	Realizar a adição de um novo cartão como forma de pagamento
Etapas:	
1	Apertar o botão "Settings"
2	Apertar o botão "None"na seção "Require Shipping Address Fields"
3	Apertar o botão "Done"
4	Selecionar o vestido, sapato marrom e sapato vermelho
5	Apertar o botão "Buy Now"
6	Apertar o botão "Pay from"
7	Apertar o botão "Add New Card. . . "
8	Verificar se aparece a tela de adição de cartão
9	Adicionar o texto "4000000000000069"no campo de número do cartão
10	Adicionar o texto "02/28"no campo de data de validade do cartão
11	Verificar se atualiza a tela mostrando a parte de trás do cartão
12	Adicionar o texto "223"no campo de CVC do cartão
13	Adicionar o texto "90210"no campo de código postal
14	Apertar o botão "Done"
15	Verificar se aparece o alerta com a mensagem de cartão expirado
16	Apertar o botão "OK"
17	Remover os últimos 4 caracteres no campo de número do cartão
18	Adicionar o texto "0341"no campo de número do cartão
19	Apertar o botão "Done"
20	Apertar o botão "Buy"
21	Verificar se aparece o alerta com a mensagem de cartão recusado
22	Apertar o botão "OK"

Tabela 4.4 Caso de Teste 4

- **Opção de Pagamento Padrão:** Este caso de teste tem como objetivo verificar a persistência das opções de pagamento na tela de escolha e, em caso de logout, garantir que a opção padrão esteja selecionada. Inicialmente são selecionados alguns produtos e em seguida o botão "Buy Now"é apertado. Na tela de checkout o botão "Pay from"é apertado direcionando o usuário para a tela de seleção de forma de pagamento. Aqui, verifica-se se o método de pagamento padrão é "Apple Pay"e, em seguida, seleciona-se o cartão Visa com final "3220". A simulação do retorno à tela de produto é realizada e o botão "Buy Now"é pressionado novamente. É feita uma verificação para garantir que a opção selecionada anteriormente seja mantida e, em seguida, o botão "Apple Pay"é pressionado. Esses passos são repetidos para assegurar que, ao sair e retornar à tela, a opção selecionada permaneça inalterada. Em seguida, o cartão Visa com final "3220"é selecionado novamente como método de pagamento e o usuário faz logout na tela de configurações. Após isso, simula-se novamente a entrada na tela de seleção de método de pagamento e verifica-se se a opção padrão, ou seja, "Apple Pay", está selecionada. A Tabela 4.5 fornece uma descrição formal desse caso de uso e nas Figuras A.14 e A.15 é possível ver como foi implementado originalmente na aplicação.

Caso de Teste 5	
Funcionalidade:	Opção de Pagamento Padrão
Objetivo:	Verificar a persistência das opções de pagamento na tela de escolha e, em caso de logout, garantir que a opção padrão esteja selecionada
Etapas:	
1	Apertar o botão "Settings"
2	Apertar o botão "None"na seção "Require Shipping Address Fields"
3	Apertar o botão "Done"
4	Selecionar o vestido, sapato marrom e sapato vermelho
5	Apertar o botão "Buy Now"
6	Apertar o botão "Pay from"
7	Verificar se a opção "Apple Pay"está selecionada
8	Apertar o botão "Visa ending in 3220"
9	Apertar o botão "Products"
10	Apertar o botão "Buy Now"
11	Apertar o botão "Pay from"
12	Verificar se a opção "Visa ending in 3220"está selecionada
13	Apertar o botão "Apple Pay"
14	Apertar o botão "Products"
15	Apertar o botão "Buy Now"
16	Apertar o botão "Pay from"
17	Verificar se a opção "Apple Pay"está selecionada
18	Apertar o botão "Visa ending in 3220"
19	Apertar o botão "Products"
20	Apertar o botão "Settings"
21	Apertar o botão "Log out"
22	Apertar o botão "Done"
23	Apertar o botão "Buy Now"
24	Apertar o botão "Pay from"
25	Verificar se a opção "Apple Pay"está selecionada

Tabela 4.5 Caso de Teste 5

4.1.3 Seleção dos Critérios de Avaliação

Diversos fatores podem ser utilizados para medir a qualidade de um framework de automatização de testes, incluindo fatores que dependem do contexto em que ele está sendo utilizado, como o tamanho do projeto, a quantidade de pessoas envolvidas no desenvolvimento e na garantia de qualidade, os recursos disponíveis, entre outros. Mesmo assim, é possível determinar alguns requisitos que catacterizam uma boa ferramenta para essa tarefa. Alguns trabalhos foram feitos realizando a comparação de frameworks de automatização de testes [8] [14], e a partir desses foi possível chegar em pontos a serem analisados na avaliação do framework proposto nesse trabalho. Abaixo é apresentado quais são esses pontos.

- **API:** É importante que o framework tenha uma API simples que não necessite que a pessoa testadora escreva uma grande quantidade de código para realizar operações simples, como apertar um botão por exemplo. Além disso, a clareza do código torna mais fácil garantir a precisão dos testes e simplifica as futuras alterações;
- **Suporte a Logs:** É importante que a pessoa testando tenha acesso a logs claros do que acontece durante os testes para que, no caso de algum erro, seja fácil encontrar onde ele está;
- **Suporte a uma grande variedade de propriedades:** Um dos objetivos de uma ferramenta de automação de testes de UI é analisar as propriedades da aplicação. Contudo, é difícil prever quais propriedades serão pertinentes em todas as análises. Portanto, o framework deve disponibilizar um conjunto adequado de abstrações para que o usuário possa especificar as propriedades relevantes.
- **Tempo de Execução:** Os testes de UI estão no topo da pirâmide de teste, o que demonstra que é o tipo de teste que mais demanda tempo, o que por consequência acarreta em mais custo. Por esse motivo é necessário que o uso da ferramenta não adicione muito tempo ao processo de testes;

4.2 Preparação

A seguir são apresentadas as implementações dos casos de teste no código original presente no repositório da aplicação selecionada e os mesmos casos implementados utilizando o *BIUTest*. Os testes foram executados utilizando o simulador nativo do XCode. O modelo do simulador foi o iPhone 14 com sistema operacional iOS 16.4. A máquina onde os testes foram executados é um MacBook Pro versão 2015[4] com sistema operacional macOS Monterey.

4.2.1 Setup

Antes de começar a executar os casos de teste, é necessário realizar uma configuração inicial para fornecer propriedades personalizadas que serão utilizadas pela aplicação durante a sua

[4] `https://support.apple.com/kb/sp719?locale=en_US`

inicialização. Isso pode incluir o envio de variáveis de ambiente, que adaptam certas funcionalidades da aplicação com base na sua implementação, ou argumentos, que serão usados durante os testes para modificar as configurações da aplicação ou para permitir a utilização de dados de teste. No contexto dos casos de teste que estamos analisando aqui, são fornecidas uma chave de autenticação e uma URL que permitem o uso de dados de teste durante a execução, em vez de dados de produção. Na Figura 4.1 é possível ver como esse setup foi feito no código original e na Figura 4.2 como foi feito utilizando o *BIUTest*.

Toda a criação e configuração do *XCUIApplication*, que representa a aplicação sendo testada, é feita automaticamente pelo *BIUTest*, dessa maneira a pessoa testadora precisa somente criar um arquivo JSON com os argumentos e variáveis de ambiente a serem utilizados.

```
23      override func setUp() {
24          // In UI tests it is usually best to stop immediately when a failure occurs.
25          continueAfterFailure = false
26
27          // UI tests must launch the application that they test. Doing this in setup will make
28          // sure it happens for each test method.
29          app = XCUIApplication()
30          let stripePublishableKey = "pk_test_6Q7qTzl8OkUj5K5ArgayVsFD00Sa5AHMj3"
31          let backendBaseURL = "https://stp-mobile-legacy-test-backend-17.stripedemos.com/"
32          app.launchArguments.append(contentsOf: [
33              "-StripePublishableKey", stripePublishableKey, "-StripeBackendBaseURL", backendBaseURL,
34          ])
35          app.launchEnvironment = ["UITesting": "true"]
36          app.launch()
37      }
```

Figura 4.1 Setup Original. Fonte: Repositório da Aplicação de Teste

```
1   {
2       "launch_arguments": [
3           "-StripePublishableKey", "pk_test_6Q7qTzl8OkUj5K5ArgayVsFD00Sa5AHMj3",
4           "-StripeBackendBaseURL", "https://stp-mobile-legacy-test-backend-17.stripedemos.com/"
5       ],
6       "launch_environment": {
7           "UITesting": "true"
8       }
9   }
```

Figura 4.2 Setup com BIUTest. Fonte: Elaborado pela autora

4.2.2 Helpers

Os casos de teste começam desabilitando a necessidade de inserir o endereço, selecionando três produtos e avançando para a tela de checkout. Para simplificar a execução dessas ações, algumas funções foram implementadas, uma vez que elas se repetem em todos os casos de teste. A Figura 4.3 ilustra como esse código foi implementado na versão original, enquanto a Figura 4.4 mostra a implementação utilizando o *BIUTest*. O fluxo de telas nessa etapa pode ser visualizado na Figura A.1.

O *BIUTest* realiza a busca dos elementos que correspondem ao identificador enviado como parâmetro, o que dispensa a pessoa responsável pelos testes de especificar exatamente onde o elemento está localizado. Neste exemplo, não é necessário indicar que o botão contendo um sapato vermelho está dentro de uma célula. Além disso, algumas funções para auxiliar na sincronização com os elementos são implementadas por padrão no *BIUTest*. Por exemplo, no código original foi necessário utilizar um **waitToAppear** para aguardar o botão estar aparecendo antes de interagir com a tela, no *BIUTest* a função **tap_button** já realiza essa espera antes de tentar interagir com o elemento.

```
39      func disableAddressEntry(_ app: XCUIApplication) {
40          app.navigationBars["Emoji Apparel"].buttons["Settings"].tapWhenHittableInTestCase(self)
41          let noneButton = app.tables.children(matching: .cell).element(boundBy: 12).staticTexts["None"]
42          waitToAppear(noneButton)
43          app.tables.firstMatch.swipeUp()
44          noneButton.tapWhenHittableInTestCase(self)
45          app.navigationBars["Settings"].buttons["Done"].tapWhenHittableInTestCase(self)
46      }
47
48      func selectItems(_ app: XCUIApplication) {
49          let cellsQuery = app.collectionViews.cells
50          cellsQuery.otherElements.containing(.staticText, identifier: "👠").element.tapWhenHittableInTestCase(self)
51          app.collectionViews.staticTexts["👠"].tapWhenHittableInTestCase(self)
52          cellsQuery.otherElements.containing(.staticText, identifier: "🧪").children(matching: .other)
53              .element(boundBy: 0).tapWhenHittableInTestCase(self)
54      }
```

Figura 4.3 Helpers Original. Fonte: Repositório da Aplicação de Teste

```
helpers = {}

helpers.disable_address_entry = function()
    biu.tap_button("Settings")
    biu.scroll_up()
    biu.tap_button("#2|None")
    biu.tap_button("Done")
end

helpers.select_items = function(items)
    for i, item in ipairs(items) do
        biu.tap_button(item)
    end
end

helpers.select_payment_method = function(method)
    biu.tap_button("Buy Now")
    biu.tap_button("Pay from")
    biu.wait_for(method)
    biu.tap_button(method)
end
```

Figura 4.4 Helpers com BIUTest. Fonte: Elaborado pela autora

4.2.3 Caso de Teste 1

Nas Figuras A.2 e A.3, podemos observar a implementação deste caso de teste tanto sem o uso do framework proposto neste trabalho quanto com ele. Neste caso, notamos que o teste original verifica apenas se as interações indicadas, como pressionar os botões, são possíveis de serem realizadas. No entanto, ao utilizar o framework *BIUTest*, não apenas garantimos a viabilidade das interações, mas também asseguramos que a tela vista pelo usuário está correta, uma vez que o framework realiza uma comparação com a referência previamente armazenada. Na Figura A.4 é possível ver o fluxo de telas desse caso de teste.

4.2.4 Caso de Teste 2

Na Figura 4.5 podemos observar que algumas instruções no código são bem específicas da implementação e pode não ser facilmente entendidas por uma pessoa sem conhecimento técnico de aplicações iOS e interno da aplicação, por exemplo o uso de *scrollViews* e *alerts* na busca por elementos. A Figura 4.6 realiza as mesmas interações com a aplicação sendo testadas com uma linguagem mais clara. A Figura A.7 demonstra o fluxo de telas realizado nesse caso de teste.

```
let elementsQuery = app.scrollViews.otherElements
let learnMore = elementsQuery.buttons["Learn more about authentication"]
learnMore.tapWhenHittableInTestCase(self)
elementsQuery.buttons["Need help?"].tapWhenHittableInTestCase(self)
app.scrollViews.otherElements.buttons["Continue"].tapWhenHittableInTestCase(self)
let success = app.alerts["Success"].buttons["OK"]

success.tapWhenHittableInTestCase(self)
```

Figura 4.5 Código original com busca de elementos detalhada. Fonte: Repositório da Aplicação de Teste

```
 8   biu.tap_button("Learn more about authentication")
 9   biu.tap_button("Need help?")
10   biu.compare_ref()
11   biu.tap_button("Continue")
12   biu.wait_for("Buy")
13   biu.tap_button("OK")
```

Figura 4.6 Código usando BIUTest com busca de elementos abstraindo implementação interna. Fonte: Elaborado pela autora

4.2.5 Caso de Teste 3

Esse caso de teste já era pequeno e relativamente simples de entender. A melhoria no código utilizando o *BIUTest* e nesse caso está mais relacionada à adição de uma função *helper* para a seleção de forma de pagamento, e poderia ser replicada sem muito esforço no código original. Na Figura A.10 é possível ver o fluxo de telas desse caso de teste.

4.2.6 Caso de Teste 4

Este caso de teste exemplifica mais uma interação, que é a de inserir texto em um campo de texto. Uma vantagem presente no código original, representado na Figura 4.7, é que os elementos podem ser armazenados em variáveis, evitando assim buscas repetidas quando eles precisam ser usados novamente. O fluxo de telas desse caso de teste pode ser visualizado na Figura A.13.

```
128         let cardNumberField = tablesQuery.textFields["card number"]
129         let cvcField = tablesQuery.textFields["CVC"]
130         let zipField = tablesQuery.textFields["Postal code"]
131         cardNumberField.tapWhenHittableInTestCase(self)
132         cardNumberField.typeText("4000000000000069")
133         let expirationDateField = tablesQuery.textFields["expiration date"]
134         expirationDateField.typeText("02/28")
135         cvcField.typeText("223")
136         zipField.typeText("90210")
```

Figura 4.7 Código original armazenando elementos em variáveis. Fonte: Repositório da Aplicação de Teste

4.2.7 Caso de Teste 5

Nesse caso, podemos observar que, apesar da vantagem de poder salvar os elementos em variáveis, nem sempre isso é possível de ser utilizado. Isso é evidenciado no código original, conforme representado na Figura 4.8, onde alguns elementos, como o botão "Products", precisam ser buscados várias vezes devido às transições de tela que ocorrem cada vez que ele é pressionado. O fluxo de telas deste caso de teste é ilustrado na Figura A.17.

```
189         app.navigationBars["Checkout"].buttons["Products"].tapWhenHittableInTestCase(self)
190         buyNowButton.tapWhenHittableInTestCase(self)
191         payFromButton.tapWhenHittableInTestCase(self)
192
193         // ...should keep Apple Pay selected
194         XCTAssertTrue(applePay.isSelected)
195         XCTAssertFalse(visa.isSelected)
196
197         // Selecting another payment method...
198         visa.tapWhenHittableInTestCase(self)
199
200         // ...and logging out...
201         app.navigationBars["Checkout"].buttons["Products"].tapWhenHittableInTestCase(self)
```

Figura 4.8 Código original repetindo busca por elementos. Fonte: Repositório da Aplicação de Teste

4.3 Resultados

Nesta seção, serão apresentados os resultados derivados das análises realizadas nos casos de teste selecionados, considerando os critérios de avaliação previamente definidos. Ao término, realizaremos uma breve discussão dos resultados, destacando os principais pontos observados durante as análises.

4.3.1 Critérios de Avaliação

4.3.1.1 API

Lines of Code (LOC) é uma métrica utilizada para medir o tamanho de um software ao quantificar as linhas de código do programa. Além disso, ela funciona como uma abordagem de estimativa para mensurar o esforço envolvido no processo de desenvolvimento de software [10]. A Tabela 4.6 mostra uma comparação na quantidade de linhas necessárias para escrever cada caso de teste. Não foram levadas em consideração as linhas utilizadas para o setup de cada caso e as partes que são repetidas, contidas nas funções *helpers*. Para essa análise foi utilizada a ferramenta *cloc*[5], que realiza a contagem de linhas de código de diversas linguagens diferentes. É possível notar que utilizando o *BIUTest* foram necessárias menos linhas de código para a escrita dos casos de teste. Nos casos realizados nessa análise, o uso do framework significou uma diminuição de, em média, 50,4% na quantidade de linhas de código.

LOC		
	XCUITest	**BIUTest**
Caso de Teste 1	13	6
Caso de Teste 2	18	9
Caso de Teste 3	12	5
Caso de Teste 4	33	17
Caso de Teste 5	35	22

Tabela 4.6 Lines of Code (LOC) - Casos de Teste

Ao compararmos os códigos dos casos de teste utilizando o *XCUITest* e o *BIUTest*, também se pode notar que, sem o uso do framework, é preciso ter um conhecimento maior sobre a implementação da aplicação. Por exemplo, no Caso de Teste 2 é possível ver na Figura A.5 que para apertar o botão "OK"foi especificado que o mesmo está presente em um alerta, já na Figura A.6 foi usada somente a instrução de apertar um botão. Essa abstração traz algumas desvantagens, primeiro em relação ao tempo de execução, que será comentado em um tópico a seguir, e segundo em relação a identificação de elementos com o mesmo nome. Para esses casos, foi implementada uma notação que pode ser utilizada para identificar qual dos elementos deve ser escolhido para realizar a interação, é possível ver essa notação no script dos helpers dos casos de uso analisados, na linha 6 da Figura 4.4. Essa abordagem pode acrescentar um

[5] `https://github.com/AlDanial/cloc`

esforço a mais na manutenção dos casos de teste, já que se algum dos botões mudar de posição o número pode precisar ser atualizado.

4.3.1.2 Suporte a Logs

Como o *BIUTest* utiliza o *XCUITest*, acaba se beneficiando do seu suporte a logs que é bastante detalhado, fornecendo um relatório de execução em tempo real durante a execução dos testes, característica que o destaca em relação à outras ferramentas de testes de UI, como mostrado em [4]. Na Figura 4.9 é possível ver um exemplo desse relatório.

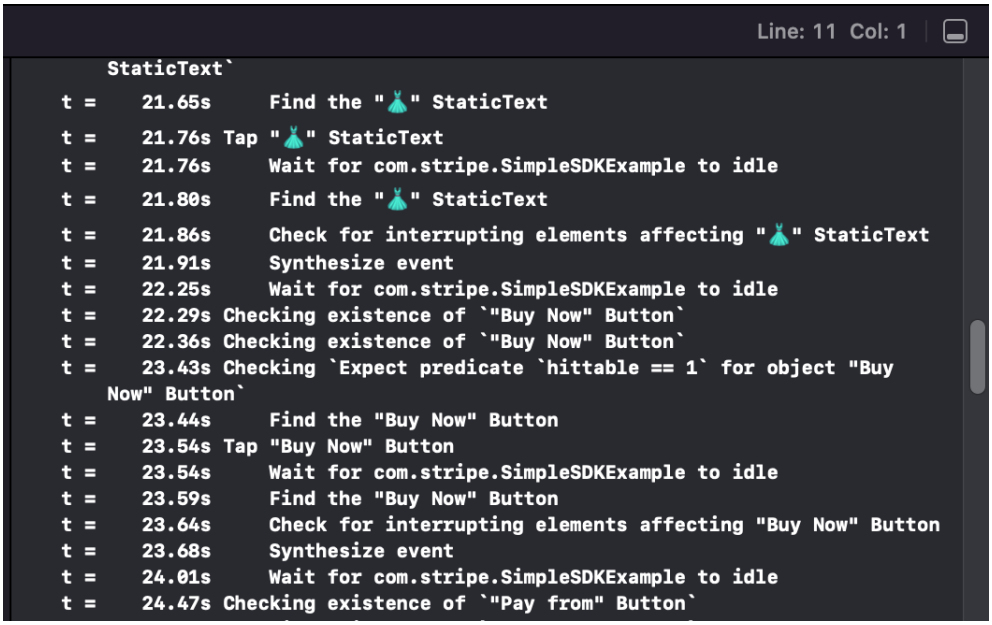

Figura 4.9 Informações de depuração no console do Xcode durante a execução dos testes. Fonte: Elaborado pela autora

Adicionalmente, o *BIUTest* também herda do *XCUITest* os logs de erro no caso de um teste falhar. Esses logs auxiliam a pessoa testadora a identificar onde o erro ocorreu e como corrigi-lo. A Figura 4.10 exemplifica um caso em que o botão "Close"não pôde ser localizado. Como o *BIUTest* possui algumas funcionalidades adicionais, eventuais erros específicos podem ocorrer. No entanto, a informação sobre esses erros é apresentada de maneira semelhante, como evidenciado na Figura 4.11. Quando a tela não corresponde à imagem de referência previamente salva, é fornecido ao usuário a imagem de referência, a captura da tela e a diferença entre elas.

4.3.1.3 Suporte a uma grande variedade de propriedades

Como foi mencionado na Seção 4.2.1, através do envio de argumentos e variáveis de ambiente é possível que a aplicação seja executada utilizando diferentes configurações, como linguagem por exemplo. Com o *BIUTest* isso é feito por meio do arquivo JSON presente em cada caso

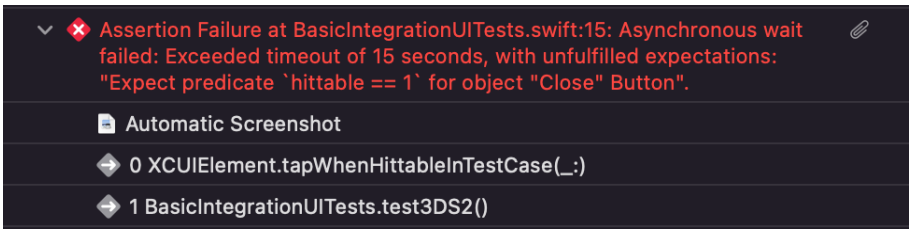

Figura 4.10 Informações de erro na falha do caso de teste no XCUITest. Fonte: Elaborado pela autora

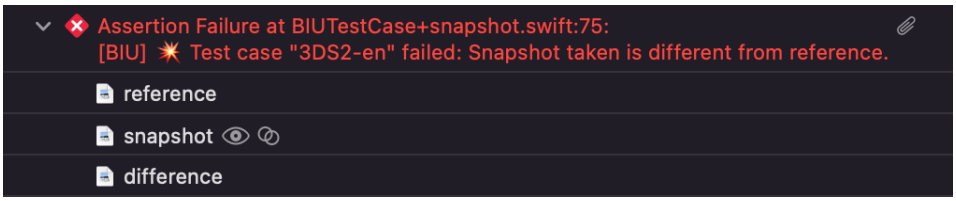

Figura 4.11 Informações de erro na falha do caso de teste no BIUTest. Fonte: Elaborado pela autora

de teste, no *XCUITest* essa passagem de argumentos e variáveis também é possível através das variáveis disponíveis na classe de comunicação com a aplicação, como foi mostrado nas Figuras 4.1 e 4.2. Além disso, no *BIUTest* é possível ter referências salvas para diversos modelos de dispositivo e sistemas operacionais, permitindo que os testes sejam executados em todos eles e garantindo a conformidade visual nos mesmos, já o *XCUITest* não tem essa funcionalidade.

4.3.1.4 Tempo de Execução

Foi feita uma comparação do tempo de execução dos casos de teste com e sem a utilização do framework *BIUTest*. A análise foi feita executando cada caso de teste cinco vezes, e então foi observado o menor tempo, o maior e o tempo médio de cada um. Na Tabela 4.7 pode-se observar os resultados encontrados.

A partir dos dados coletados durante as execuções, é perceptível que a utilização do framework acrescentou uma quantidade considerável de tempo, uma média de 3,8 segundos, em cada caso de teste. No total, houve um aumento de 6,7% no tempo de execução quando todos os casos de teste foram combinados. Esse aumento no tempo era esperado, uma vez que o nível de abstração do *BIUTest* é mais elevado, o que significa que o framework precisa buscar o elemento necessário para realizar a interação em mais partes da interface. Além disso, a capacidade de armazenar o elemento em uma variável para ser reutilizado posteriormente também contribui para reduzir o tempo, uma vez que não será necessário procurá-lo novamente quando ocorrer outra interação. Essa é uma opção de desenvolvimento que pode ser feita como melhoria para o framework.

Tempo de Execução (s)					
		Menor	**Maior**	**Média**	**% Média**
Caso de Teste 1	**XCUITest**	37,2	39,6	38,2	
	BIUTest	40,7	41,7	41,1	+7,6%
Caso de Teste 2	**XCUITest**	44,9	45,5	45,2	
	BIUTest	48,9	52,7	50,5	+11,7%
Caso de Teste 3	**XCUITest**	32,6	33,7	33,1	
	BIUTest	34,1	36,8	34,9	+18,1%
Caso de Teste 4	**XCUITest**	50,9	52,5	51,6	
	BIUTest	56,8	59,0	57,6	+11,6%
Caso de Teste 5	**XCUITest**	64,6	66,5	65,4	
	BIUTest	67,3	69,3	68,2	+4,3%

Tabela 4.7 Tempo de Execução - Casos de Teste

4.3.2 Considerações Finais

Através da análise dos resultados com base nos critérios de avaliação selecionados, é evidente que o uso do *BIUTest* apresentou vantagens e também desvantagens em diferentes aspectos. O framework oferece uma API mais simples e legível, exigindo menos conhecimento técnico e um entendimento específico da implementação da aplicação para utilizá-lo. Essa característica também facilita a atualização e manutenção dos testes de forma menos trabalhosa. No entanto, isso resulta em um tempo de execução dos casos de teste mais longo, que por si só já são extensos. Além disso, em casos onde exista mais de um elemento com o mesmo nome na tela sendo testada, é necessário uma notação específica para identificar qual deles deve ser utilizado para a interação, adicionando um esforço a mais na manutenção do caso de teste em questão.

A escolha entre a utilização de um framework ou outro pode depender de vários fatores, como o tamanho do projeto, da empresa, o número de pessoas envolvidas no desenvolvimento e o nível de conhecimento técnico dessas pessoas, entre outros. Considerando diversos pontos, é possível que uma ferramenta seja mais adequada para um contexto do que a outra, e, principalmente, o uso de uma não exclui a possibilidade de usar a outra. Portanto, é viável escolher casos específicos nos quais seja mais apropriado utilizar uma ou outra ferramenta.

CAPÍTULO 5

Conclusão e Trabalhos Futuros

Durante o desenvolvimento de uma aplicação, é crucial assegurar que o estado de sua interface nos fluxos existentes seja determinístico. Isso significa que ao seguir uma mesma sequência de interações com as mesmas entradas, o estado final da aplicação deve ser o mesmo. Empresas de software aplicam diversos tipos de testes com esse propósito, como testes de interface de usuário (UI), de regressão e de integração, que podem ser realizados manualmente ou por meio de ferramentas de automatização. Os testes automatizados oferecem vantagens como custos mais baixos, maior frequência de testes, identificação precoce de defeitos e maior qualidade do sistema em comparação com testes manuais.

Ao longo dos anos, várias ferramentas foram desenvolvidas para auxiliar na execução de testes automatizados, e diferentes técnicas podem ser empregadas, dependendo do contexto da aplicação, do tamanho da equipe, dos recursos disponíveis e dos objetivos de validação. Os testes automatizados de UI são essenciais para verificar o correto funcionamento dos componentes da interface e como eles são apresentados ao usuário final. Neste trabalho, foi proposto o framework *BIUTest* com o objetivo de abordar ambas as áreas de validação em aplicações iOS.

Foi realizada uma análise do uso desse framework por meio de uma comparação com uma aplicação que já possui testes de UI implementados, replicando os mesmos casos de teste utilizando o *BIUTest*. A análise revelou vantagens do framework, incluindo uma API de fácil leitura e entendimento, que requer menos conhecimento interno da aplicação. Além disso, o *BIUTest* oferece a capacidade de comparar telas durante a execução dos testes com imagens de referência, garantindo que a aparência seja consistente em diferentes dispositivos e versões do sistema operacional. Levando em conta que fluxos importantes da aplicação sejam testados utilizando o framework, é possível ajudar a assegurar que continuam com seu funcionamento e visual corretos após mudanças realizadas na aplicação. No entanto, o uso do framework também apresentou desvantagens, como um aumento no tempo de execução dos casos de teste.

Em relação a trabalhos futuros, durante as análises realizadas foram identificadas quatro possibilidades de expansão do framework proposto. A seguir são listadas cada uma delas.

- **Melhoria da Busca por Elementos:** A busca por elementos para realizar cada interação é um dos pontos que pode ser melhorado. De acordo com o caso de teste sendo executado, permitir a reutilização de elementos em interações repetidas, evitando buscas desnecessárias, pode resultar em um menor tempo na execução dos testes.

- **Melhoria de Sincronização:** Pode ser implementada também uma melhoria de sincronização durante a execução dos testes, evitando que interações ocorram durante animações

ou transições de tela, eliminando a necessidade de a pessoa testadora definir tempos de espera.

- **Suporte à Gravação:** Outra oportunidade consiste na adição de suporte à gravação de testes, permitindo que os scripts sejam gerados durante a execução da aplicação e utilizados em execuções futuras dos casos de teste.
- **Implementação em Android:** Considerando que muitas aplicações iOS possuem contrapartes em dispositivos Android, uma oportunidade de aprimoramento seria a implementação do *BIUTest* para aplicações nessa plataforma. Isso permitiria que a mesma pasta contendo o script e as referências a serem comparadas fosse utilizada para testar aplicativos em ambas as plataformas.

APÊNDICE A

Casos de Teste

A.1 Helpers

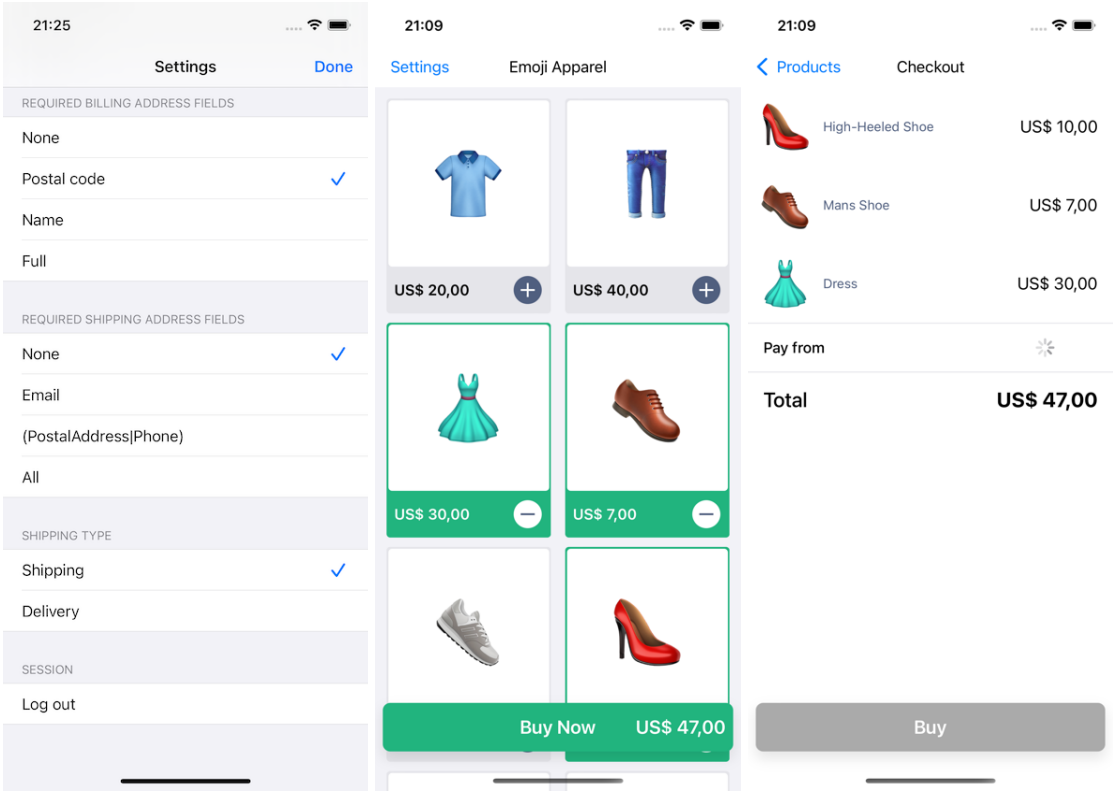

Figura A.1 Fluxo Inicial dos Casos de Teste. Fonte: Elaborado pela autora

A.2 Caso de Teste 1

```
66      func testSimpleTransaction() {
67          disableAddressEntry(app)
68          selectItems(app)
69
70          app.buttons["Buy Now"].tapWhenHittableInTestCase(self)
71          let payFromButton = app.buttons.matching(identifier: "Pay from").element
72          waitToAppear(payFromButton)
73          payFromButton.tapWhenHittableInTestCase(self)
74          let visa = app.tables.staticTexts["Visa ending in 4242"]
75          visa.tapWhenHittableInTestCase(self)
76          app.buttons["Buy"].tapWhenHittableInTestCase(self)
77          let success = app.alerts["Success"].buttons["OK"]
78          success.tapWhenHittableInTestCase(self)
      }
```

Figura A.2 Caso de Teste 1 Original. Fonte: Repositório da Aplicação de Teste

```
1  helpers.disable_address_entry()
2  helpers.select_items { "👠", "👞", "👗" }
3  helpers.select_payment_method("Visa ending in 4242")
4
5  biu.compare_ref()
6  biu.tap_button("Buy")
7  biu.wait_for("Success")
8  biu.tap_button("OK")
```

Figura A.3 Caso de Teste 1 com BIUTest. Fonte: Elaborado pela autora

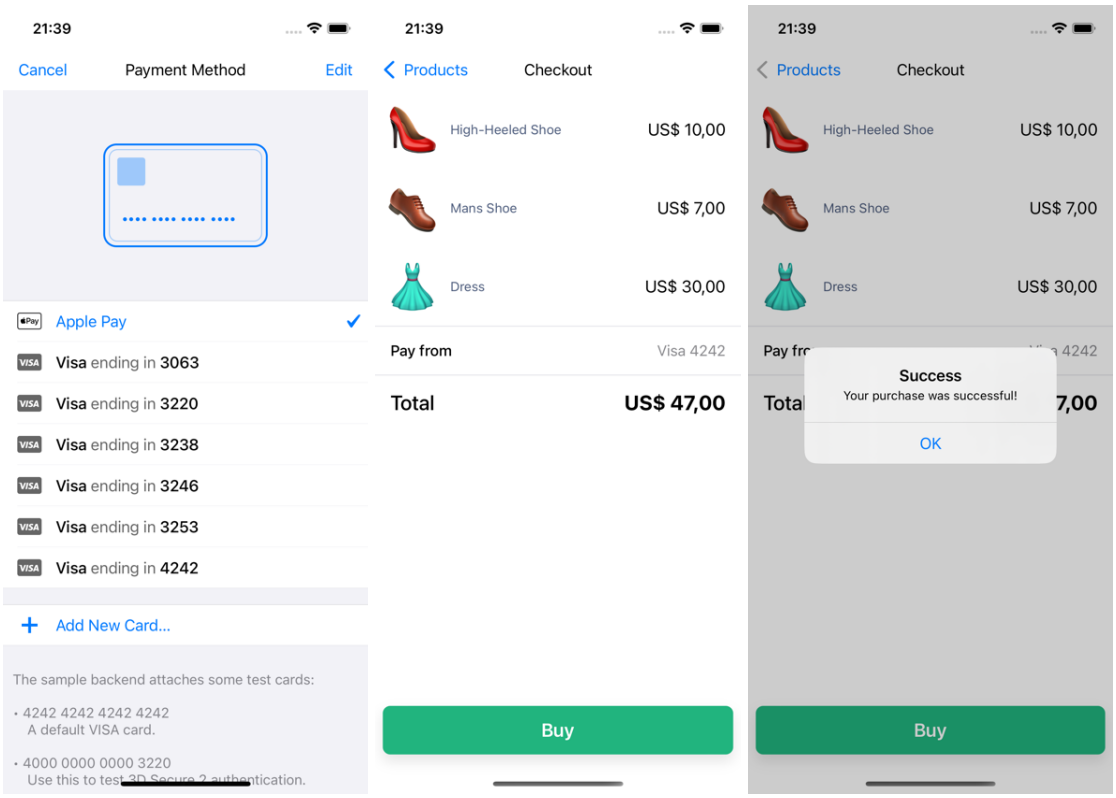

Figura A.4 Fluxo Caso de Teste 1. Fonte: Elaborado pela autora

A.3 Caso de Teste 2

```
    func test3DS2() {
        disableAddressEntry(app)
        selectItems(app)

        let buyNowButton = app.buttons["Buy Now"]
        buyNowButton.tapWhenHittableInTestCase(self)
        let payFromButton = app.buttons.matching(identifier: "Pay from").element
        payFromButton.tapWhenHittableInTestCase(self)
        let visa = app.tables.staticTexts["Visa ending in 3220"]
        visa.tapWhenHittableInTestCase(self)
        app.buttons["Buy"].tapWhenHittableInTestCase(self)

        let elementsQuery = app.scrollViews.otherElements
        let learnMore = elementsQuery.buttons["Learn more about authentication"]
        learnMore.tapWhenHittableInTestCase(self)
        elementsQuery.buttons["Need help?"].tapWhenHittableInTestCase(self)
        app.scrollViews.otherElements.buttons["Continue"].tapWhenHittableInTestCase(self)
        let success = app.alerts["Success"].buttons["OK"]

        success.tapWhenHittableInTestCase(self)
    }
```

Figura A.5 Caso de Teste 2 Original. Fonte: Repositório da Aplicação de Teste

```
helpers.disable_address_entry()
helpers.select_items { "👠", "👞", "👗" }
helpers.select_payment_method("Visa ending in 3220")

biu.tap_button("Buy")
biu.wait_for("Complete Authentication")
biu.compare_ref()
biu.tap_button("Learn more about authentication")
biu.tap_button("Need help?")
biu.compare_ref()
biu.tap_button("Continue")
biu.wait_for("Buy")
biu.tap_button("OK")
```

Figura A.6 Caso de Teste 2 com BIUTest. Fonte: Elaborado pela autora

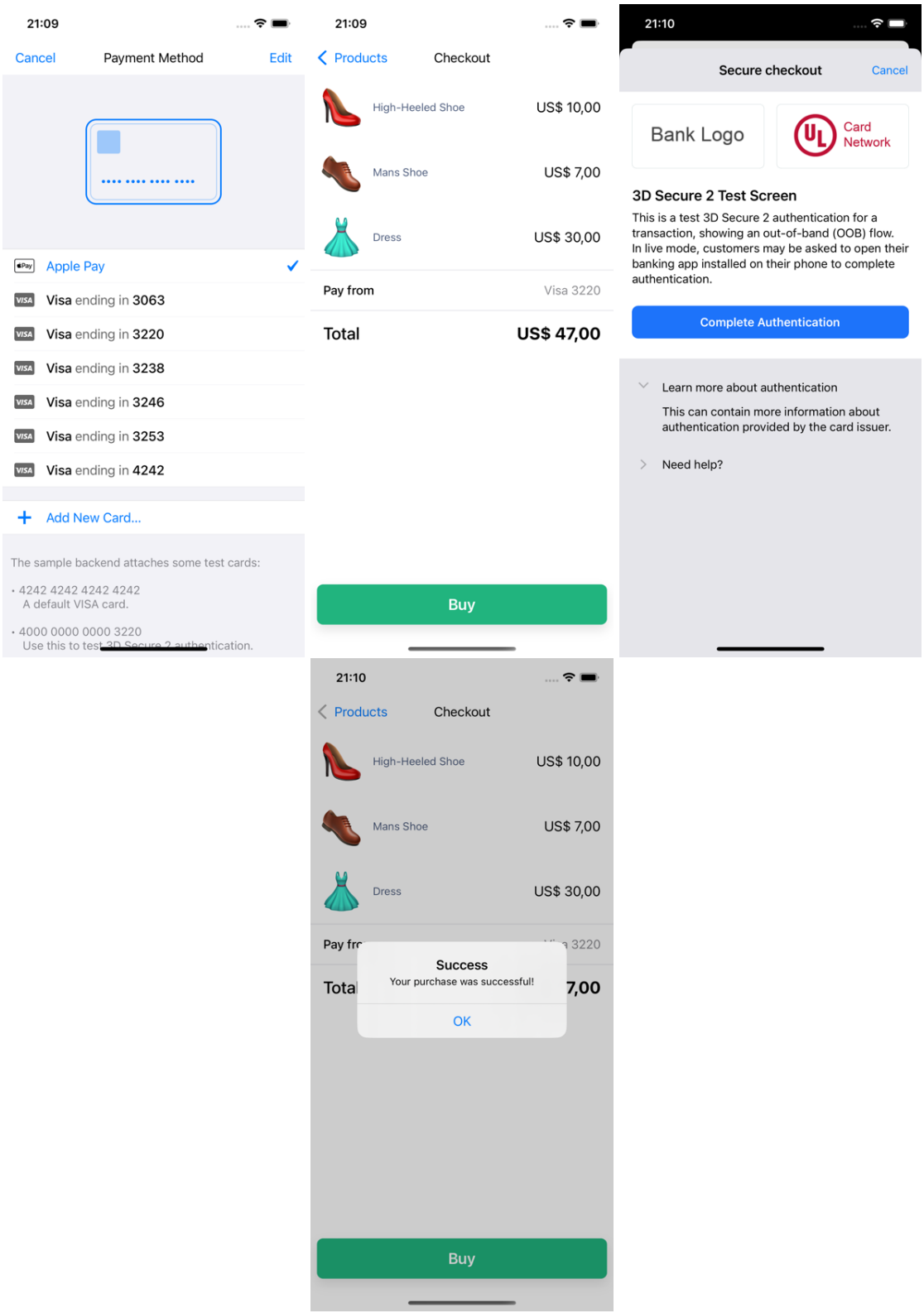

Figura A.7 Fluxo Caso de Teste 2. Fonte: Elaborado pela autora

A.4 Caso de Teste 3

```
103     func testPopApplePaySheet() {
104         disableAddressEntry(app)
105         selectItems(app)
106
107         let buyNowButton = app.buttons["Buy Now"]
108         buyNowButton.tapWhenHittableInTestCase(self)
109         let payFromButton = app.buttons.matching(identifier: "Pay from").element
110         payFromButton.tapWhenHittableInTestCase(self)
111         let tablesQuery = app.tables
112         let applePay = tablesQuery.staticTexts["Apple Pay"]
113         applePay.tapWhenHittableInTestCase(self)
114         app.buttons["Buy"].tapWhenHittableInTestCase(self)
    }
```

Figura A.8 Caso de Teste 3 Original. Fonte: Repositório da Aplicação de Teste

```
1   helpers.disable_address_entry()
2   helpers.select_items { "👠", "👞", "👗" }
3   helpers.select_payment_method("Apple Pay")
4
5   biu.tap_button("Buy")
6   biu.wait_seconds("1")
7   biu.compare_ref()
```

Figura A.9 Caso de Teste 3 com BIUTest. Fonte: Elaborado pela autora

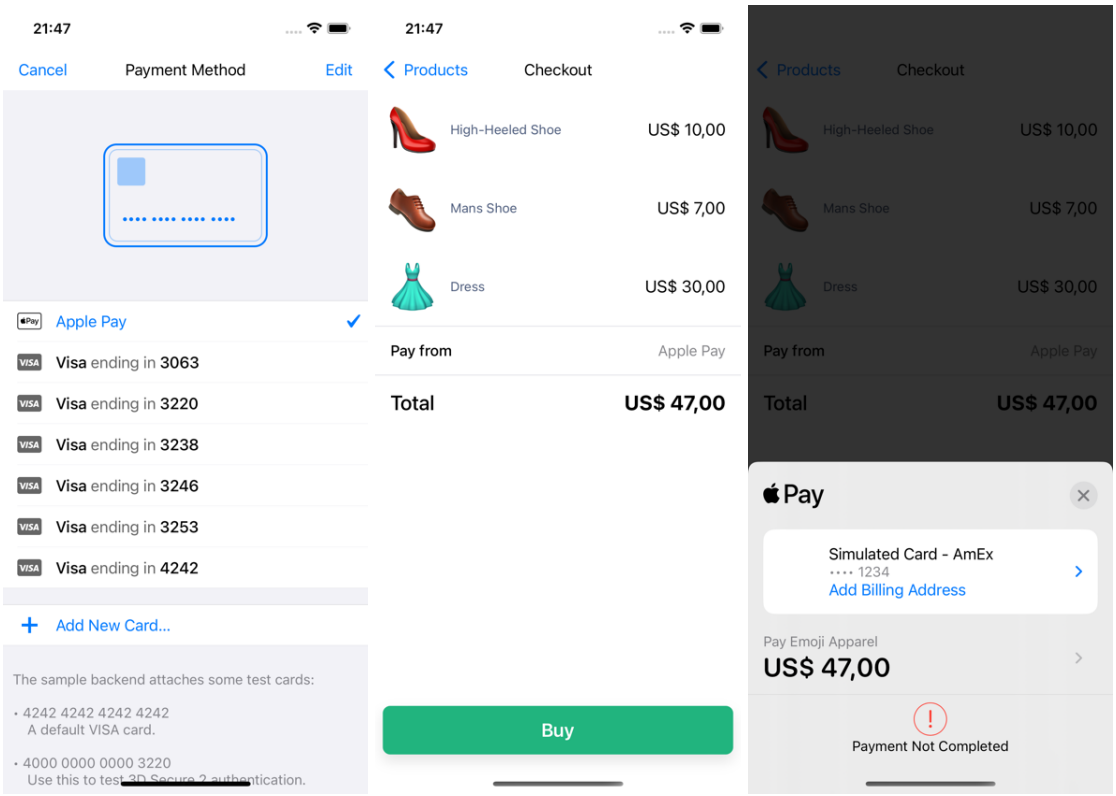

Figura A.10 Fluxo Caso de Teste 3. Fonte: Elaborado pela autora

A.5 Caso de Teste 4

```
117     func testCCEntry() {
118         disableAddressEntry(app)
119         selectItems(app)
120 
121         let buyNowButton = app.buttons["Buy Now"]
122         buyNowButton.tapWhenHittableInTestCase(self)
123         let payFromButton = app.buttons.matching(identifier: "Pay from").element
124         payFromButton.tapWhenHittableInTestCase(self)
125         let tablesQuery = app.tables
126         let addButton = app.tables.staticTexts["Add New Card…"]
127         addButton.tapWhenHittableInTestCase(self)
128 
129         let cardNumberField = tablesQuery.textFields["card number"]
130         let cvcField = tablesQuery.textFields["CVC"]
131         let zipField = tablesQuery.textFields["Postal code"]
132         cardNumberField.tapWhenHittableInTestCase(self)
133         cardNumberField.typeText("4000000000000069")
134         let expirationDateField = tablesQuery.textFields["expiration date"]
135         expirationDateField.typeText("02/28")
136         cvcField.typeText("223")
137         zipField.typeText("90210")
138 
139         let addcardviewcontrollernavbardonebuttonidentifierButton = app.navigationBars["Add a Card"]
140             .buttons["AddCardViewControllerNavBarDoneButtonIdentifier"]
141         addcardviewcontrollernavbardonebuttonidentifierButton.tapWhenHittableInTestCase(self)
142         app.alerts["Your card has expired."].buttons["OK"].tapWhenHittableInTestCase(self)
143         cardNumberField.tapWhenHittableInTestCase(self)
144         let deleteString = String(repeating: XCUIKeyboardKey.delete.rawValue, count: 4)
145         cardNumberField.typeText(deleteString)
146         cardNumberField.typeText("0341")
147         addcardviewcontrollernavbardonebuttonidentifierButton.tapWhenHittableInTestCase(self)
148         let buyButton = app.buttons["Buy"]
149         buyButton.tapWhenHittableInTestCase(self)
150         let errorButton = app.alerts["Error"].buttons["OK"]
151         errorButton.tapWhenHittableInTestCase(self)
    }
```

Figura A.11 Caso de Teste 4 Original. Fonte: Repositório da Aplicação de Teste

```
helpers.disable_address_entry()
helpers.select_items { "👠", "👞", "👗" }
helpers.select_payment_method("Add New Card…")

biu.enter_text { text = "4000000000000069", field = "card number" }
biu.compare_ref()
biu.enter_text { text = "02/28", field = "expiration date" }
biu.compare_ref()
biu.enter_text { text = "223", field = "CVC" }
biu.enter_text { text = "90210", field = "Postal code" }
biu.tap_button("Done")

biu.wait_for("Your card has expired.")
biu.compare_ref()
biu.tap_button("OK")

biu.delete_text { letter_count = 4, field = "card number" }
biu.enter_text { text = "0341", field = "card number" }
biu.tap_button("Done")

biu.wait_for("Buy")
biu.tap_button("Buy")
biu.wait_for("Error")
biu.tap_button("OK")
```

Figura A.12 Caso de Teste 4 com BIUTest. Fonte: Elaborado pela autora

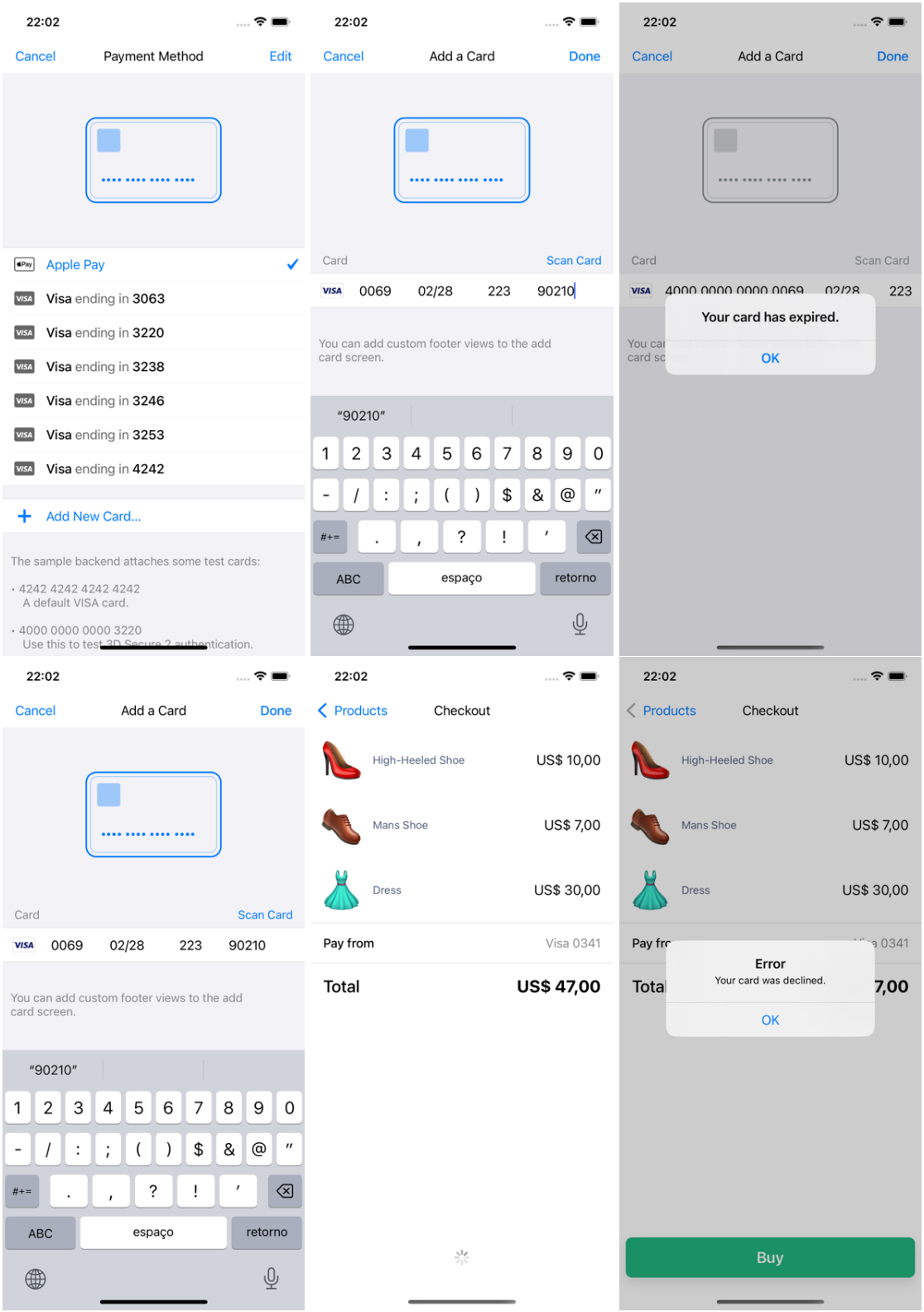

Figura A.13 Fluxo Caso de Teste 4. Fonte: Elaborado pela autora

A.6 Caso de Teste 5

```
    func testPaymentOptionsDefault() {
        // Note that the example backend creates a new Customer every time you start the app
        // A STPPaymentOptionsVC w/o a selected card...
        disableAddressEntry(app)
        selectItems(app)
        let buyNowButton = app.buttons["Buy Now"]
        buyNowButton.tapWhenHittableInTestCase(self)
        let payFromButton = app.buttons.matching(identifier: "Pay from").element
        payFromButton.tapWhenHittableInTestCase(self)

        let tablesQuery = app.tables

        // ...preselects Apple Pay by default
        let applePay = tablesQuery.cells["Apple Pay"]
        waitToAppear(applePay)
        XCTAssertTrue(applePay.isSelected)

        // Selecting another payment method...
        let visa = tablesQuery.cells["Visa ending in 3220"]
        visa.tapWhenHittableInTestCase(self)

        // ...and resetting the PaymentOptions VC...
        // Note that STPPaymentContext clears its cache and refetches every time it's initialized,
        // which happens whenever CheckoutViewController is pushed on
        app.navigationBars["Checkout"].buttons["Products"].tapWhenHittableInTestCase(self)
        buyNowButton.tapWhenHittableInTestCase(self)
        payFromButton.tapWhenHittableInTestCase(self)

        // ...should keep the 3220 card selected
        XCTAssertTrue(visa.isSelected)
        XCTAssertFalse(applePay.isSelected)

        // Reselecting Apple Pay...
        applePay.tapWhenHittableInTestCase(self)
```

Figura A.14 Caso de Teste 5 Original - Parte 1. Fonte: Repositório da Aplicação de Teste

```
187
188         // ...and resetting the PaymentOptions VC...
189         app.navigationBars["Checkout"].buttons["Products"].tapWhenHittableInTestCase(self)
190         buyNowButton.tapWhenHittableInTestCase(self)
191         payFromButton.tapWhenHittableInTestCase(self)
192
193         // ...should keep Apple Pay selected
194         XCTAssertTrue(applePay.isSelected)
195         XCTAssertFalse(visa.isSelected)
196
197         // Selecting another payment method...
198         visa.tapWhenHittableInTestCase(self)
199
200         // ...and logging out...
201         app.navigationBars["Checkout"].buttons["Products"].tapWhenHittableInTestCase(self)
202         app.navigationBars["Emoji Apparel"].buttons["Settings"].tapWhenHittableInTestCase(self)
203         app.tables.children(matching: .cell).element(boundBy: 18).staticTexts["Log out"].tapWhenHittableInTestCase(self)
204         app.navigationBars["Settings"].buttons["Done"].tapWhenHittableInTestCase(self)
205
206         // ...and going back to PaymentOptionsVC...
207         buyNowButton.tapWhenHittableInTestCase(self)
208         payFromButton.tapWhenHittableInTestCase(self)
209
210         // ..should not retain the visa default
211         waitToAppear(applePay)
212         XCTAssertTrue(applePay.isSelected)
213         XCTAssertFalse(visa.isSelected)
214     }
215 }
```

Figura A.15 Caso de Teste 5 Original - Parte 2. Fonte: Repositório da Aplicação de Teste

```
helpers.disable_address_entry()
helpers.select_items { "👠", "👞", "👗" }

biu.tap_button("Buy Now")
biu.tap_button("Pay from")
biu.wait_for("Apple Pay")
biu.compare_ref() -- check if apple play is preselected

biu.tap_button("Visa ending in 3220")

biu.tap_button("Products")
biu.tap_button("Buy Now")
biu.tap_button("Pay from")
biu.compare_ref() -- check if 3220 card is still selected

biu.tap_button("Apple Pay")
biu.tap_button("Products")
biu.tap_button("Buy Now")
biu.tap_button("Pay from")
biu.compare_ref() -- check if Apple Pay is still selected

biu.tap_button("Visa ending in 3220")

biu.tap_button("Products")
biu.tap_button("Settings")
biu.scroll_up()
biu.tap_button("Log out")
biu.tap_button("Done")

biu.tap_button("Buy Now")
biu.tap_button("Pay from")
biu.wait_for("Apple Pay")
biu.compare_ref() -- check if Apple Pay is selected
```

Figura A.16 Caso de Teste 5 com BIUTest. Fonte: Elaborado pela autora

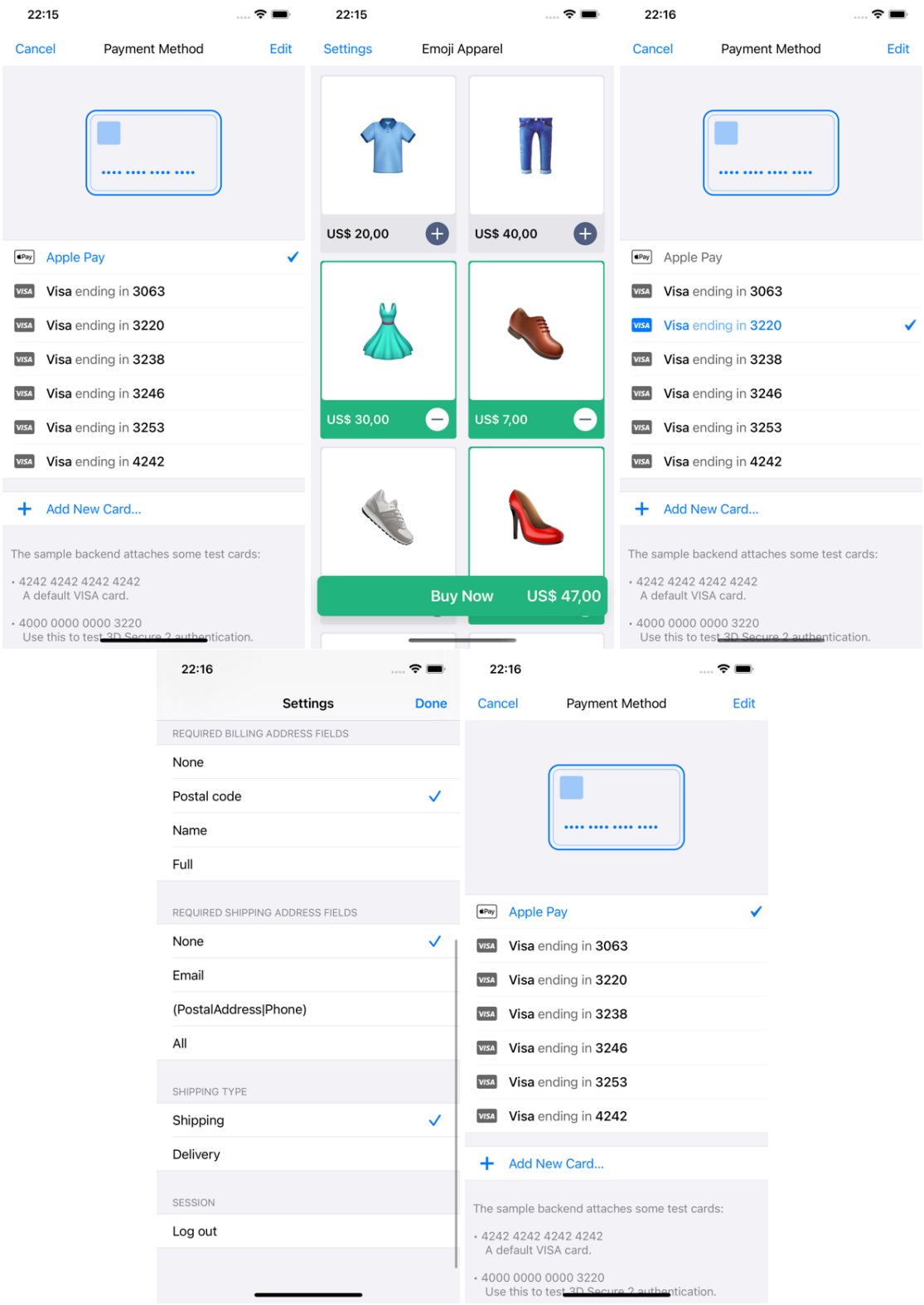

Figura A.17 Fluxo Caso de Teste 5. Fonte: Elaborado pela autora

Referências Bibliográficas

[1] Emil Alégroth. *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*. PhD thesis, Chalmers University of Technology and Goteborg University, 2015.

[2] Apple. User Interface Tests | Apple Developer Documentation. `https://developer.apple.com/documentation/xctest/user_interface_tests`. [Online; accessed 22-July-2023].

[3] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison Wesley, 2009.

[4] Henrique Forioni de Lima. Estudo comparativo de frameworks de automatização de testes de ui para aplicativos ios. Bachelor Thesis, 2019.

[5] Elfriede Dustin. *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison Wesley, 2002.

[6] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126, 2018.

[7] Sidong Feng, Mulong Xie, and Chunyang Chen. Efficiency matters: Speeding up automated testing with gui rendering inference. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 906–918. IEEE Press, 2023.

[8] Shuai Hao, Bin Liu, Suman Nath, Ramesh Govindan, and William G.J. Halfond. Puma: Programmable ui-automation for large scale dynamic analysis of mobile apps. In *The International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, June 2014.

[9] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *2009 International Conference on Software Testing Verification and Validation*, pages 201–209, 2009.

[10] Anureet Kaur. Comparative analysis of line of code metric tools. *International journal of scientific research in science, engineering and technology*, 2:1285–1288, 2016.

[11] Rakesh Kumar Lenka, Srikant Kumar, and Sunakshi Mamgain. Behavior driven development: Tools and challenges. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 1032–1037, 2018.

[12] Joe Ligman, Marco Pistoia, Omer Tripp, and Gegi Thomas. Improving design validation of mobile application user interface implementation. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 277–278, 2016.

[13] Bismal Majeed, Saba Khalil Toor, Kanwal Majeed, and Moazzama Nadeem Ahmad Chaudhary. Comparative study of open source automation testing tools: Selenium, katalon studio & test project. In *2021 International Conference on Innovative Computing (ICIC)*, pages 1–6, 2021.

[14] Meiliana, Irwandhi Septian, Ricky Setiawan Alianto, and Daniel. Comparison analysis of android gui testing frameworks by using an experimental study. *Procedia Computer Science*, 135:736–748, 2018. The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life.

[15] Robert Gomes Melo. Frevo: um framework e uma ferramenta para automação de testes. Master's thesis, Universidade Federal de Pernambuco, 2016.

[16] Andre Augusto Menegassi and Andre Takeshi Endo. Automated tests for cross-platform mobile apps in multiple configurations. *IET Software*, 14(1):27–38, feb 2020.

[17] Helena Olsson, Hiva Alahyari, and Jan Bosch. Climbing the"stairway to heaven"a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. *Proceedings - 38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012*, 2012.

[18] Šarūnas Packevičius, Greta Rudžionienė, and Eduardas Bareiša. Automated visual testing of application user interfaces using static analysis of screenshots. *International Journal of Software Engineering and Knowledge Engineering*, 31(02):167–191, 2021.

[19] Elis Pelivani and Betim Cico. A comparative study of automation testing tools for web applications. In *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2021.

[20] Neha Sharma and Shilpi Singh. Software testing techniques: A literature review. *International Journal of Innovative Research in Technology*, 2020.

[21] Harshit Singh, Shambhu Kumar Jha, Deepa Gupta, and Ajay Vikram Singh. Gui testing android application. In *2022 10th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 1–6, 2022.

[22] Aditya Atul Tirodkar and Sundeep Singh Khandpur. Earlgrey: ios ui automation testing framework. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 12–15, 2019.

[23] Avi Tsadok. *Pro iOS Testing - XCTest Framework for UI and Unit Testing*. Apress, 2020.

[24] Maneela Tuteja and Gaurav Dubey. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2012.

[25] Peter Warren. Teaching programming using scripting languages. *Journal of Computing Sciences in Colleges*, 17:205–216, 2001.

[26] *https://developer.android.com/training/testing/other-components/ui-automator*. [Online; accessed 05-September-2023].