UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Estudo Comparativo de Frameworks de Automatização de Testes de UI para Aplicativos iOS

Henrique Forioni de Lima

São Carlos – SP

Estudo Comparativo de Frameworks de Automatização de Testes de UI para Aplicativos iOS

Henrique Forioni de Lima

Orientador: **Profa. Simone do Rocio Senger de Souza**

Coorientador: **Ricardo Ferreira Vilela**

Monografia final de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.
Área de Concentração: Testes de Software

USP – São Carlos
Outubro de 2019

RESUMO

LIMA, H. F.. **Estudo Comparativo de Frameworks de Automatização de Testes de UI para Aplicativos iOS** . 2019. 54 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

O presente trabalho tem por finalidade comparar três ferramentas de automatização de testes de UI na plataforma iOS, sendo elas: XCUITest, EarlGrey e KIF. A comparação será feita por meio de experimentação, o qual os três frameworks serão aplicados para testar funcionalidades de um aplicativo desenvolvido pela indústria de *software*. As funcionalidades foram testadas por meio de casos de testes definidos neste trabalho. Após a conclusão dos testes, os frameworks foram avaliados por meio de critérios de avaliação. Alguns dos critérios analisados neste trabalho são: tempo de execução, documentação, configuração inicial, depuração, interação com elementos, suporte a recursos de gravação e sincronização.

Palavras-chave: Engenharia de Software, Teste de Software, Teste de Interface de Usuário, iOS.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contextualização e Motivação

Mais de 3 bilhões de pessoas no mundo possuem um *smartphone* hoje em dia, sendo o sistema operacional iOS responsável por 22% deste número, (STATISTA. . . , 2019). Todos os dias milhões de usuários dependem de aplicativos móveis para navegar pela internet, acessar redes sociais (Facebook, Instagram, Twitter), acessar o e-mail, realizar transações bancárias entre outras diversas atividades do dia a dia. Neste sentido, muitas empresas utilizam aplicativos móveis como principal foco de negócio (Uber, iFood, Nubank), enquanto outras como forma de alavancar os negócios e atender a grande demanda.

Usuários facilmente perdem interesse ou desinstalam aplicativos que tiveram uma experiência ruim em decorrência de algum problema, por esta razão o *feedback* dos usuários é um fator fundamental para o sucesso de um aplicativo. Neste contexto, levando em consideração o crescente impacto económico associado aos aplicativos móveis, fica evidente a necessidade de aplicativos confiáveis e consequentemente a realização de testes para garantir a qualidade do produto, melhorando assim a satisfação do usuário.

Como a interação com o aplicativo é feita pela interface do usuário, testes de interface de usuário (User Interface - UI) são extremamente úteis para garantir uma ótima experiência do usuário. No entanto, com o aumento da complexidade dos aplicativos, a processo de testes de UI feitos manualmente se torna muito custoso e muitas vezes inviável.

Neste contexto, a automatização dos testes é recomendada quando se deseja aumentar a eficiência do processo de teste ou para aumentar a confiabilidade dos testes, o que pode ser bastante útil em contextos onde prazos de entrega são curtos, (INTRODUçãO. . . , 2019).

Entretanto, um obstáculo para a automatização de testes de UI pode ser a falta de conhecimento sobre as ferramentas disponíveis, como frameworks, pois existem diversas ferramentas de automatização, com diferentes especificações e propósitos distintos, para diferentes plataformas. Dependendo do cenário, uma ferramenta pode ser mais eficaz que outra, em outros casos a ferramenta pode ser inviável para a situação. Por estas razões a escolha de uma ferramenta de automatização pode se tornar um processo desafiador.

Nenhuma ferramenta de automatização de teste de UI pode ser considerada superior às demais, considerando todos aspectos de automação e teste. Desta forma, diversas ferramentas

devem ser comparadas para encontrar características distintas entre elas e assim identificar em quais cenários essas ferramentas são mais eficazes, (MEILIANAA IRWANDHI SEPTIANA, 2018).

Nesse contexto, a experimentação contribui para verificação de novas teorias. Por meio de experimentos é possível explorar fatores críticos e elucidar novos fenómenos para que as teorias possam ser formuladas e então corrigidas. A experimentação oferece o modo sistemático, disciplinado, computável e controlado para avaliação da atividade humana. Novos métodos, técnicas, linguagens e ferramentas não devem ser apenas propostos sem experimentação e validação, (TRAVASSOS, 2002).

Diante dos aspectos observados, o objetivo principal deste estudo é demonstrar o estado da arte sobre frameworks de automatização de teste UI para aplicações iOS, por meio da condução de um estudo experimental que dispõe-se a levantar os principais aspectos sobre as características desses frameworks. Além disso, uma das motivações deste trabalho é estimular a importância dos testes automatizados de UI. Desta forma este trabalho apresenta a proposta de avaliar três frameworks de automatização de testes de UI: XCUITest, EarlGrey e KIF, que serão investigados por meio de um estudo experimental.

1.2 Objetivos

A proposta deste estudo é investigar as vantagens, desvantagens e limitações de três frameworks de testes de interface quando aplicados para o teste de funcionalidades em aplicativos iOS. Para isso pretende-se desenvolver um estudo experimental com o intuito de obter evidências sobre os frameworks investigados por meio de critérios de avaliação definidos neste trabalho.

Além disso, deseja-se demonstrar quais são os pontos fortes e fracos dos frameworks para utilização na automatização dos testes de UI, de forma que seja possível descobrir características distintas entre os frameworks, contribuindo na tarefa de escolha do framework mais adequado para o cenário do desenvolvedor.

1.3 Organização do Trabalho

Este trabalho está estruturado em quatro capítulos, conforme descrito a seguir. No capítulo 1 é apresentado a contextualização, motivação e objetivos do trabalho. No capítulo 2 é apresentado uma revisão bibliográfica sobre os conhecimentos necessários para o entendimento do trabalho. No capítulo 3 é descrito o planejamento do estudo comparativo, abordando a metodologia, seleção das variáveis envolvidas (frameworks, critérios, casos de testes) e o que foi feito para preparação. No capítulo 4 é apresentado os resultados obtidos em tabelas, e em seguida é feito uma discussão aprofundada sobre as diferenças dos frameworks investigados. No capítulo 5 apresentam-se as conclusões e contribuições.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

2.1 Considerações Iniciais

Nesta seção será explicado a definição e importância do teste de software, também explicará sobre fases e técnicas de teste. Em seguida será abordado conceitos de testes dentro do contexto de aplicativos móveis, como tecnologias e ferramentas utilizadas na plataforma iOS.

2.2 Teste de Software

2.2.1 Importância e Definição de teste

Dificilmente alguma pessoa completa o dia sem a participação de algum software, seja para auxiliar em alguma atividade, entretenimento ou até em situações de alto risco, consequentemente erros cometidos pelos desenvolvedores podem causar desde inconveniência para o usuário até graves acidentes. Como tecnicamente é impossível desenvolver um programa que seja completamente livre de falhas, todo software sempre precisa ser testado (JORGENSEN, 2013).

Teste de software pode ser definido como um processo ou uma série de atividades com o objetivo de verificar se o software faz aquilo que foi proposto e não faz algo não intencional (MYERS, 2004). Testes não são feitos somente para encontrar defeitos, mas também para assegurar a aceitabilidade e a qualidade de um produto, e o sucesso de qualquer produto de software depende grandemente de sua qualidade.

Um estudo realizado pelo *IBM System Science Institute* concluiu que o custo relativo para consertar defeitos encontrados durante etapas finais do desenvolvimento cresce drasticamente comparado às etapas inicias. Defeitos encontrados na fase de manutenção são cerca de 15 vezes mais custosos do que aqueles encontrados durante a implementação, (DAWSON *et al.*, 2010). Por este motivo é importante encontrar defeitos o mais rápido possível, logo é fundamental que durante o desenvolvimento do *software*, sejam aplicados técnicas, estratégias e ferramentas que permitam realizar a atividade de testes de maneira eficaz, de modo a aumentar a qualidade e diminuir custos do projeto.

2.2.2 Fases de Teste

A atividade de testes é dividida em diferentes fases. Cada fase possui um objetivo específico e são focadas em diferentes níveis do desenvolvimento do sistema. O objetivo é dividir os testes de maneira incremental, iniciando por testes em unidades, em seguida é testado a integração entre estas unidades, então é testado o sistema por completo e por fim verifica-se a aceitação, (ISTQB, 2018). Assim podemos estabelecer as seguintes fases: teste de unidade, teste de integração, teste de sistema e teste de aceitação. Cada uma destas fases possuem abordagens e responsabilidades diferentes que serão explicadas a seguir.

Teste de Unidade

Nesta fase de teste o foco são componentes (unidades) que são testados de forma isolada do resto do sistema. As unidades podem ser métodos, classes, funções ou qualquer parte pequena e testável do programa. Por este motivo o teste de unidade pode ser realizado enquanto o sistema ainda está em desenvolvimento e normalmente é realizado pelo próprio desenvolvedor. O objetivo é encontrar defeitos nas menores unidades para evitar que erros sejam propagados em níveis mais altos do teste.

Teste de Integração

O teste de integração tem como foco testar a interação entre duas ou mais unidades trabalhando em conjunto. É possível que duas unidades, que passaram nos testes unitários, apresentem defeitos ao serem testadas agrupadas. Portanto, o teste de integração não foca na funcionalidade individual dos componentes presentes no teste, a intenção é procurar defeitos na comunicação entre os componentes. É fundamental que tenha sido realizado testes de unidade antes da realização do teste de integração, para que nenhum defeito de uma unidade se propague neste nível.

Nesta fase procura-se defeitos na comunicação, manipulação de dados, trocas de mensagens e incompatibilidades de interface entre unidades do sistema.

Assim como no teste de unidade, o teste de integração exige conhecimento sobre a estrutura interna do código, por isso é normalmente realizado pelos desenvolvedores.

Teste de Sistema

O teste de sistema se concentra em testar um sistema completo e integrado, tendo como objetivo principal verificar o comportamento geral do *software* de acordo com os requerimentos do produto. Ao contrário dos testes de unidade e integração, não é necessário conhecimento do código e aspectos internos sobre o desenvolvimento, portanto é normalmente realizado por testadores independentes com uma abordagem de teste caixa-preta, ou seja, preocupa-se apenas com as entradas e saídas.

É importante que, para aumentar a eficácia dos testes, as especificações do produto e o comportamento esperado do programa estejam claros e bem documentados.

Teste de Aceitação

De maneira lógica, após o produto passar pelos testes de sistema, encontra-se muito próximo ou já está na sua fase final, portante nesta fase de teste espera-se que não seja encontrado uma quantidade significativa de defeitos, pois pode representar um risco ao projeto. A responsabilidade principal no teste de aceitação é verificar se o produto está pronto para ser entregue ao cliente (usuário final).

Diferente do teste de sistema, será avaliado os requerimentos do negócio, que podem incluir questões legais ou regulatórias, além de verificar se o produto é compatível com as necessidades do usuário, ou seja, sua aceitação. Por isso o foco dos testes é a experiência do usuário final. Alguns dos testes mais comuns nesta fase são: teste *alpha*, quando realizado no ambiente de desenvolvimento, e teste *beta*, quando realizado no ambiente do cliente, (ISTQB, 2018).

2.2.3 Técnicas de Teste

Durantes as fases de testes, diferentes técnicas de teste podem ser aplicadas. Para a escolha da técnica deve ser levado em consideração o conhecimento do testador, tipo do sistema e ferramentas disponíveis. As técnicas são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste e existem dois tipos: funcional e estrutural, (DO *et al.*, 2000).

Funcional (Caixa-Preta)

Esta técnica de teste é realizada baseado apenas nas entradas e saídas do *software*, de acordo com suas especificações. Deste modo, não é considerado a estrutura interna do programa e o testador não necessita saber sobre como foi implementado, ou qual tecnologia foi utilizada. Os testes são executados e os resultados obtidos são comparados com resultados esperados. O teste de interface do usuário é um exemplo desta técnica, onde as funcionalidades são testadas apenas interagindo com a interface gráfica, sem conhecimento sobre o código.

Estrutural (Caixa-Branca)

O teste estrutural, diferente do teste de caixa-preta, avalia o comportamento interno do *software*, portanto o testador possui conhecimento da implementação e estrutura interna do programa testado. As entradas são escolhidas analisando o código fonte e elaborando casos de testes que cubram todos (ou a maioria) dos possíveis fluxos do código, também pode ser analisado estados internos do programa. Um exemplo desta técnica é o teste unitário.

2.3 Teste em Dispositivos Móveis

2.3.1 Testes de Interface do Usuário (UI)

O teste de interface do usuário em aplicativos móveis tem como objetivo verificar o correto funcionamento das funcionalidades do aplicativo, na mesma perspectiva de um usuário. Enquanto testes unitários e de integração são fundamentais para garantir a qualidade de um aplicativo, o teste de UI tem como foco a experiência final do usuário. Atualmente, é mais comum que testes de UI em aplicativos móveis seja feito de maneira manual. Normalmente o testador segue um roteiro, que consiste em instruções passo a passo, e interage com o dispositivo como um usuário, visualmente verificando os resultados. Em situações otimistas, onde o roteiro e a documentação do aplicativo são bem definidas, o teste manual pode produzir resultados satisfatórios. No entanto, o processo pode facilmente se tornar excessivamente tedioso devido à quantidade de procedimentos a serem testados, além de estar sujeito a falha humana. O tempo gasto nos testes manuais também é um fator que pode aumentar o custo de desenvolvimento do aplicativo.

O teste de UI automatizado utiliza ferramentas capazes de reproduzir a interação com o aplicativo, sendo possível controlar a execução dos testes e comparar resultados esperados dos obtidos de maneira automatizada. Desta forma elimina a possibilidade de falhas humanas e aumenta a eficiência do processo de testes, reduzindo o tempo gasto nestas atividades, (BARTLEY, 2008).

2.3.2 Ambiente de Teste

Existem dois tipos de ambiente nos quais os testes de UI podem ser realizados: em um dispositivo real ou em um simulador. A principal vantagem de utilizar um dispositivo real é obter uma experiência mais próxima possível da realidade, o que permite analisar como o aplicativo irá se comportar no mesmo dispositivo em que os usuários estarão usando. Por outro lado, a utilização do simulador possui vantagens como escalabilidade, baixo custo e praticidade, pois é possível testar diferentes modelos de celular em uma única máquina, sem a necessidade de obter os dispositivos reais.

Atualmente o mercado de dispositivos móveis no mundo possui uma grande variedade de aparelhos, com diferenças no tamanho de tela, capacidade de processamento e memória. De acordo com uma pesquisa feita pela (MIXPANEL, 2018), sobre a fragmentação dos usuários de iPhone ao redor do mundo, visto na Figura 1, nota-se que existem pelo menos 11 modelos diferentes de iPhone com um número relevante de usuários (acima de 3%).

Idealmente, qualquer tipo de teste que possa ter influência das particularidades de cada dispositivo, deve ser testado em todas as variações de modelos em que o aplicativo está disponível, que na maioria das vezes é inviável para o testador. Por este motivo, o uso do simulador é bastante

útil para contornar esta situação.

Figura 1 – Porcentagem da utilização de modelos de iPhone ao redor do mundo.

Fonte: (MIXPANEL, 2018)

2.3.3 Acessibilidade

A Apple fornece ferramentas que possibilitam pessoas com alguma deficiência utilizar seu sistema operacional. Isto inclui pessoas com deficiências visuais, motoras e auditivas. Um recurso interessante é o *VoiceOver*[1], o qual o sistema operacional lê e descreve para o usuário todos os elementos presentes na tela, de forma que o usuário seja capaz de interpretar e interagir com os aplicativos sem a necessidade de visualizar a tela do dispositivo.

O funcionamento deste recurso depende das propriedades de acessibilidade, que estão disponíveis para os desenvolvedores nos elementos da plataforma. Algumas destas propriedades são:

- *accessibility label*: um rótulo sucinto que descreve um elemento, essa propiedade é usada para ser lida pelo *VoiceOver*
- *accessibility identifier*: Utilizado apenas pelo desenvolvedor para identificar unicamente um elemento, o usuário não tem acesso.

[1] https://www.apple.com/br/accessibility/mac/vision/

- *accessibility hint*: Uma breve descrição sobre qual ação o elemento realiza.
- *accessibility value*: O valor atual do elemento, por exemplo, para um campo de texto, essa propiedade representa seu conteúdo.

Além da importância em garantir a acessibilidade do aplicativo para pessoas com deficiência visual, estas propriedades são utilizadas pelos frameworks de automatização de testes para identificar os elementos que serão interagidos. Portanto, são fundamentais para realização de testes de UI.

As duas propriedades mais utilizadas pelos frameworks para identificar um elemento são: *accessibility label* e *accessibility identifier*. A vantagem do *accessibility identifier* é que o desenvolvedor pode definir qualquer identificador para esta propriedade, sem se preocupar com o usuário, já o *accessibility label* deve sempre conter um identificador relevante para o usuário, pois este será utilizado para guiá-lo no *VoiceOver*.

2.3.4 Ferramentas de teste para plataforma iOS

2.3.4.1 XCode

Criado pela Apple, é o principal Ambiente de Desenvolvimento Integrado (*IDE*) para o desenvolvimento de aplicativos para Mac, iPhone, iPad, Apple Watch e Apple TV. O Xcode possui um completo conjunto de ferramentas para desenvolvedores programar, criar interfaces de usuário (Interface Builder[2]), realizar testes unitários e de interface, entre diversos outros recursos.

2.3.4.2 XCTest

XCTest é um framework de testes unitários também capaz de realizar medições de performance, já integrado no XCode desde a versão 5, (XCTEST,). Os testes podem verificar condições, especificadas pelo desenvolvedor, durante a execução do código. Caso alguma condição presente no teste não seja satisfeita, o teste ira falhar. É possível visualizar de forma detalhada os resultados obtidos pelos testes em forma de relatório, que pode ser visto pelo XCode.

2.3.4.3 XCUITest

Em 2015, a Apple anunciou o framework de automação de testes de interface de usuário, o XCUITest, construído em cima do XCTest, fornece uma API focada na realização de testes de UI, (XCUITEST, 2019). Os testes podem ser escritos em Objective-C e Swift e a interação com a aplicação é feita unicamente por três classes, são elas: XCUIApplication, XCUIElement e XCUIElementQuery, abaixo é feito um descrição sobre cada uma.

[2] https://developer.apple.com/xcode/interface-builder/

XCUIApplication: É o *proxy* para a aplicação que está sendo testada. Pode ser considerado o componente raiz na árvore que representa a hierarquia de os todos elementos presentes na tela. É a partir desta classe que as buscas por elementos são feitas. O código abaixo exibe como obter uma instância desta classe.

```
let app = XCUIApplication()
```

XCUIElementQuery: Representa uma busca para localizar elementos na tela, uma busca retorna todos os elementos encontrados que correspondem aos parâmetros fornecidos. O código abaixo exibe um exemplo de uma busca por todos os botões do aplicativo.

```
let buttonsQuery = app.buttons
```

XCUIElement: Corresponde a um único elemento de UI na aplicação. Com este elemento é possível realizar ações e gestos, como clicar, tocar, deslizar e digitar texto. Além disso é possível acessar algumas propriedades deste elemento, como checar se ele existe. O código abaixo exibe como obter um elemento a partir de uma busca, e então realiza uma ação nele.

```
let button = buttonsQuery["id"]
button.tap()
```

2.3.4.4 EarlGrey

EarlGrey é um framework desenvolvido pela Google, teve seu código aberto ao público em 2016 e seu diferencial são os recursos aprimorados de sincronização, que propõe aumentar a estabilidade dos testes e torná-los altamente repetíveis, (EARLGREY, 2019). Earlgrey suporta Swift e Objective-C como linguagem para escrita dos testes.

A estrutura básica para interagir com elementos do aplicativo pode ser vista abaixo:

```
EarlGrey.selectElement(with: MATCHER)
        .perform(ACTION)
```

Onde *MATCHER* é utilizado para identificar elementos e pode ser qualquer item do conjunto da API de seleção, chamado GREYMatcher e *ACTION* representa uma ação da API chamada GREYAction. Abaixo é possível ver alguns dos valores disponíveis.

API de Seleção (GREYMatcher):

```
grey_sufficientlyVisible()
grey_kindOfClass(UITextField.self)
grey_accessibilityID("Botao")
```

```
grey_accessibilityLabel("Botao")
```

API de Ações (GREYAction):

```
grey_tap()
grey_doubleTap()
grey_typeText("Texto")
```

2.3.4.5 KIF

KIF, do inglês "Keep it Functional", é um popular framework de código aberto desenvolvido pela Square em 2011, tem como proposta ser fácil de utilizar ao mesmo tempo que alavanca as propriedades de acessibilidade fornecida pela plataforma iOS. Embora tenha sido escrito em Objective-C, também é suportado a linguagem Swift para escrita dos testes, (KIF, 2019).

Para interagir com os elementos, o framework fornece funções específicas para cada uma das ações disponíveis. Desta forma, é necessário apenas chamar a função correspondente a ação que deseja-se realizar, um dos argumentos da função será uma propriedade de acessibilidade, para identificar o elemento no qual a ação será realizada.

```
kif().tapView(withAccessibilityID: "id1")
kif().enterText("teste", intoViewWithAccessibilityLabel: "Campo
    de Texto")
```

O código acima mostra um exemplo para clicar em um elemento que possui a propriedade *accessibility identifier* igual a "id1", e digitar o texto "teste"dentro de um elemento que possui o *accessibility label* igual a "Campo de Texto".

2.4 Considerações Finais

Nessa seção, foram apresentados conceitos e ferramentas que foram utilizados no desenvolvimento do estudo experimental, os quais são fundamentais para o entendimento no decorrer do trabalho. No próximo capítulo será apresentado o planejamento do trabalho.

Capítulo 3

PLANEJAMENTO DO TRABALHO

3.1 Metodologia

Esta seção descreve o método aplicado para atingir os objetivos deste estudo comparativo.

Primeiramente serão definidos os critérios de avaliação, os quais serão utilizados para avaliar os frameworks investigados. A análise destes critérios será feita por meio de pesquisa e experimentação, os frameworks serão investigados por múltiplas fontes e então serão aplicados na prática, por meio de realização de testes de interface em uma aplicação real desenvolvida pela indústria de software. Portanto, com a condução deste estudo experimental espera-se identificar resultados mais significativos sobre os frameworks, avaliados de forma imparcial.

Deste modo, o estudo comparativo será conduzido da seguinte forma:

1. Estudo completo dos frameworks e suas APIs
2. Configuração dos frameworks no projeto do aplicativo alvo
3. Projeto e desenvolvimento dos casos de testes
4. Execução dos casos de testes
5. Análise dos resultados de acordo com os critérios de avaliação
6. Tabulação e discussão

3.2 Planejamento

Esta seção tem como objetivo descrever as informações necessárias para condução deste estudo. Será explicado sobre a escolha dos frameworks, critérios de avaliação, objeto de teste (aplicativo) e os casos de testes.

3.2.1 Seleção dos Frameworks

Para identificação dos frameworks que serão investigados neste estudo foi considerado apenas os mais utilizados e exclusivos para plataforma iOS. Após pesquisas em diversas fontes relacionadas ao desenvolvimento iOS, conclui-se que os três frameworks mais populares e bem

aceitos pela comunidade são: XCUITest, EarlGrey e KIF. Frameworks que não são atualizados há anos ou que são pouco utilizados atualmente, como o *Frank*[1] foram desconsiderados.

Em relação a versão dos frameworks, foi considerado a versão mais recente e estável até o momento, portanto os frameworks avaliados neste estudo são:

1. XCUITest (XCode 10.2)
2. EarlGrey 1.15.1
3. KIF (Keep It Functional) 3.7.8

Todos frameworks possuem integração com o XCode e suportam Swift e Objective-C como linguagem para escrita dos testes. Uma diferença é que testes executados pelo XCUITest são executados em um processo separado do processo da aplicação, enquanto os frameworks EarlGrey e KIF executam no mesmo processo da aplicação. A tabela 1 apresenta um resumo das características destes frameworks.

Tabela 1 – Características dos frameworks investigados

	XCUITest	**EarlGrey**	**KIF**
Desenvolvido por	Apple	Google	Square
Código	Fechado	Aberto	Aberto
Linguagem	Swift/Objective-C	Swift/Objective-C	Swift/Objective-C
Processo	Processo separado da aplicação	Mesmo da aplicação	Mesmo da aplicação

3.2.2 Seleção dos Critérios de Avaliação

A qualidade de um framework de automatização pode ser medida por diversos fatores, que dependem do contexto e do objetivo do desenvolvedor, ainda assim, é possível definir alguns requisitos básicos que caracterizam um bom framework, de acordo com (HAO B. LIU; GOVINDA, 2014), o autor cita os seguintes pontos:

- *Suporte para uma grande variedade de propiedades*: Um dos objetivos de uma ferramenta de automatização de UI é analisar as propiedades do aplicativo, porém é impraticável prever quais propiedades serão úteis para todos tipos de análises. Portanto, o framework deve fornecer um conjunto de abstrações suficientes para o usuário especificar as propiedades de interesse.
- *Flexibilidade na exploração de estado*: O framework deve permitir que o usuário customize, em alto nível de abstração, a exploração com os elementos de UI, ou seja, pode-se definir a ordem e quais ações serão realizadas além de checar estados dos elementos. Permitir que essas decisões sejam feitas programaticamente torna possível optimizar o comportamento de acordo com a análise a ser feita.

[1] https://github.com/TestingWithFrank/Frank

- *Acesso ao estado do aplicativo*: Além de ter acesso as propiedades dos elementos de UI, muitas análise são necessárias acessar alguns estados internos do aplicativo.
- *Linguagem de script*: A legibilidade e manutenção de testes de UI são aspectos importates, pois o entendimento do código facilita a corretude dos testes e futuras modificações, além de que se a ferramenta necessita aprender uma linguagem de programação nova, pode dificultar a motivação para escrita dos testes.

Além dos aspectos mencionados acimas, existem várias outras formas de avaliar um framework de testes de UI, de acordo com (MEILIANAA IRWANDHI SEPTIANA, 2018), alguns requisitos importantes são:

- *Tempo de execução*: Espera-se que os testes automatizados sejam mais rápidos que o manual, e o tempo pode ter um impacto significativo no custo do processo de testes principalmente quando há uma grande quantidade de casos de testes.
- *Depuração*: Fornecer recursos de depuração é importante para detectar problemas e corrigi-los de maneira rápida e eficiente, além de aumentar a confiabilidade dos testes ao permitir que o testador obtenha informações relevantes sobre a execução dos testes.
- *Suporte a Gravação*: O recurso de gravação permite criar *scripts* de testes a partir de interações manuais no aplicativo. É um recurso que pode ser muito útil para agilizar o processo de desenvolvimento dos testes.
- *Configuração Inicial*: Um processo complexo e demorado de configuração inicial para começar utilizar um framework pode prejudicar sua adoção.
- *Suporte Simulador/Físico*: Suporte a testes realizados tanto em emuladores quanto em aparelhos reais permite maior flexibilidade para o testador.
- *Documentação*: Boa documentação contribui para a aprendizagem do framework e acelera o processo de desenvolvimento dos testes, diminuindo o tempo gasto buscando por dúvidas em relação à utilização do framework.

Para a elaboração dos critérios de avaliação utilizados neste trabalho, foram levados em considerações todos os requisitos mencionados acima, também foi considerado critérios utilizados em estudos similares, envolvendo a plataforma Android, como em (MEILIANAA IRWANDHI SEPTIANA, 2018) e (SINAGA *et al.*, 2018). Alguns critérios foram adaptados de acordo com as particularidades da plataforma iOS.

Abaixo é apresentado todos os critérios de avaliação que serão aplicados neste estudo comparativo, que foram divididos entre Critérios Gerais e Critérios Técnicos para melhor organização:

Critérios Gerais:

- **CG1 - Configuração inicial:** Este critério avalia a facilidade no processo de configuração do framework.
- **CG2 - Documentação:** Este critério avalia a quantidade e confiabilidade das informações disponíveis sobre o framework, por meios oficiais e não oficiais. Também é analisado a facilidade de aprender sobre sua utilização através de conteúdo disponíveis na internet.
- **CG3 - Depuração:** Este critério avalia os recursos disponíveis do frameworks para facilitar a depuração, como o conteúdo de mensagens de erros, informações relevantes fornecidas e qualquer outra funcionalidade que possa contribuir neste aspecto.
- **CG4 - Suporte simulador/aparelho físico:** Este critério avalia se o framework suporta simuladores e aparelhos físicos para execução dos testes.
- **CG5 - Suporte a recurso de gravação:** Este critério avalia se o framework tem suporte para recursos de gravação.

Critérios Técnicos:

- **CT01 - Clicar em botão:** Este critério avalia se o framework é capaz de clicar em botões.
- **CT02 - Clicar em imagem:** Este critério avalia se o framework é capaz de clicar em imagens.
- **CT03 - Selecionar item no *picker*[2]:** Este critério avalia se o framework é capaz de selecionar um item entre os disponíveis no *picker*.
- **CT04 - Digitar em campo de texto:** Este critério avalia se o framework é capaz de digitar em campos de texto.
- **CT05 - Deslizar a tela (*scroll*):** Este critério avalia se o framework é capaz de deslizar a tela para exibir elementos não visíveis. (Quando o conteúdo presente na tela é maior que o tamanho da tela do celular)
- **CT06 - Clicar em aba de navegação:** Este critério avalia se o framework é capaz de clicar em uma aba de navegação, comum em aplicativos iOS.
- **CT07 - Verificar se um elemento existe:** Este critério avalia se o framework é capaz de verificar a existência de um elemento específico.
- **CT08 - Verificar se um elemento está visível:** Este critério avalia se o framework é capaz de verificar a visibilidade de um elemento específico.
- **CT09 - Suporte a atraso programado:** Este critério avalia se o framework é capaz de aguardar por um determinado tempo até que uma condição seja satisfeita.

[2] Elemento nativo da plataforma iOS que exibe uma lista de itens e permite a seleção de um único item.

- **CT10 - Sincronização com requisições de rede:** Este critério avalia se o framework é capaz de detectar requisições de rede e aguardar automaticamente. (Evita interações enquanto o aplicativo está em um estado indesejado)

- **CT11 - Sincronização com animações:** Este critério avalia se o framework é capaz de detectar animações nos elementos e aguardar automaticamente. (Evita interações enquanto o aplicativo está em um estado indesejado)

- **CT12 - Interagir com *dialogs* da aplicação:** Este critério avalia se o framework é capaz de detectar e interagir com os elementos presentes em um *dialog* dentro da aplicação.

- **CT13 - Interagir com *dialogs* do sistema:** Este critério avalia se o framework é capaz de detectar e interagir com os elementos presentes em um *dialog* exibido pelo sistema operacional.

- **CT14 - Interagir com telas fora da aplicação:** Este critério avalia se o framework é capaz de interagir com os elementos de uma tela que não pertence à aplicação sendo testada.

- **CT15 - Finalizar/iniciar aplicação:** Este critério avalia se o framework é capaz de finalizar e iniciar a aplicação durante a execução dos casos de testes.

- **CT16 - Tempo de execução:** Este critério avalia o tempo levado para realização dos casos de testes.

3.2.3 Seleção do Aplicativo

O aplicativo escolhido para ser utilizado neste trabalho é um aplicativo de seguros médicos, o qual o autor está envolvido no processo de desenvolvimento junto com uma equipe, na empresa Tokenlab[3]. O aplicativo permite que o cliente de uma empresa de seguros médicos possa realizar pedidos de reembolsos, consultar um guia médico, solicitar uma autorização de um procedimento médico, entre outras funcionalidades, tudo através do aplicativo.

A escolha do aplicativo aconteceu considerando os seguintes fatores:

- Disponibilidade do código fonte: O acesso ao código foi liberado pela empresa para realização deste trabalho.

- Complexidade: O aplicativo possui diversas funcionalidades e grande variedade de elementos, portanto será suficiente para ser avaliado por todos os critérios de avaliação definidos. Um aplicativo muito simples poderia limitar os resultados obtidos.

[3] Empresa de desenvolvimento de *software* com sede em São Carlos

O aplicativo escolhido está disponível na App Store[4] e possui milhares de acessos diários, aumentando a relevância do estudo experimental. Além disso, durante seu desenvolvimento, os recursos de acessibilidade foram implementados, portanto, todos os elementos possuem as propriedades *accessibility identifier* e *accessibility label* definidas. Logo, não foi necessário realizar alterações no código do aplicativo para possibilitar a interação com os elementos pelos frameworks.

3.2.4 Seleção dos Casos de Testes

Como dito anteriormente, o aplicativo possui inúmeras funcionalidades e mais de 50 telas, testa-lo por completo não é o objetivo deste estudo, assim faz-se necessário escolher funcionalidades específicas para desenvolver os casos de testes. As seguintes funcionalidades foram escolhidas:

- Cadastro
- *Login* e *Logout*
- Enviar *Feedback*

Levou-se em consideração para a escolha das funcionalidades a diversidade de elementos e interações possíveis. Para cada funcionalidade, foram escritos casos de testes para verificar o comportamento esperado nestes fluxos, mais detalhes sobre cada uma das funcionalidades será apresentado abaixo.

Cadastro: Caso o usuário não esteja registrado no sistema, ele pode se cadastrar preenchendo um formulário com dados pessoais e definindo uma senha de autenticação. O usuário deve clicar no botão '*Primeiro Acesso?*' para iniciar o processo de cadastro, que é feito em duas etapas.

A primeira etapa consiste em fornecer informações como nome, CPF e data de nascimento, caso algum desses campos esteja inválido, o aplicativo exibe uma mensagem de erro indicando qual campo não foi preenchido corretamente. Para avançar para segunda etapa é necessário aceitar os termos de uso. Na segunda etapa o usuário cria uma senha seguindo os requisitos de segurança especificados.

As telas do aplicativo referentes ao fluxo de cadastro podem ser vistas nas Figuras 2, 3 e 4.

[4] Loja oficial da Apple para distribuição de aplicativos da plataforma iOS

Figura 2 – Tela inicial do aplicativo para acessar o fluxo de cadastro

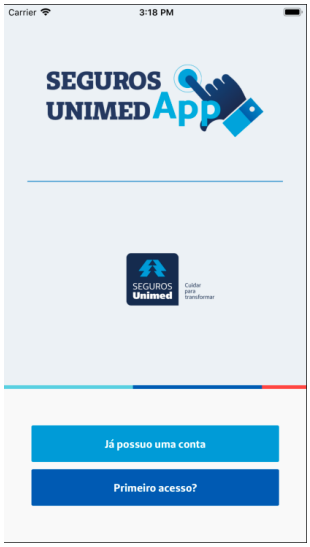


Figura 3 – Tela reference a primeira etapa do fluxo de cadastro do aplicativo

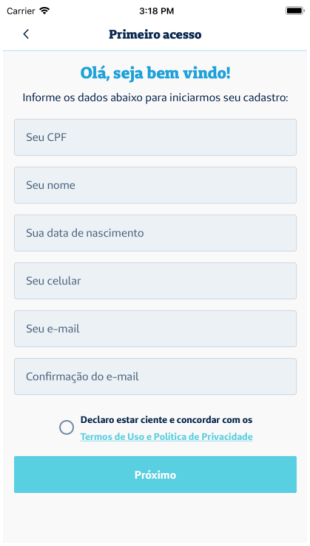


Figura 4 – Tela reference a segunda etapa do fluxo de cadastro do aplicativo

Fonte: Elaborada pelo autor.

***Login* e *Logout*:** Dado que um usuário esteja previamente cadastrado, o aplicativo permite que este usuário acesse sua conta após fornecer seu CPF e senha. Uma vez dentro de sua conta, o usuário pode a qualquer momento encerrar sua sessão navegando para a aba '*MAIS*' e clicando em *Sair*. As Figuras 5 e 6 exibem as telas utilizadas para o fluxo de *Login* e *Logout* respectivamente.

Figura 5 – Tela para realizar o Login no aplicativo

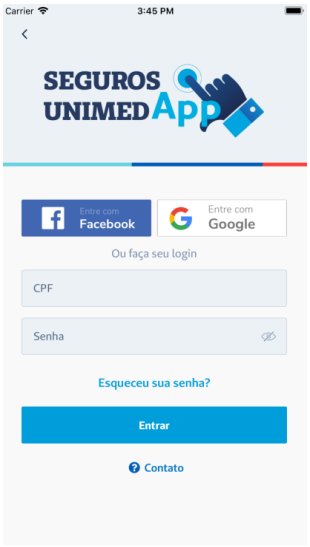


Figura 6 – Tela para realizar o Logout no aplicativo

Fonte: Elaborada pelo autor.

Enviar *Feedback*: Permite que um usuário autenticado envie um *feedback* para a empresa responsável. Dentro do aplicativo o usuário navega para aba *CONTATO* e clica no botão *Compartilhe sua opinião com a gente!*, então ele será direcionado para a tela onde poderá escrever uma mensagem e, opcionalmente, anexar documentos. É necessário escolher um assunto do *feedback* antes de enviar. A Figura 7 exibe a tela de contato e a Figura 8 a tela para enviar *feedback*.

Figura 7 – Tela de contato para acessar a tela de enviar feedback no aplicativo

Figura 8 – Tela de enviar feedback no aplicativo

Fonte: Elaborada pelo autor.

Foram desenvolvidos cinco casos de testes que verificam o funcionamento das funcionalidades descritas acima no aplicativo, os quais serão utilizados no estudo experimental. Cada caso de teste é detalhado abaixo:

Caso de Teste 1	
Funcionalidade:	Cadastro
Objetivo:	Validar um fluxo de cadastro bem sucedido
Etapas:	
1	Clicar no botão “Primeiro acesso?"
2	Digitar CPF válido no campo "Seu CPF"
3	Digitar nome válido no campo "Seu Nome"
4	Selecionar data de nascimento válida no campo "Sua data de nascimento"
5	Digitar celular válido no campo "Seu celular"
6	Digitar e-mail válido no campo "Seu e-mail"
7	Digitar confirmação de e-mail válido no campo "Confirmação do e-mail"
8	Aceitar termos de uso
9	Clicar no botão "Próximo"
10	Digitar senha válida no campo "Sua senha *"
11	Digitar confirmação de senha válida no campo "Confirme sua senha"
12	Clicar no botão "Próximo"
13	Verificar dialog de confirmação

Caso de Teste 2	
Funcionalidade:	Cadastro
Objetivo:	Verificar se uma mensagem de erro é exibida no campo "Seu CPF" ao digitar um CPF inválido e verificar se o aplicativo informa o usuário, através de um *dialog*, caso não tenha aceitado os termos de uso.
Etapas:	
1	Clicar no botão “Primeiro acesso?"
2	Digitar CPF inválido no campo "Seu CPF"
3	Verificar por mensagem de campo inválido
4	Apagar o CPF inválido e digitar um CPF válido
5	Digitar nome válido no campo "Seu Nome"
6	Selecionar data de nascimento válida no campo "Sua data de nascimento"
7	Digitar celular válido no campo "Seu celular"
8	Digitar e-mail válido no campo "Seu e-mail"
9	Digitar confirmação de e-mail válido no campo "Confirmação do e-mail"
10	Clicar no botão "Próximo"
11	Verificar *dialog* informando que termos de uso não foi aceito

Caso de Teste 3	
Funcionalidade:	Login e Logout
Objetivo:	Validar fluxo de *Login* e *Logout* com sucesso
Etapas:	
1	Clicar no botão "Já possuo uma conta"
2	Digitar CPF válido no campo "CPF"
3	Digitar senha válida no campo "Senha"
4	Clicar no botão "Entrar"
5	Verificar se acessou a página inicial
6	Navegar para aba "MAIS"
7	Clicar no botão "Sair"
8	Verificar se encerrou a sessão

Caso de Teste 4	
Funcionalidade:	Enviar Feedback
Objetivo:	Validar um fluxo de enviar feedback com sucesso
Etapas:	
1	Verificar se o usuário está autenticado
2	Navegar para aba "CONTATO"
3	Clicar no botão “Compartilhe sua opinião com a gente"
4	Selecionar um item no campo "Assunto"
5	Escrever uma mensagem no campo "Mensagem"
6	Clicar no botão “Anexar"
7	Aceitar a permissão de acesso aos arquivos do sistema
8	Clicar no botão "Escolher uma foto"
9	Selecionar uma foto da galeria
10	Clicar no botão “Enviar”
11	Verificar *dialog* de sucesso

Caso de Teste 5	
Funcionalidade:	Enviar Feedback
Objetivo:	Verificar se o aplicativo exibe mensagem de erro ao tentar enviar feedback sem escolher um assunto
Etapas:	
1	Verificar se o usuário está autenticado
2	Navegar para aba "CONTATO"
3	Clicar no botão “Compartilhe sua opinião com a gente"
4	Escrever uma mensagem no campo "Mensagem"
5	Clicar no botão “Anexar"
6	Verificar *dialog* com mensagem informando para selecionar um assunto

3.3 Preparação

Para melhorar a organização dos casos de testes e evitar código duplicado, foi aplicado um padrão chamado Robot Pattern [5], criado por Jake Wharton. Neste padrão, é criado uma classe específica (chamada de *Robot*) para cada tela do aplicativo, esta classe ira possuir funções com nomes intuitivos que realizam determinadas ações naquela tela. Com isso, os casos de testes se tornam legíveis e fáceis de entender, pois toda implementação esta omitida dentro destas classes.

A utilização desta técnica, além das vantagens citadas acima, também possibilitou maior confiabilidade no estudo comparativo, pois os três frameworks vão utilizar o mesmo código para os casos de teste, a diferença estará na implementação das funções, onde cada um terá sua própria implementação.

O código dos casos de teste utilizado para os três frameworks pode ser visto abaixo, onde cada função representa a execução de um caso de teste diferente, entre os definidos na seção 3.2.4.

Código-fonte 1: Implementação dos casos de testes

```
1   func testCase1() {
2       LoginRobot()
3           .navigateFirstAccess()
4           .typeCPF(with: "43777129852")
5           .typeName(with: "Henrique Lima")
6           .typeBirthDate()
7           .typeCellphone(with: "333333333")
8           .typeEmail(with: "teste@email.com")
9           .typeConfirmationEmail(with: "teste@email.com")
10          .acceptTerms()
```

[5] https://academy.realm.io/posts/kau-jake-wharton-testing-robots/

```
11              .goToNextStep()
12              .typePassword(with: "Teste123*")
13              .typePasswordConfirmation(with: "Teste123*")
14              .tapFinishButton()
15              .assertFinishedRegistration()
16      }
17
18      func testCase2() {
19        LoginRobot()
20              .navigateFirstAccess()
21              .typeCPF(with: "000000000")
22              .assertInvalidMessage()
23              .clearCPF()
24              .typeCPF(with: "43777129852")
25              .typeName(with: "Henrique Lima")
26              .typeBirthDate()
27              .typeCellphone(with: "333333333")
28              .typeEmail(with: "teste@email.com")
29              .typeConfirmationEmail(with: "teste@email.com")
30              .goToNextStep()
31              .assertInvalidDialog()
32      }
33
34      func testCase3() {
35          LoginRobot()
36              .tapLoginButton()
37              .typeCPF(with: "43777129852")
38              .typePassword(with: "Teste123*")
39              .assertUserLoggedIn()
40              .navigateToMoreOptions()
41              .tapLogoutButton()
42              .assertUserLoggedOut()
43      }
44
45      func testCase4() {
46         LoginRobot()
47           .navigateHome()
48           .navigateToContact()
49           .openFeedback()
```

```
50          .selectFeedbackSubject(with: "Dúvida")
51          .typeFeedbackMessage(with: "Testando 123")
52          .addFeedbackAttachment()
53          .tapSendFeedback()
54          .assertFeedbackSent()
55      }
56
57      func testCase5() {
58          LoginRobot()
59              .navigateHome()
60              .navigateToContact()
61              .openFeedback()
62              .typeFeedbackMessage(with: "Testando")
63              .tapSendFeedback()
64              .assertMissingSubjectDialog()
65      }
```

Em relação ao ambiente, todos os testes foram executados utilizando o simulador nativo do XCode, o modelo simulado foi o iPhone 7 com sistema operancional iOS 12.0. A máquina onde os testes foram executados é um Mac Mini versão 2012 [6].

[6] https://support.apple.com/kb/sp659?locale=en_US

Capítulo 4

RESULTADOS E DISCUSSÃO

4.1 Resultados

Nesta seção serão apresentados os resultados obtidos após a condução do estudo experimental, por meio de tabelas e gráficos. Os resultados foram divididos entre critérios gerais e técnicos.

4.1.1 Critérios Gerais

Todos os critérios gerais de avaliação, definidos na seção 3.2.2 foram investigados. Para uma análise mais objetiva, os critérios: configuração inicial, documentação e depuração foram divididos em critérios mais específicos relacionados ao assunto abordado.

As Tabelas 2, 3 e 4 apresentam os resultados dos critérios relacionados a configuração inicial, documentação e depuração, respectivamente. Enquanto a Tabela 5 apresenta os resultados dos critérios gerais restantes.

Tabela 2 – Resultados Configuração Inicial

Configuração Inicial				
Código	**Critério**	**XCUITest**	**EarlGrey**	**KIF**
CG01.1	Adiciona dependências externas ao projeto	Não	Sim	Sim
CG01.2	Necessita configuração adicional para Swift	Não	Não	Sim
CG01.3	Necessita execução de *scripts* para instalação	Não	Sim	Não

Tabela 3 – Resultados Documentação

Documentação				
Código	**Critério**	**XCUITest**	**EarlGrey**	**KIF**
CG02.1	Documentação completa e atualizada	Sim	Não	Não
CG02.2	Facilidade em encontrar vídeos informativos e tutoriais	Sim	Não	Não
CG02.3	Facilitade em encontrar exemplos práticos na internet	Sim	Não	Não

Tabela 4 – Resultados Depuração

Depuração				
Código	**Critério**	**XCUITest**	**EarlGrey**	**KIF**
CG03.1	Mensagens de erros detalhadas e informativas	Sim	Sim	Não
CG03.2	Visualização da hierarquia dos elementos na tela	Sim	Sim	Não
CG3.3	Captura de tela automática nas falhas	Sim	Sim	Não
CG3.4	Informações em tempo de execução	Sim	Não	Não

Tabela 5 – Resultados Critérios Gerais

Outros				
Código	**Critério**	**XCUITest**	**EarlGrey**	**KIF**
CG4	Suporte simulador/aparelho real	Sim	Sim	Sim
CG5	Suporte a recurso de gravação	Sim	Não	Não

4.1.2 Critérios Técnicos

Conforme mencionado anteriormente, cinco casos de testes, com objetivos distintos, foram empregados para investigação dos critérios técnicos, os resultados obtidos são apresentados na Tabela 6.

Os frameworks KIF e EarlGrey não foram capazes de executar o Caso de Teste 4, que envolve anexar documentos na tela de *Feedback* (Figura 8), devido a falta de compatibilidade com telas externas ao aplicativo (Critério CT14).

Nenhum outro critério não atendido impossibilitou a execução dos casos de testes, portanto todos os outros casos de testes foram aplicados com sucesso pelos três frameworks investigados.

Tabela 6 – Resultados dos critérios técnicos

Código	**Critério**	**XCUITest**	**EarlGrey**	**KIF**
CT01	Clicar em botão	Sim	Sim	Sim
CT02	Clicar em imagem	Sim	Sim	Sim
CT03	Selecionar item no *picker*	Sim	Sim	Sim
CT04	Digitar em campo de texto	Sim	Sim	Sim
CT05	Deslizar a tela (*scroll*)	Sim	Sim	Sim
CT06	Clicar em aba de navegação	Sim	Sim	Sim
CT07	Verificar se um elemento existe	Sim	Sim	Sim
CT08	Verificar se um elemento está visível	Não	Sim	Não
CT09	Suporte a atraso programado	Sim	Sim	Sim
CT10	Sincronização com requisições de rede	Não	Sim	Não
CT11	Sincronização com animações	Não	Sim	Não
CT12	Interagir com *dialogs* da aplicação	Sim	Sim	Sim
CT13	Interagir com *dialogs* do sistema	Sim	Não	Sim*
CT14	Interagir com telas fora da aplicação	Sim	Não	Não
CT15	Finalizar/iniciar aplicação nos testes	Sim	Não	Não

Em relação ao critério CT13, o framework KIF foi capaz apenas de confirmar o *dialog* mas não conseguiu realizar outros tipos de interações.

A medição do tempo de execução foi feita pelo comando *measure* do XCTest, que executa o código dez vezes seguidas e calcula uma média dos resultados, desta forma os tempos de execução encontrados são mais confiáveis e significativos. Os resultados podem ser vistos na Figura 9.

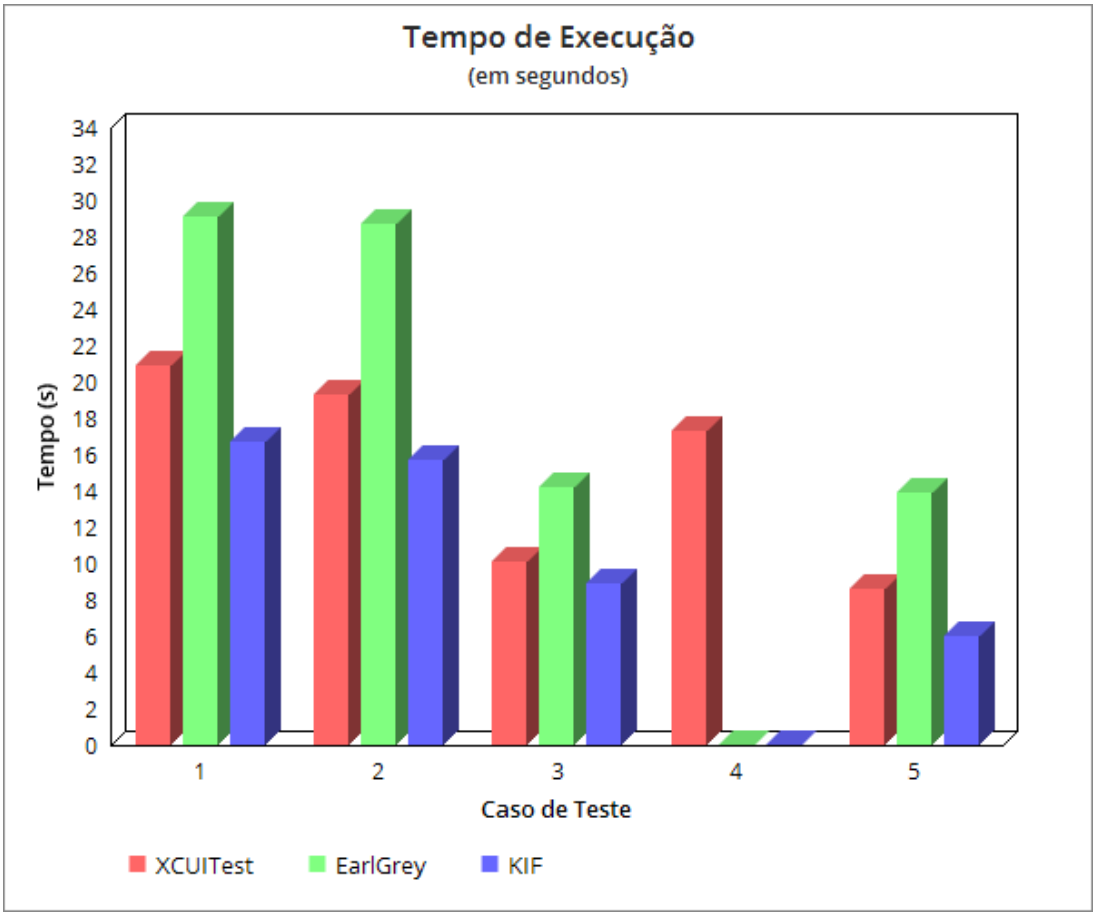


Figura 9 – Tempo de execução (em segundos) dos casos de testes para cada framework

Como KIF e EarlGrey não completaram o Caso de Teste 4, o tempo foi representado como zero.

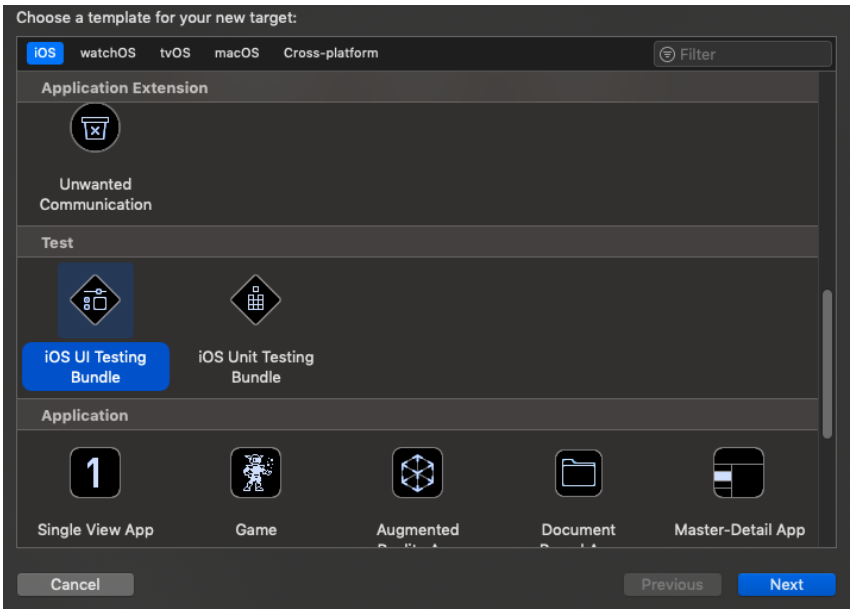
4.2 Análise e Discussão

Nesta seção os frameworks serão abordados com maior nível de detalhamento em relação aos resultados obtidos. A discussão foi dividida em tópicos, que em sua totalidade, abordam todas as particularidades encontradas sobre os frameworks investigados.

4.2.1 Configuração Inicial

Nenhum dos frameworks investigados possuem um processo complexo de configuração inicial. No entanto, o XCUITest leva vantagem neste aspecto, pois por padrão, já está totalmente integrado ao XCode, portanto para configura-lo foi necessário somente adicionar um novo *Target* de testes no projeto, clicando em "*Add Target*", dentro do XCode, e selecionando "*iOS UI Testing Bundle*" como mostra Figura 10.

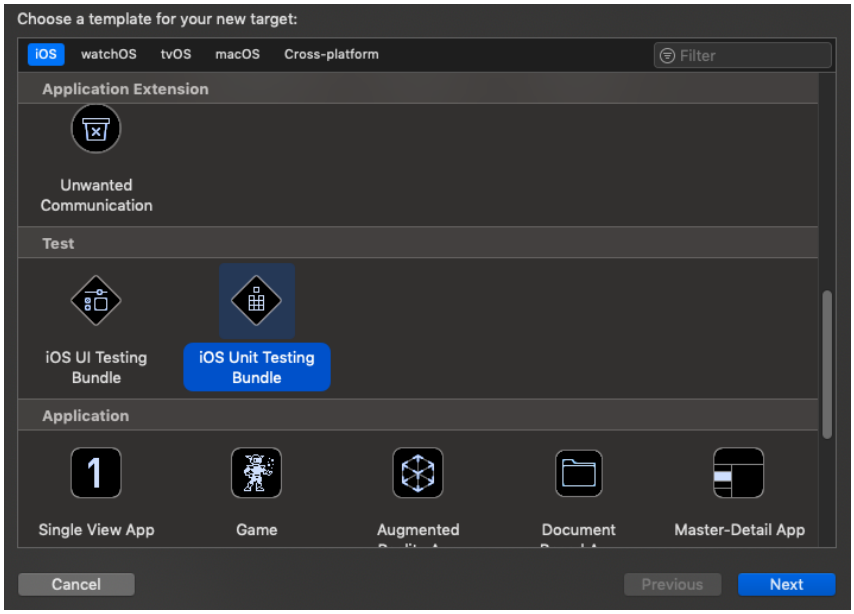
Figura 10 – Etapa de configuração do XCUITest no XCode



Fonte: Elaborada pelo autor.

Para configuração do EarlGrey e do KIF foi necessário adicionar dependências externas ao projeto. Este processo foi realizado com o auxílio do gerenciador de dependências chamado *Carthage* [1]. Em seguida, semelhante ao XCUITest, foi criado um *Target* de testes para cada framework, no entanto foi selecionado a opção *"iOS Unit Testing Bundle"* como mostra a Figura 11.

Figura 11 – Etapa de configuração do EarlGrey e KIF no XCode



Fonte: Elaborada pelo autor.

[1] https://github.com/Carthage/Carthage

Observa-se que EarlGrey e KIF são executados por meio de um *Target* de testes unitários ao invés de UI, por este motivo os testes executam no mesmo processo da aplicação, as consequências desta diferença em relação ao XCUITest serão discutidos na Seção 4.2.7.

Por fim, para o EarlGrey ainda foi necessário executar mais dois comandos no terminal que finalizam sua instalação:

```
gem install earlgrey \\
earlgrey install -t EarlGreyTarget (nome do Target)
```

O framework KIF não necessitou de etapas adicionais na instalação, por outro lado, como a linguagem escolhida para escrita dos testes foi Swift e não Objective-C, foi preciso adicionar o Código-fonte 2 dentro do arquivo de testes:

Código-fonte 2: Código para suporte a linguagem Swift no KIF

```
1 extension XCTestCase {
2     func KIF(file: String = #file, _ line: Int = #line) ->
      KIFUITestActor {
3         return KIFUITestActor(inFile: file,
4                               atLine: line,
5                               delegate: self)
6     }
7 }
8 extension KIFTestActor {
9     func KIF(file: String = #file, _ line: Int = #line) ->
      KIFUITestActor {
10        return KIFUITestActor(inFile: file,
11                              atLine: line,
12                              delegate: self)
13    }
14 }
```

O Código-fonte 2 permite que as funções do framework sejam acessadas, utilizando a linguagem Swift, por meio da função *KIF()*, por exemplo:

```
KIF().tapView(withAccessibilityLabel: "Entrar")
```

4.2.2 Documentação

Entre os 3 frameworks investigados, XCUITest destaca-se pela qualidade e quantidade de documentação. Por ser desenvolvido pela Apple, contém uma documentação oficial[2] além de

[2] https://developer.apple.com/documentation/xctest/user_interface_tests

possuir vídeos informativos criados na WWDC [3].

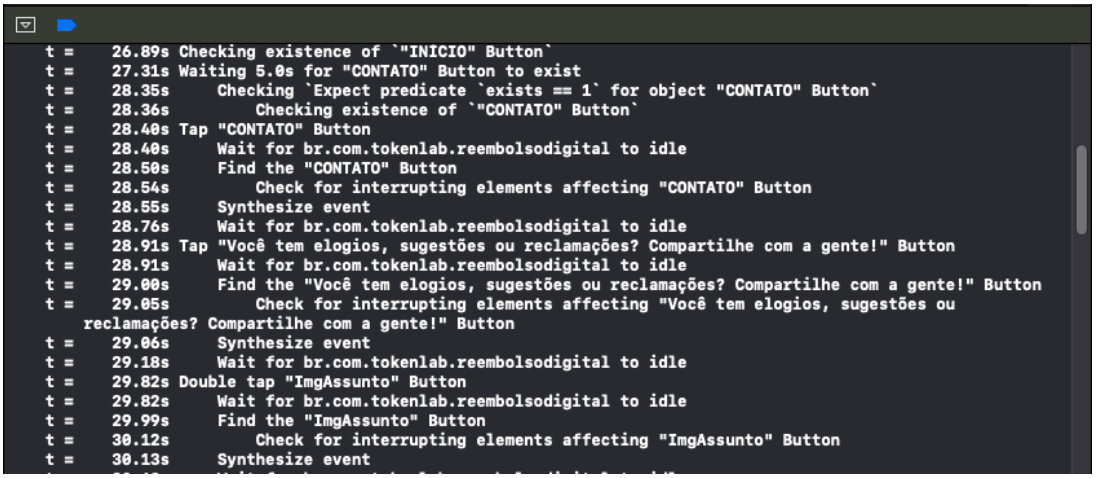
Em relação aos frameworks KIF e EarlGrey, ambos de código aberto, desenvolvidos por colaboradores e não possuem uma empresa específica atrelada ao desenvolvimento, deste modo, as informações sobre ambos são encontradas em seus respectivos repositórios de versionamento. Notou-se que tanto para o KIF quanto EarlGrey a documentação presente no GitHub[4] não era atualizada a mais de 2 anos e o conteúdo era focado na linguagem Objective-C.

Além da documentação oficial, notou-se que XCUITest também possui maior quantidade de conteúdo informativo espalhado na internet, como Youtube[5], postagens em blogs e forúns de discussão, que são recursos bastante úteis para a aprendizagem de cada framework. Exemplos práticos da utilização do framework para o KIF e EarlGrey eram breves e não atendiam as necessidades do autor, por outro lado, foi encontrado muito conteúdo prático relacionado ao XCUITest, apesar de não haver garantia de confiabilidade dos recursos.

4.2.3 Depuração

XCUITest foi o único framework capaz de fornecer um relatório de execução em tempo real. Durante a execução dos casos de testes, foi possível visualizar no console do XCode qual ação estava sendo executada naquele instante (informando o tempo em segundos). A Figura 12 mostra um exemplo deste relatório, enquanto EarlGrey e KIF não possuem recurso semelhante.

Figura 12 – Informações de depuração no console do XCode durante a execução pelo XCUITest



Fonte: Elaborada pelo autor.

Em relação às mensagens de erro exibidas ao ocorrer uma falha no teste, XCUITest e EarlGrey se mostraram superiores ao KIF, fornecendo informações detalhadas sobre possíveis problemas, enquanto o KIF apenas indica brevemente a ação que não foi possível realizar.

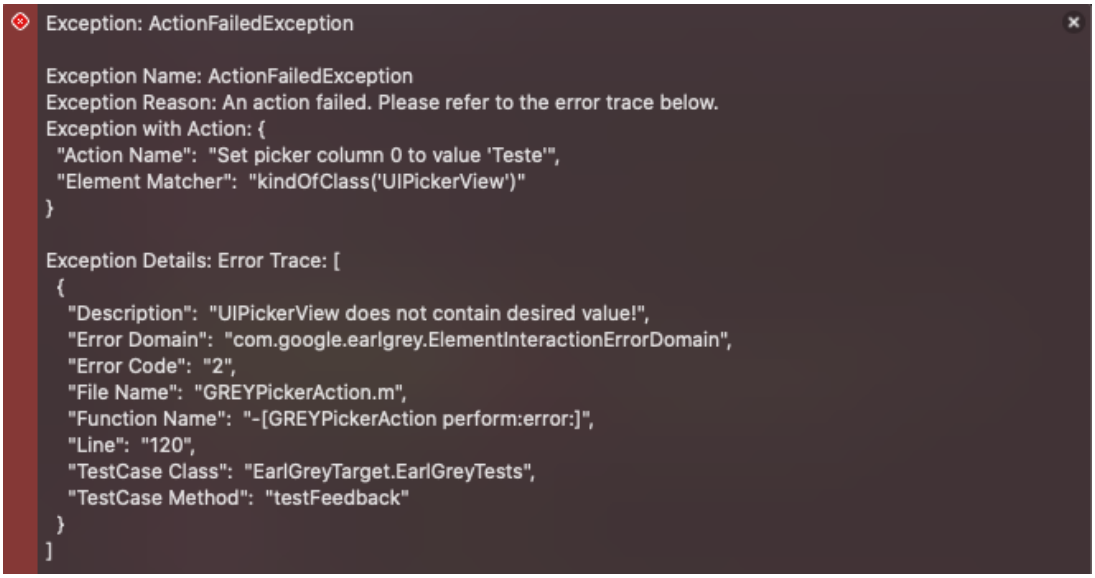
[3] https://developer.apple.com/videos/developer-tools/testing
[4] https://github.com
[5] https://www.youtube.com

No Caso de Teste 4, ao tentar selecionar um item não disponível no *picker* para seleção de um assunto do *feedback*, o framework KIF exibiu a breve mensagem: *"Failed to select from Picker."*, enquanto o framework XCUITest exibiu a seguinte mensagem:

```
Assertion Failure: <unknown>:0: Requested adjust to value 'Teste'
which is not one of the possible values Elogio, Reportar erro,
Sugestão de melhoria, Dúvida, Outros assuntos for the picker wheel
"Selecione um assunto"
```

O erro é bem descrito e os valores possíveis são indicados, o framework EarlGrey exibiu a mensagem da Figura 13, que mostra várias informações sobre o erro.

Figura 13 – Log de erro exibido pelo EarlGrey ao selecionar valor inválido no *Picker*



Fonte: Elaborada pelo autor.

Ainda quando a mensagem de erro não é suficiente, um recurso útil é visualizar a hierarquia de elementos na tela, encontrada pelo framework, para verificar se o elemento desejado está presente. O framework KIF não fornece este recurso, EarlGrey exibe a hierarquia automaticamente quando há uma falha e com o XCUITest, é possível visualizá-la através do comando

```
XCUIApplication().debugDescription
```

Outro aspecto importante para depuração é a capacidade de capturar a tela no momento em que algum teste falha (Critério CG3.3), o que torna mais fácil encontrar possíveis defeitos na aplicação. O experimento mostrou que XCUITest e EarlGrey possuem este recurso por padrão, e foi possível analisar as imagens capturadas pelo próprio XCode, por outro lado, KIF não realizou capturas de tela.

Foi identificado, por pesquisas na internet, a possibilidade de realizar capturas de telas, porém não foi considerado neste experimento pois exige configurações adicionais que não pertencem ao framework, desenvolvidas por outras fontes. Assim optou-se por funcionalidades nativas de modo a garantir uma análise mais justa dos frameworks.

4.2.4 Interação com elementos

Os resultados da Tabela 6 mostram que todos os frameworks conseguem interagir com os elementos mais comuns presentes em aplicações iOS (Critérios CT01, CT02, CT03, CT04, CT05, CT06), o diferencial entre eles é a forma em que estes elementos são encontrados na tela e o modo que a interação é feita. Abaixo serão discutidos estes dois aspectos.

Em relação ao modo de encontrar elementos, com o framework XCUITest é preciso fazer uma busca explícita pelo tipo do elemento desejado. O código abaixo mostra exemplos de como é feito as interações pelo XCUITest:

```
// XCUITest
app.buttons["Login"].tap() // botão
app.staticTexts["Texto"].tap() // textos estáticos
app.checkBoxes["Check"].tap() // checkbox
app.images["Img"].tap() // imagem
app.textFields["CPF"].typeText("00319984028") // campo de texto
app.secureTextFields["Senha"].typeText("Teste123*") // campo de
    senha
app.pickerWheels["Assunto"].adjust(toPickerWheelValue: "Elogio"
    ) // picker
```

Enquanto nos frameworks KIF e EarlGrey, as mesmas interações podem ser feitas sem especificar o tipo do elemento, somente é preciso especificar o tipo do identificador, como pode ser visto abaixo:

```
// EarlGrey:
EarlGrey.selectElement(with: grey_accessibilityLabel("Label")).
    perform(grey_tap())
EarlGrey.selectElement(with: grey_accessibilityID("ID")).
    perform(grey_tap())

//KIF:
KIF().tapView(withAccessibilityLabel: "Label")
```

```
KIF().tapView(withAccessibilityIdentifier: "ID")
```

Opcionalmente, é possível personalizar a busca por elementos no EarlGrey por tipos de elementos específicos de acordo com sua classe, por exemplo: *grey_kindOfClass(UIButton.self)*.

Notou-se que com a abordagem do XCUITest é necessário um conhecimento maior sobre a implementação do aplicativo, pois em alguns casos o tipo do elemento não é explícito e pode ser confundido, como imagens que parecem botões. No Caso de Teste 1, por exemplo, todos os campos de textos foram utilizados o comando *app.textFields* , porém ao tentar utiliza-lo para o campo de senha ocorreu uma falha, então foi preciso alterar para *app.secureTextFields*. Este problema não ocorre nos frameworks KIF e EarlGrey, em que o elemento é encontrado apenas pelo identificador.

Em relação ao modo que as interações são feitas, observa-se que EarlGrey realiza as ações mais próximas de um usuário real. Ao digitar um texto, por exemplo, EarlGrey utiliza o teclado disponível do aparelho, interagindo com todas as teclas necessárias para reproduzir o texto completo, enquanto seus concorrentes adicionam o texto no elemento sem interagir com o teclado presente na tela. Desta maneira, cenários em que apenas o teclado numérico é exibido na tela, ao tentar digitar um texto contendo uma letra, o EarlGrey irá falhar, enquanto XCUITest e KIF não, criando uma situação inconsistente, pois o usuário jamais seria capaz de digitar uma letra.

Outro ponto que favorece o realismo nas interações do EarlGrey é a checagem por visibilidade (critério CT08), o único framework que possui este recurso. Para determinar se um elemento é interagível, EarlGrey faz uma checagem de visibilidade, internamente, analisando os pixels não obstruídos por outro elemento, desta maneira evita-se a interação com um elemento em que o usuário não conseguiria visualizar, tornando o teste mais robusto. Também é possível realizar assertivas em relação à visibilidade de determinados elementos.

Além disso, no Caso de Teste 4, nota-se que ao interagir com o *picker*, visto na Figura 14, para selecionar um item, o EarlGrey desliza o componente até chegar no item desejado, similar a uma interação humana, enquanto KIF e XCUITest modificam o valor instantaneamente.

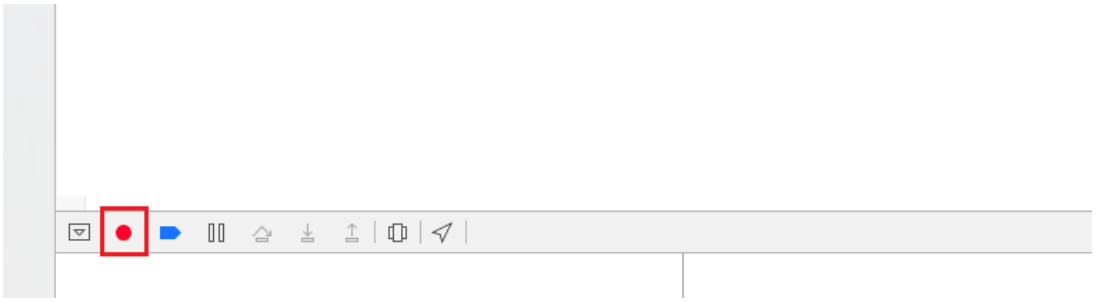
Em relação ao critério CG5, apenas o XCUITest possui recurso de gravação. Ao clicar no botão indicado pela Figura 15, o aplicativo é compilado e executado, então pode-se realizar o teste desejado interagindo com a aplicação simulando a interação de um usuário real, após parar a gravação, é gerado automaticamente o código que reproduz todas as interações feitas manualmente. Este recurso se mostrou bastante útil para iniciar o desenvolvimento dos testes, principalmente pela agilidade e facilidade. Por outro lado, dependendo da complexidade da tela, o recurso nem sempre reproduz com exatidão as ações desejadas, e também notou-se que o código gerado normalmente necessita de alterações para melhor se adequar aos objetivos do teste ou para melhorar a legibilidade do código. Portanto, apesar de ser um recurso facilitador, a medida que o desenvolvedor ganha mais experiência no desenvolvimento dos testes, espera-se

Figura 14 – *Picker* para seleção de assuntos presente no Caso de Teste 4

Fonte: Elaborada pelo autor.

que este recurso se torne menos relevante, pois escrever o código manualmente mostrou-se mais eficiente.

Figura 15 – Botão no XCode para ativar o recurso de gravação do XCUITest



Fonte: Elaborada pelo autor.

4.2.5 Performance

Pela Figura 9, nota-se que o framework KIF possui a melhor performance entre os três, pois obteve o menor tempo de execução em todos os casos de testes. Por outro lado, EarlGrey foi significativamente mais lento, em alguns casos seu tempo de execução levou mais do que o dobro do tempo levado pelo KIF. Uma provável explicação para esse resultado é a abordagem mais realista nas interações com os elementos, presente no EarlGrey, como foi explicado na seção anterior. XCUITest obteve tempos de execução próximos ao KIF, levando poucos segundos a mais.

4.2.6 Sincronização

Durante o desenvolvimento dos testes é necessário levar em consideração que o aplicativo pode estar ocupado em determinados momentos, como quando está aguardando a resposta de uma requisição de rede, carregando alguma tela, ou completando alguma animação. Nestes instantes, o framework pode não conseguir encontrar ou interagir com o elemento desejado.

Por padrão, XCUITest tenta duas vezes, em um curto intervalo de tempo, encontrar um elemento, e caso não encontre nestas tentativas o teste irá falhar, fato que foi constatado através das mensagens exibidas pelo framework em tempo de execução:

```
t =     18.95s          Find the "Anexar" Button (retry 1)
t =     19.99s          Find the "Anexar" Button (retry 2)
```

O framework KIF, por sua vez, tenta encontrar o elemento continuamente durante um tempo pré definido (10 segundos), que pode ser alterado pelas configurações do framework.

Em casos que o comportamento padrão não é suficiente, XCUITest e KIF possuem funções para aguardar um determinado tempo antes de tentar interagir com o elemento (critério CT09), abaixo está o código utilizado para ambos.

```
// XCUITest
    element.waitForExistence(timeout: 10)
// KIF
    KIF().waitForView(withAccessibilityLabel: "elemento")
```

O EarlGrey se destacou neste quesito por possuir recursos de sincronização (critério CT10 e CT11). De acordo com (EARLGREY...,) "*EarlGrey automaticamente aguarda que o aplicativo fique ocioso, rastreando a fila principal de despacho, a fila de operações, de rede e animações, além de vários outros sinais, e realiza interações somente quando o aplicativo está ocioso*". Por este motivo EarlGrey não necessita de maneiras explícitas para aguardar um elemento, uma vez que o recurso de sincronização faz isso automaticamente. A sincronização automática foi muito útil durante os desenvolvimento dos testes pois não foi necessário acrescentar códigos adicionais para tratar estas situações.

Um problema com tempos de espera fixados pelo desenvolvedor, como no caso do XCUITest e KIF, é a imprevisibilidade do teste, uma vez que é impossível saber com exatidão quantos segundos uma requisição de rede pode levar, por exemplo. Tempos longos demais podem reduzir a performance do teste, enquanto tempos curtos podem gerar resultados incorretos. Por este motivo a dependência de valores fixos é um fator que impacta a confiabilidade dos testes. Neste sentido, o EarlGrey teve vantagem em relação aos outros.

4.2.7 Processo

Como dito anteriormente, tanto os testes no EarlGrey quanto no KIF são executados no mesmo processo da aplicação, enquanto no XCUITest é executado em um processo separado, com os resultados obtidos pelos críterios CT13, CT14, CT15 foi possível avaliar os impactos positivos e negativos deste quesito.

Como vantagem de possuir um processo separado, XCUITest conseguiu cumprir todos critérios citados acima.

Foi possível inicializar e finalizar a aplicação através dos métodos launch() e terminate() da classe XCUIApplication, pelo código:

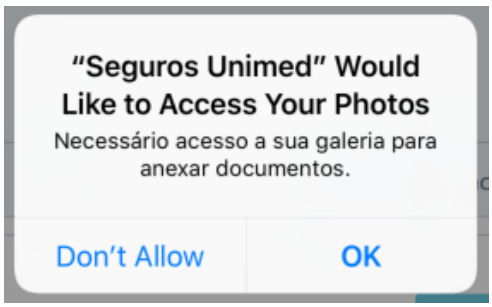
```
let app = XCUIApplication()
app.launch()
app.terminate()
```

EarlGrey e KIF não conseguem fazer o mesmo durante os testes, portanto ao executar um conjunto de casos de testes, o próximo teste será executado a partir do estado deixado pelo teste anterior, sem a possibilidade de reiniciar o aplicativo entre os casos de testes.

No Caso de Teste 4, para anexar documentos no aplicativo, o sistema pergunta para o usuário por permissões de acesso aos arquivos internos do aparelho, através de um *dialog*, mostrado na Figura 16. Como este *dialog* é gerenciado pelo sistema operacional e não pela aplicação, Earlgrey não foi capaz de confirmar a permissão, o KIF por sua vez, possui uma função que contorna esta situação e e confirma a permissão no *dialog*, através do código:

```
KIF().acknowledgeSystemAlert()
```

Figura 16 – *Dialog* do sistema exibido para pedir permissão para acessar fotos do usuário



Fonte: Elaborada pelo autor.

Este recurso do framework KIF foi satisfatório para realizar esta etapa do caso de teste, no entanto não é possível detectar o texto e os botões presentes no *dialog*, ele apenas será confirmado clicando no botão "OK", que é o suficiente para maioria dos casos, porém em situações em que o teste necessite verificar textos ou negar o acesso, não será possível pelo framework KIF.

Portanto somente com XCUITest foi possível interagir com o *dialog* do sistema por completo, através do código abaixo, é possível realizar as interações sem limitações, como em qualquer outro elemento:

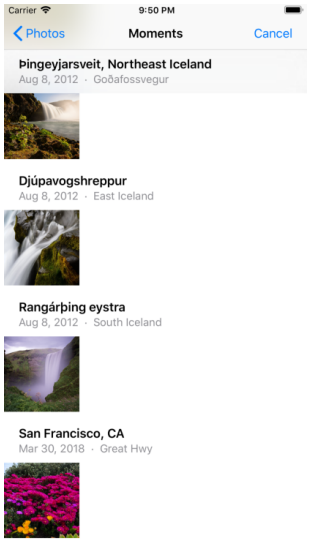
```
addUIInterruptionMonitor(withDescription: "System Dialog") {
    alert -> Bool in
     let okButton = alert.buttons["OK"]
     if okButton.exists {
          okButton.tap()
     }
}
```

Ainda no Caso de Teste 4, ao clicar no botão "Anexar"na tela de *Feedback* (Figura 8), é exibido um menu de opções para escolher a origem dos arquivos, como mostra a Figura 17. Ao selecionar "Escolher uma foto"então é aberto uma tela com a galeria de fotos do sistema como da Figura 18.

Tanto o menu quanto a galeria de fotos do sistema são janelas externas ao aplicativo, devido a recursos de segurança presentes no iOS. Como sistema operacional é responsável por exibir estes elementos, o EarlGrey e KIF não foram capaz de realizar nenhum tipo de ação nestas janelas, impossibilitando a conclusão do Caso de Teste 4. XCUITest interagiu normalmente com ambas janelas.

Figura 17 – Tela do Feedback após clicar no botão "Anexar", com menu aberto

Figura 18 – Tela da galeria de fotos do sistema, exibida ao clicar em "Escolher uma foto"



Fonte: Elaborada pelo autor.

Portanto foi visto que quando os testes são executados no mesmo processo da aplicação, *dialogs* e janelas externas são um fator limitante. Por outro lado, isto permite que EarlGrey

e o KIF acessem o estado interno do aplicativo, como variáveis, classes e fuções, durante a execução dos testes. Este tipo de acesso permite que o EarlGrey e KIF realizem testes do tipo Caixa-Branca. Como o objetivo deste estudo é a realização de testes de interface do usuário (Caixa-Preta), as possibilidades e possíveis vantagens desta característica não foram exploradas.

4.3 Considerações Finais

O experimento realizado neste estudo permitiu extrair informações sobre todos os critérios de avaliação propostos, além disso, também foi identificado aspectos relevantes sobre os frameworks que não faziam parte dos critérios. Com base nos conhecimentos adquiridos e considerando a opinião do autor, é possível fazer as seguintes afirmações sobre os frameworks:

- XCUITest possui o mais simples e rápido processo de configuração inicial.
- XCUITest possui a documentação mais completa e maior facilidade em encontrar conteúdo informativo atualizado na internet.
- KIF é o framework com menos recursos para depuração.
- EarlGrey realiza as interações com elementos de maneira mais semelhante a um usuário real.
- XCUITest é o único framework que possui recurso de gravação para escrita dos testes.
- KIF é o framework mais rápido em termos de tempo de execução.
- EarlGrey possui maior flexibilidade para realizar buscas por elementos.
- KIF possui métodos simples e objetivos, mas poucos flexíveis para interagir com elementos.
- Todas interações com elementos no KIF dependem das propriedades de acessibilidade (*accessibility identifier* e *acessibility label*) dos elementos, enquanto XCUITest e EarlGrey possuem alternativas de busca por elementos que não necessariamente dependem destas propriedades (como buscas pelo tipo do elemento).
- KIF e EarlGrey não são capazes de interagir com telas externas ao aplicativo
- EarlGrey garante maior confiabilidade nos testes quando a aplicação faz requisições de rede e possui animações, pois possui recursos de sincronização que evitam a necessidade de definir tempos de espera nos testes.
- EarlGrey é o único framework capaz de checar pela visibilidade de um elemento.

Capítulo 5

CONCLUSÃO

5.1 Contribuições

Até onde se sabe, este é o primeiro estudo comparativo que inclui os 3 frameworks presentes neste trabalho: XCUITest, EarlGrey e KIF. A maioria dos estudos similares, pelo conhecimento do autor, são focados na plataforma Android ou ferramentas multiplataforma como o Appium[1]. Portanto, o foco no desenvolvimento iOS deste trabalho permitiu avaliar os frameworks de forma mais específica. Este trabalho também apresenta detalhes técnicos que podem auxiliar outros desenvolvedores que pretendem utilizar algum dos frameworks investigados.

A condução do estudo experimental, tanto em relação à escolha dos critérios de avaliação, como também a escolha do aplicativo e dos casos de testes, se mostrou bem sucedido no seu propósito de encontrar características distintas entre os frameworks, uma vez que foi possível apontar vantagens e desvantagens de cada um. Assim este trabalho pode servir como um guia para a escolha do framework ideal de acordo com as necessidades do desenvolvedor.

Uma contribuição importante deste trabalho é que, pela forma que os casos de teste foram estruturados, mostrou-se que é totalmente viável utilizar mais de um framework ao mesmo tempo, no mesmo projeto. Visto que os 3 frameworks foram integrados no projeto do aplicativo, para realização do experimento, de maneira satisfatória. Desta forma o testador pode aproveitar os benefícios de cada framework em partes diferentes do aplicativo. Por exemplo, algumas funcionalidades podem ser testadas com um framework, e outras funcionalidades com outro.

Como contribuição pessoal, este trabalho possibilitou ao autor ampliar os conhecimentos sobre o tópico de testes de *software*, e testes automatizados de interface, principalemente para aplicações iOS. Além disso foi possível aprender, de maneira prática, sobre três ferramentas de automatização de testes de UI. Os conhecimentos adquiridos neste trabalho serão utilizados na carreira profissional do autor.

Outra contribuição importante é que este trabalho serviu como uma iniciativa para o início do processo de automatização dos testes de UI na empresa Tokenlab. Desta forma, será dado continuidade aos testes desenvolvidos neste trabalho, para incluir novas funcionalidades do aplicativo. Espera-se que com a automatização dos testes de UI, a responsabilidade pelos testes

[1] http://appium.io/

seja dividida entre a equipe de qualidade (QA) e os desenvolvedores, diminuindo a sobrecarga do QA, que necessita testar duas plataformas.

REFERÊNCIAS

BARTLEY, M. Improved time to market through au tomated software testing. 2008. Citado na página 16.

DAWSON, M.; BURRELL, D.; RAHIM, E.; BREWSTER, S. Integrating software assurance into the software development life cycle (sdlc). **Journal of Information Systems Technology and Planning**, v. 3, p. 49–53, 01 2010. Citado na página 13.

DO, S.; SOUZA, S.; MALDONADO, J.; PINTO, S.; FABBRI, F.; AURI, M.; VINCENZI, A.; BARBOSA, E.; DELAMARO, M.; JINO, M. **INTRODUÇÃO AO TESTE DE SOFTWARE**. [S.l.: s.n.], 2000. Citado na página 15.

EARLGREY. 2019. Disponível em: <https://github.com/google/EarlGrey>. Acesso em: 28/10/2019. Citado na página 19.

EARLGREY Syncronization. Disponível em: <https://github.com/google/EarlGrey/blob/master/docs/api.md#synchronization-apis>. Acesso em: 28/10/2019. Citado na página 46.

HAO B. LIU, S. N. W. G. J. H. S.; GOVINDA, R. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. 2014. Citado na página 22.

INTRODUçãO aos testes automatizados. 2019. Disponível em: <http://talkingabouttesting.coursify.me/>. Acesso em: 28/10/2019. Citado na página 11.

ISTQB. Certified tester, foundation level syllabus. 01 2018. Citado 2 vezes nas páginas 14 e 15.

JORGENSEN, P. C. **Software Testing: A Craftman's Approach**. [S.l.: s.n.], 2013. Citado na página 13.

KIF. 2019. Disponível em: <https://github.com/kif-framework/KIF>. Acesso em: 28/10/2019. Citado na página 20.

MEILIANAA IRWANDHI SEPTIANA, R. S. A. D. Comparison analysis of android gui testing frameworks by using an experimental study. 2018. Citado 2 vezes nas páginas 12 e 23.

MIXPANEL. 2018. Disponível em: <https://mixpanel.com/trends/#report/iphone_models>. Acesso em: 28/10/2019. Citado 2 vezes nas páginas 16 e 17.

MYERS, G. J. **The Art of Software Testing**. [S.l.: s.n.], 2004. Citado na página 13.

SINAGA, A. M.; ADIWIBOWO, P.; SILALAHI, A.; YOLANDA, N. Performance of automation testing tools for android applications. **2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)**, p. 534–539, 2018. Citado na página 23.

STATISTA, Smartphone users worldwide 2016-2021. 2019. Disponível em: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>. Acesso em: 28/10/2019. Citado na página 11.

TRAVASSOS, G. H. Introdução à engenharia de software experimental. 2002. Citado na página 12.

XCTEST. Disponível em: <https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/04-writing_tests.html>. Acesso em: 28/10/2019. Citado na página 18.

XCUITEST. 2019. Disponível em: <https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html>. Acesso em: 28/10/2019. Citado na página 18.