UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DIRETORIA DE GRADUAÇÃO E EDUCAÇÃO PROFISSIONAL
DEPARTAMENTO DE COMPUTAÇÃO
ENGENHARIA DE SOFTWARE

LUCAS FERNANDES COELHO

UM ESTUDO EXPLORATÓRIO COM TESTES AUTOMATIZADOS EM MÚLTIPLAS VERSÕES DE APLICAÇÕES MÓVEIS DESENVOLVIDAS USANDO O FRAMEWORK XAMARIN

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2017

LUCAS FERNANDES COELHO

UM ESTUDO EXPLORATÓRIO COM TESTES AUTOMATIZADOS EM MÚLTIPLAS VERSÕES DE APLICAÇÕES MÓVEIS DESENVOLVIDAS USANDO O FRAMEWORK XAMARIN

Trabalho de Conclusão de Curso apresentada na Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Bacharel em 2017

Orientador: Prof. Doutor André Takeshi Endo

CORNÉLIO PROCÓPIO

2017

RESUMO

COELHO, Lucas Fernandes. Um estudo exploratório com testes automatizados em múltiplas versões de aplicações móveis desenvolvidas usando o framework Xamarin. 27 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2017.

Atualmente tem crescido o interesse das empresas por *frameworks* de desenvolvimento *mobile cross-platform*. Contudo realizar testes automatizados com esses *frameworks* pode ser trabalhoso e cansativo, tendo em vista que algumas vezes é necessário escrever vários casos de teste para cada tipo de dispositivo. Para sanar este problema foi desenvolvida a ferramenta x-PATeSCO. Este trabalho estudará a eficácia dos *scripts* de teste gerados pela ferramenta ao considerar versões diferentes de uma mesma *app*.

Palavras-chave: Teste de software, Appium, Xamarin

SUMÁRIO

1 INTRODUÇÃO

Atualmente os dispositivos móveis são parte da vida das pessoas e estão disponíveis nos mais diferentes formatos, tais como os *tablets, smartphones e wearables*. Esses dispositivos portáteis possuem alta capacidade de processamento, espaço de armazenamento e uma variedade diversificada de sensores (BOUSHEHRINEJADMORADI et al., 2015). Eles são controlados por um sistema operacional (SO). Em uma pesquisa realizada pela International Data Corporation (IDC, 2017) sobre o Market Share dos SOs por dispositivo, no primeiro trimestre de 2017 os sistemas mais utilizados foram o Android (GOOGLE, 2017a) com 85% e Apple iOS (APPLE, 2017b) com 14%. Cada SO possui uma loja de aplicativos (*app stores*) onde os desenvolvedores podem submeter suas *apps* e os usuários fazem download e instalam em seus dispositivos. Em uma análise destas *app stores* feita pela appfigures (CHANTELLE, 2017), a quantidade de *apps* ativas em 2016 na loja oficial do Android, a Google Play (GOOGLE, 2017c). Contava com aproximadamente 2,81 milhões de *apps* e a iOS App Store (APPLE, 2017a), da Apple, tinha algo em torno de 2,26 milhões.

Com a crescente evolução dos sistemas computacionais, tem-se uma preocupação com a qualidade desses sistemas desenvolvidos, tanto na sua construção quanto no resultado, a utilização pelo usuário final (MALDONADO et al., 2004). A Engenharia de Software tem buscado formas de se produzir sistemas obtendo um resultado de maior qualidade e com baixo custo (PRESSMAN, 2009). Teste de software é uma das abordagens mais praticadas para avaliar a qualidade dos sistemas computacionais e é um dos temas mais pesquisados na engenharia de software (ORSO; ROTHERMEL, 2014). No desenvolvimento de *apps* é preciso que haja um processo rigoroso de testes, já que caso algum defeito seja encontrado apenas quando a *app* já está nas *app stores*, isso pode influenciar a opinião do usuário sobre a *app*, classificando-a negativamente. Para fazer a correção, o time de desenvolvimento terá que encontrar o defeito e corrigi-lo, o que leva mais tempo e gera mais custo, e o usuário precisará baixar a nova versão. Contudo não há formas de obrigar o usuário a realizar essa atualização, quando ela estiver disponível (PRESSMAN, 2009).

1.1 MOTIVAÇÃO

No cenário de desenvolvimento de software convencional o time estuda um problema, encontra uma solução e em seguida realiza a implementação da funcionalidade (BERNARDO; KON, 2008). Somente depois de todo o processo é que o desenvolvedor realizará os testes no software.

Testes manuais necessariamente precisam de uma ou mais pessoas para testar manualmente o que foi desenvolvido pela equipe. A execução desses testes tem como objetivo encontrar falhas no sistema durante o processo de desenvolvimento, contudo é uma tarefa cansativa e que demanda muito tempo dos testadores. Com isso, é muito comum que não sejam verificados novamente todos os casos de teste a cada mudança no código. Isto pode gerar muitos erros no software, podendo atrasar a entrega do produto final ou no pior caso, o defeito pode ser encontrado pelo usuário quando a *app* já está em produção. Todos esses problemas geram custos para a empresa e conforme a quantidade de desenvolvedores na empresa cresce, o número de testadores deve acompanhar este crescimento.

Testes automatizados são programas ou *scripts* que têm a finalidade de testar as funcionalidades criadas. A vantagem de se automatizar os testes é a possibilidade de verificar tudo o que já foi desenvolvido rapidamente e a qualquer momento. Os testes automatizados, quando bem aplicados pela equipe de desenvolvimento são uma excelente ferramenta para minimizar a quantidade de defeitos, uma vez que, quando o desenvolvedor produz uma funcionalidade ela deve ter um caso de teste automatizado correspondente. Com isso, se a equipe decidir incluir novas funcionalidades que alterem a estrutura dos códigos que já tinham sido desenvolvidos, basta que o testador execute novamente todos os *scripts* e verifique se algum deles parou de funcionar. Além disso, como os casos de teste são códigos que serão interpretados por máquinas, o testador pode incluir diversas combinações de valores que um ser humano realizando testes manuais provavelmente não testaria.

Para este estudo será utilizada a ferramenta x-PATeSCO (*cross-Platform App Test Script reCOrder*), desenvolvida por Menegassi e Endo (2017). Baseada no *framework* Appium (JSFOUNDATION, 2017), esta ferramenta tem como objetivo principal realizar testes automatizados de *apps*. Com a constante evolução das *apps* de acordo com as novas necessidades ou tenologias faz-se necessário que os desenvolvedores atualizem as *apps* frequentemente. Com isso os *scripts* de teste criados para uma versão anterior da *app* podem falhar. Contudo muitas das vezes o problema surge apenas porque foi preciso mudar um controle da tela de lugar ou algum código identificador de algum controle foi alterado. Isso acaba sendo um grande problema

para os testadores, já que, isso não é um defeito no código ou falha na implementação e mesmo assim eles precisam verificar todos os *scripts* procurando por essas alterações, tomando tempo de outras atividades. Esse problema é particularmente importante se as apps forem multiplataformas.

As *apps* que serão testadas foram desenvolvidas utilizando o Xamarin, um *framework cross-platform* de desenvolvimento *mobile*. Utilizando o Xamarin é possível criar *apps* para as plataformas Android, iOS e *Universal Windows Platform*, ou simplesmente UWP (MICROSOFT, 2017d), utilizando uma única base de código. Toda a lógica de aplicação é escrita utilizando C# (C#, 2017) e XAML (XAML, 2017), uma linguagem declarativa baseada no XML. Quando a *app* for compilada, o *framework* transformará todo o código escrito na linguagem nativa da plataforma selecionada. Isso demanda maior versatilidade nos testes, sendo necessário testar a mesma *app* em vários dispositivos com diferentes configurações e SOs. As *apps* multiplataforma são desenvolvidas com um *framework* específico e utilizando uma única base de código é capaz de distribuir a mesma *app* em diversas plataformas.

1.2 OBJETIVOS

O objetivo deste trabalho é verificar se os *scripts* de teste gerados com a ferramenta x-PATeSCO são robustos, considerando diferentes versões de uma mesma *app*. Para isso, os testes serão gerados com base em três *apps* desenvolvidas usando o Xamarin e disponibilizadas por uma empresa de TI.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho foi dividido da seguinte forma: o Capítulo 2 apresenta o embasamento teórico necessário para ambientação do tema e está subdividido nos assuntos: testes de software, aplicações móveis, a apresentação do *framework* Xamarin com um projeto de exemplo, a apresentação do *framework* Appium e a ferramenta x-PATeSCO, bem como os trabalhos relacionados. O Capítulo 3 apresenta a proposta de trabalho, explicando como será conduzido, e o cronograma de execução das atividades propostas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado da seguinte forma: na Seção 2.1 será explicado o conceito de testes de software. Na Seção 2.2 serão explicadas as diferenças das plataformas móveis e as formas de desenvolver *apps*. Na Seção 2.3 será explicado com mais detalhes o *framework* Xamarin. Na Seção 2.4 será apresentado o *framework* Appium e a ferramenta x-PATeSCO. Na Seção 2.5 serão apresentados os trabalhos relacionados.

2.1 CONCEITOS DE TESTE DE SOFTWARE

O processo de teste de um software pode ser dividido em quatro etapas: planejamento dos testes, projetar os casos de testes, execução e avaliação dos resultados (MYERS, 1979; BEIZER, 1990; MALDONADO et al., 1991; PRESSMAN, 2009). Essas quatro etapas fazem parte do processo de desenvolvimento do software e podem ser aplicados em três níveis: o teste de unidade, de integração e de sistema. No primeiro nível, o teste de unidade, os desenvolvedores concentram-se na menor parte do sistema separadamente, a unidade, que no caso de uma linguagem de programação que tenha o paradigma de Orientação a Objetos, a menor parte pode ser um método. O teste de integração visa identificar problemas na integração dessas unidades, testando a interdependência dos métodos e interfaces. No teste de sistema, verifica-se a operação do sistema como um todo, verificando se há algum defeito na criação das interfaces ou se houve algum erro na especificação anteriormente validada com o cliente (PRESSMAN, 2009).

Existem várias técnicas e critérios para realizar testes e elas podem ser divididas em dois grupos: a técnica funcional e técnica estrutural. Na funcional que também é conhecida como teste caixa preta (MYERS, 1979), isso se dá pelo fato de que o sistema é observado como se fosse uma caixa fechada, podendo-se observar apenas os lados de fora. Nesse caso, o testador insere dados e valida os resultados obtidos com o que foi especificado, não se preocupando com os detalhes da implementação daquela funcionalidade. Portanto é fundamental que a especificação tenha sido bem elaborada. Alguns exemplos de critérios de teste funcional

são: Particionamento em Classes de Equivalência, Análise do Valor Limite e Grafo de Causa-Efeito (PRESSMAN, 2009). Já a técnica estrutural, conhecida como caixa branca (PRESSMAN, 2009), vem para complementar à técnica funcional, já que o desenvolvedor tem total acesso ao código, utilizando-o para elaborar os testes de acordo com a própria lógica interna do programa. Alguns exemplos de critérios de teste estrutural são: *Line Coverage* e *Branch Coverage*.

2.2 APLICAÇÕES MÓVEIS

Existem várias formas para desenvolver *apps* para as plataformas móveis e que podem ser classificadas em três grupos: aplicações nativas, aplicações Web e aplicações híbridas (MALDONADO et al., 2004). Aplicações nativas são desenvolvidas para uma plataforma específica. A forma de desenvolvimento é definida pala organização proprietária. Esta empresa fornece para os desenvolvedores o *Software Development Kit* (SDK) e uma *Integrated Development Environment* (IDE), que é o conjunto de ferramentas necessárias para desenvolver uma *app* para a aquela plataforma. Ao optar por uma aplicação nativa, o desenvolvedor tem acesso a todas as *Application Programming Interface* (API) para aquele SO sem restrições. A desvantagem de desenvolver nativamente é que a aplicação fica restrita a uma única plataforma, tendo que reescrever todo o programa caso seja necessário utilizar a mesma *app* em outras plataformas. Aplicações Web são desenvolvidas utilizando as mesmas tecnologias que são utilizadas na Web, como HTML5 (HTML5, 2017), CSS3 (CSS3, 2017) e JavaScript (ECMA, 2017). Essas aplicações não são instaladas no dispositivo, executam num servidor Web e são desenvolvidas de forma que se adaptem as diferentes resoluções dos computadores, *smartphones* e *tablets*. Como são executadas em um navegador Web não possuem um acesso tão avançado as APIs nativas dos dispositivos.

Aplicações híbridas, também chamadas de *cross-platform* são desenvolvidas utilizando *frameworks* que abstraem todo o modo de criação de *apps*. Utilizando apenas o que o *framework* fornece é possível criar aplicações para os principais SOs do mercado. Essas aplicações híbridas são divididas em duas amplas classes. A primeira que é chamada de *Web-based framework* (BOUSHEHRINEJADMORADI et al., 2015), onde permitem que os desenvolvedores utilizem as linguagens Web, como HTML5, CSS3 e JavaScript. Exemplos de *Web-based frameworks* são o Adobe PhoneGap (ADOBE, 2017), Apache Cordova (APACHE, 2017), IBM MobileFirst (IBM, 2017), IONIC (IONIC, 2017). Os desenvolvedores utilizam as linguagens Web para criar a lógica de aplicação e a interface do usuário. Contudo essas linguagens não suportam totalmente todos os recursos dos dispositivos como câmera, microfone, agenda de contatos e

preferências do telefone. Portanto para utilizar essas funcionalidades os *frameworks* fornecem bibliotecas que são utilizadas em tempo de execução para acessar os recursos de hardware, por isso são popularmente chamados de aplicações híbridas. Entretanto com esses *frameworks* não é possível criar aplicações de alto desempenho como jogos ou algo que utilize animação. Outro problema é que toda a interface é feita nas linguagens da Web, então alguns controles que ainda não estejam implementados podem aparecer diferentes dos controles nativos, ficando claro para o usuário que aquela *app* não foi desenvolvida de forma nativa.

A segunda classe é chamada de *native framework*. Alguns exemplos de *frameworks* são o Facebook React Native (FACEBOOK, 2017), Apportable (APPORTABLE, 2017), MyApp-Converter (MYAPPCONVERTER, 2017) e o Xamarin (XAMARIN, 2017a). A diferença para os *frameworks* mostrados acima é que os *native frameworks* utilizam tudo do SO nativamente. Toda a construção da *app* é feita em uma linguagem e ao ser compilado para outra plataforma alvo o *framework* consome todas as APIs nativas. Por exemplo ao escrever uma aplicação em Xamarin os desenvolvedores utilizam a linguagem C# para criar a lógica de negócio e XAML para construir a interface de usuário. No processo de compilação todo o código XAML será interpretado e renderizado de acordo com a plataforma e versão de SO selecionados, invocando os controles nativos.

2.3 XAMARIN

Xamarin é um *framework* de desenvolvimento *cross-platform* para dispositivos móveis e mantido pela Microsoft (MICROSOFT, 2017a). O *framework* Xamarin possui três técnicas diferentes para criar aplicações móveis: Xamarin.iOS (XAMARIN.IOS, 2017), Xamarin.Android (XAMARIN.ANDROID, 2017) e Xamarin.Forms (XAMARIN.FORMS, 2017).

2.3.1 XAMARIN.IOS E XAMARIN.ANDROID

As tecnologias Xamarin.iOS e Xamarin.Android foram baseadas no Mono (MONO, 2017), uma versão de software livre do .NET Framework (.NETFRAMEWORK, 2017). As *apps* feitas em uma dessas duas técnicas, ficam limitadas a plataforma escolhida. Por exemplo, caso o desenvolvedor escolha o Xamarin.Android, nenhuma tela que foi criada para interação do usuário será reaproveitada quando o time de desenvolvimento decidir fazer a mesma *app* para iOS, obrigando os desenvolvedores a recriarem toda a interface e lógica de apresentação, reaproveitado apenas a lógica de aplicação.

Na Figura 5, é possível notar que todo o código da lógica da aplicação é escrito em

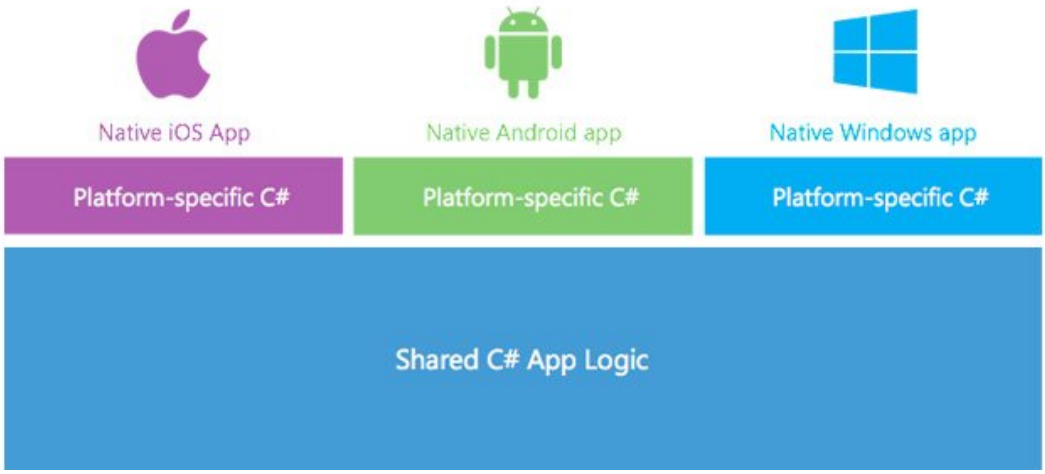


Figura 5: Arquitetura de desenvolvimento de aplicativos Xamarin.Android e Xamarin.iOS. Fonte: (ROSE, 2016)

C# e utilizado em todas as plataformas (Shared C# App Logic). Contudo, para cada uma das plataformas será necessário recriar a lógica de apresentação (Plataform-specific C#). As *apps* são escritas na linguagem C# e compiladas utilizando uma versão específica da DLL (Dynamic-link Library) de acordo com a plataforma. No caso, MonoTouch.dll (XAMARIN.IOS, 2017) para iOS e MonoAndroid.dll (XAMARIN.ANDROID, 2017) para Android. Após o processo de compilação o resultado é um pacote de aplicação idêntico aos que são feitos nas IDEs padrão, sendo impossível distinguir um *.apk* feito no Android Studio (GOOGLE, 2017b) e outro feito em Xamarin.

2.3.2 XAMARIN.FORMS

O Xamarin.Forms é uma abstração da forma de criar aplicações móveis. Utilizando essa tecnologia é possível escrever um único código que será interpretado e compilado individualmente em cada plataforma. As interfaces de usuário são renderizadas e transformadas em controles nativos. Diferente do Xamarin.Android e Xamarin.iOS que apenas a lógica de aplicação é compartilhada, no Xamarin.Forms tanto a interface quanto a própria lógica de aplicação são escritas uma única vez. Quando uma *app* é compilada, ela utiliza a API nativa de cada plataforma. O Xamarin.Forms visa trazer agilidade para os times de desenvolvimento, sendo necessário que todos conheçam apenas uma linguagem de programação. Além disso, caso tenha algum problema ou o time decida fazer alguma melhoria, basta atualizar apenas uma

base de código. Como é mostrado na Figura 6, tanto a lógica de aplicação (Shared C# App Logic) quanto a lógica de apresentação e interface (Shared C# User Interface Code).

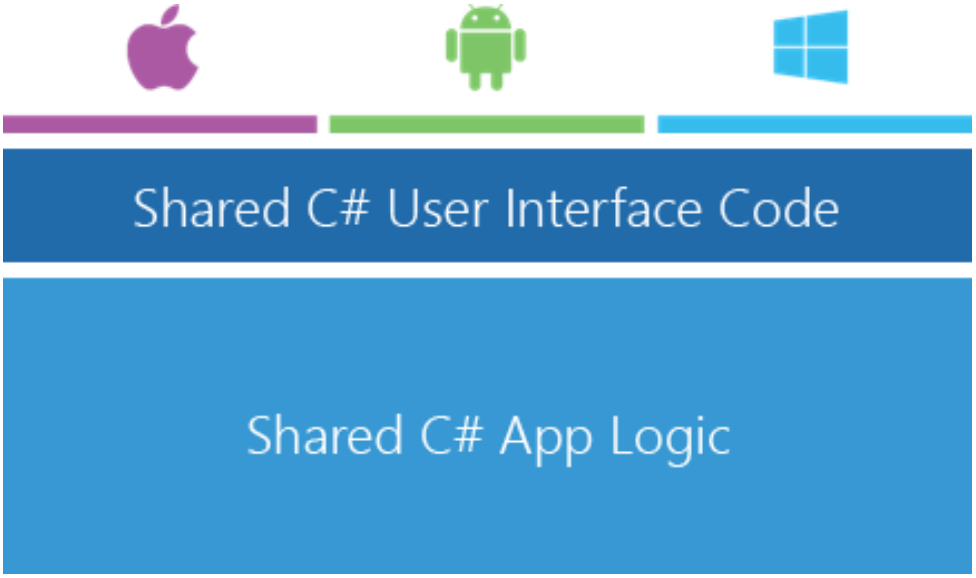


Figura 6: Arquitetura de desenvolvimento de aplicativos Xamarin.Forms.
Fonte: (NICOLI, 2017)

Para exemplificar a utilização do Xamarin.Forms será utilizado um projeto de cálculo de Índice de Massa Corporal (IMC). O IMC é o resultado de uma fórmula matemática que indica como a sua saúde em relação a sua massa corporal, contudo esse índice é apenas um ponto de partida, já que o IMC não define exatamente o seu estado nutricional. A fórmula para realizar o cálculo é: IMC = P (peso em quilos)/A^2 (altura x altura, em metros) (SAÚDE, 2017). Lembrando que o IMC somente é válido para pessoas adultas e com idades entre 20 a 59 anos. A aplicação desenvolvida é composta de apenas uma tela que possuem duas entradas de dados, a primeira é a altura que deve ser inserida pela medida de uma pessoa adulta, e em metros. A próxima entrada é o peso que deve ser informado em quilogramas. Os dois controles aceitam somente números que podem conter casas decimais. Logo abaixo existe o botão CALCULAR, que chamará o método da fórmula do cálculo e passará os dois valores como parâmetro. Caso nenhum dos campos tenham valores em branco, nulos ou zeros, ao clicar no botão a *app* exibirá o resultado do cálculo e um texto informando em qual das faixas de peso a pessoa está enquadrada. Os possíveis resultados são:

Baixo peso: Caso o resultado seja menor que 18,5.

Peso adequado: Caso o resultado seja maior ou igual a 18,5 e menor que 25.

Sobrepeso: Caso o resultado seja maior ou igual a 25 e menor que 30.

Obesidade: Caso o resultado seja maior ou igual a 30.

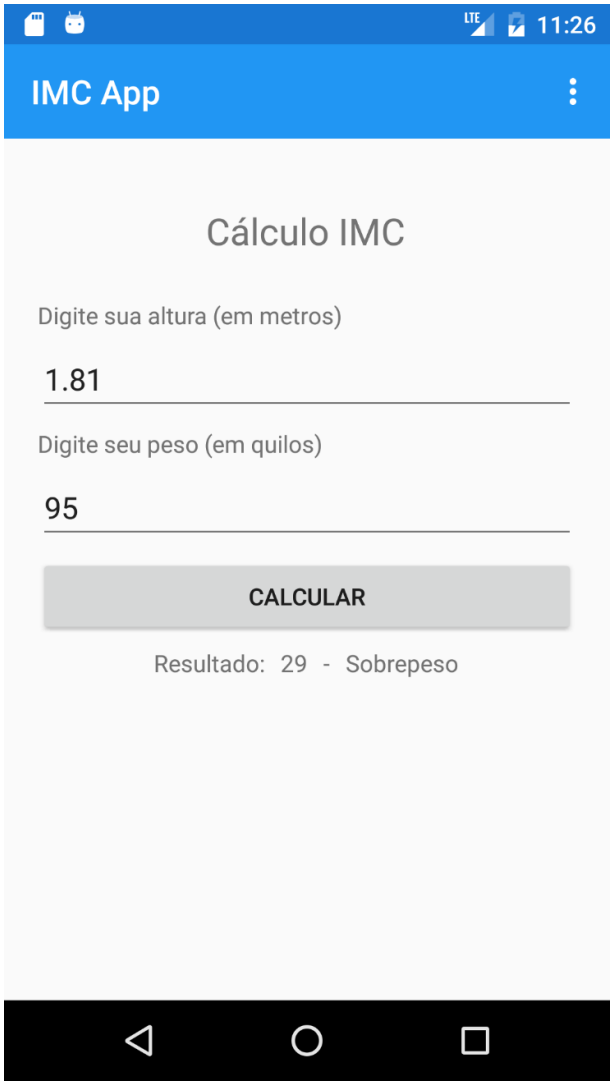


Figura 7: Aplicativo exemplo –Tela inicial.

Fonte: Autoria própria

2.3.3 ESTRUTURA DO PROJETO

As IDEs necessárias para desenvolver uma aplicação Xamarin são o Xamarin Studio (XAMARIN, 2017b) no Apple Mac OS (APPLE, 2017b) e o Visual Studio (MICROSOFT, 2017g) no Microsoft Windows (MICROSOFT, 2017b). Para desenvolver este exemplo, foi utilizado um computador com Microsoft Windows 10 e o Visual Studio na versão 2017.

Um projeto Xamarin.Forms é um conjunto de projetos, denominado *Solution* e é composto por no mínimo quatro projetos: Um Portable, um Android, um iOS e outro UWP. Todo o código será escrito no projeto Portable e quando for executar a saída será necessário compilar cada projeto individualmente para a plataforma desejada. Caso seja necessário utilizar algum recurso muito específico de alguma das plataformas será necessário criar um código dentro do

projeto de cada uma delas, já que o que não estiver na Portable não será compartilhado com os outros projetos.

O projeto foi desenvolvido utilizando o padrão Model View ViewModel (MVVM) (MICROSOFT, 2017c), criado pela Microsoft e era anteriormente utilizado pelo WPF (MICROSOFT, 2017h) e Silverlight (MICROSOFT, 2017e). Este padrão visa separar as responsabilidades dos objetos. A View é a interface do usuário e a sua única função é exibir os controles, como botões ou textos. A Model é onde fica toda a lógica de negócio e os dados, as classes de acesso ao banco de dados geralmente estão nas Models. A ViewModel faz a ligação entre as Views e as Models, afinal a View e a Model não estão relacionadas diretamente. Na ViewModel é programada toda a lógica de apresentação. A comunicação com a View é feita através dos *databindings*. Ao escrever o XAML da View, pode-se associar um controle da interface à uma propriedade da ViewModel. Com o *databinding*, a vantagem disso é que qualquer alteração que for feita em qualquer um dos arquivos, seja por interação do usuário na interface ou por algum cálculo feito internamente na ViewModel, a propriedade será notificada e alterada, não sendo necessário utilizar eventos para monitorar as ações do usuário. Caso o time de desenvolvimento seja bem separado e bem definido, ainda é possível que a View e a ViewModel sejam programadas por pessoas ou times diferentes, um deles cuidando apenas do XAML da View e outro criando os métodos na classe C# da ViewModel, basta que apenas o time que criará a View descreva em detalhes o que será exibido ao usuário e qual o seu comportamento. Uma outra vantagem de ter toda a lógica de apresentação na ViewModel é que isso torna o código completamente testável, uma vez que uma ViewModel nada mais é do que uma simples classe e totalmente compatível com qualquer framework de teste de unidade que suportem a linguagem C#.

É uma boa prática de programação dividir as responsabilidades em pastas, a título de organização. Como é possível ver na Figura 8 as ViewModels do projeto são classes C# e cada View é um arquivo com extensão .XAML e que possui também uma classe C# relacionada a View, chamada de *code-behind*. No *code-behind* também é possível escrever toda a lógica de apresentação, contudo isso cria um acoplamento muito forte entre a View e o seu respectivo *code-behind*, caso em algum momento o time de desenvolvimento precise remover uma View ou alterar a sua ordem ou lógica de apresentação, será necessário refatorar todo o código, procurando por dependências. Portanto, como a ViewModel possui um baixo acoplamento com as Views, podem ser separadas a qualquer momento. Nesse exemplo existe a View MainPage.xaml e MainPage.xaml.cs que é o seu *code-behind*. Por questões de organização as ViewModels que contém a lógica de apresentação da View, recebem o mesmo nome e acrescenta-se o sufixo ViewModel. No exemplo a MainPage será nomeada como MainPageViewModel.cs. Outra classe

importante para o projeto é a BaseViewModel; essa classe tem a abstração de alguns métodos e atributos que serão utilizados em todas as outras ViewModels.

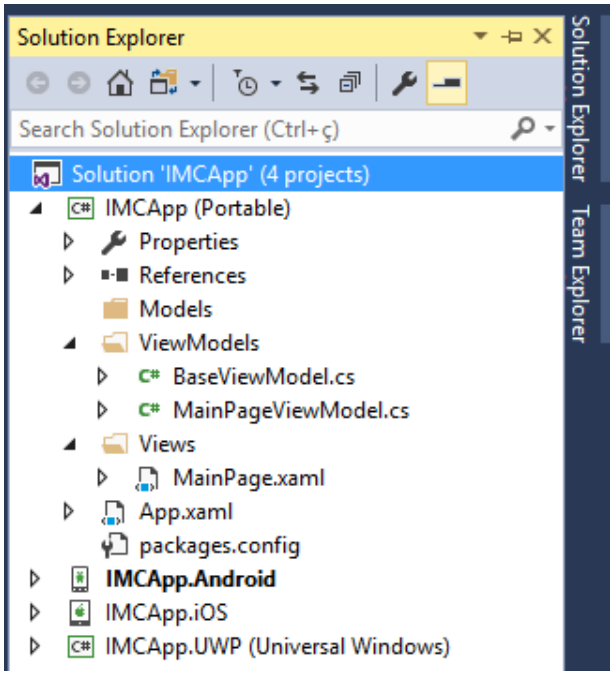


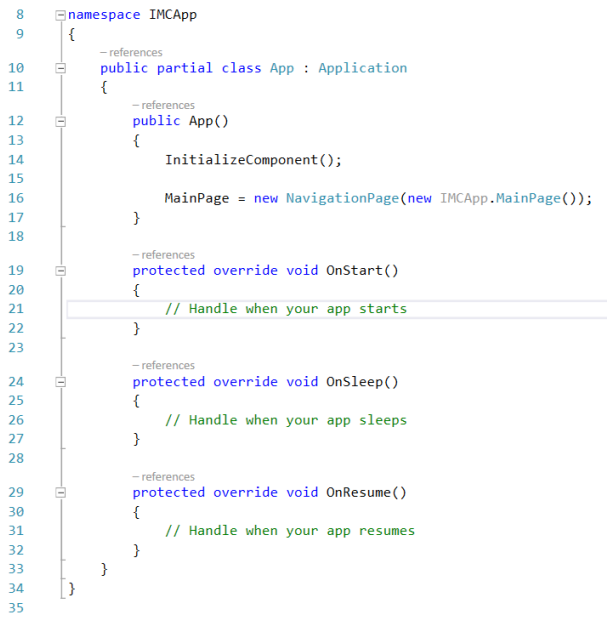
Figura 8: Estrutura de arquivos do projeto.

Fonte: Autoria própria

O arquivo App.xaml apesar de ter as mesmas propriedades de uma View, como a extensão XAML e o code-behind, ela é apenas um objeto que extrai os métodos de ciclo de vida das *apps*, como é possível ver na Figura 9.

Através dos métodos da classe App.xaml.cs é possível controlar os comportamentos de ciclo de vida da aplicação em cada uma das plataformas. Caso seja inserida alguma lógica de apresentação no método *OnStart()*, está será a primeira instrução a ser executada assim que a *app* for iniciada. No *OnSleep()* a programação só terá efeito quando a *app* ficar em segundo plano, por exemplo, quando o usuário navegar por outro aplicativo. Já o *OnResume()* só será executado quando a *app* voltar a ficar ativo, em primeiro plano.

A aplicação é iniciada na classe App.xaml.cs, onde é instanciado o primeiro objeto, no método construtor da classe. MainPage é a primeira página da aplicação e no caso será atribuída uma NavigationPage e passando MainPage como parâmetro. Utilizando o NavigationPage é permitido fazer uso da navegação por pilha, presente em todas as plataformas mobile, onde a cada nova tela aberta o SO empilha a nova tela sobre a anterior e quando o usuário realiza a ação para voltar, ele desempilha e destrói aquela tela, podendo desempilhar até chegar a tela inicial, a MainPage.



```
8   namespace IMCApp
9   {
        – references
10      public partial class App : Application
11      {
            – references
12          public App()
13          {
14              InitializeComponent();
15
16              MainPage = new NavigationPage(new IMCApp.MainPage());
17          }
18
            – references
19          protected override void OnStart()
20          {
21              // Handle when your app starts
22          }
23
            – references
24          protected override void OnSleep()
25          {
26              // Handle when your app sleeps
27          }
28
            – references
29          protected override void OnResume()
30          {
31              // Handle when your app resumes
32          }
33      }
34  }
35
```

Figura 9: Trecho de código da classe App.xaml.cs.

Fonte: Autoria própria

Na Figura 10 é mostrado o conteúdo de um arquivo XAML. Toda a interface do usuário é escrita neste arquivo, utilizando da abstração do Xamarin.Forms ao inserir a tag <Button>, na linha 20, o Xamarin transformará em tempo de compilação essa tag em um controle, invocando a API nativa referente a um botão, de acordo com a plataforma escolhida. É possível notar que alguns controles, como nas entradas de texto das linhas 17 e 19, foi utilizado o conceito de *databindings*, presente no Xamarin.Forms. Com isso ao inserir ou alterar um valor novo no campo, como ele está ligado a uma propriedade de uma classe, não é necessário monitorar as alterações desse controle, já que todas as mudanças serão automaticamente refletidas na classe. Quase todos os atributos de uma tag podem ser definidas por *databindings*, não se limitando apenas a textos ou números. Por exemplo o controle de interface StackLayout, da linha 23, ele é responsável por agrupar elementos de interface. Ele possui uma propriedade IsVisible da qual o valor está associado a uma propriedade Booleana da classe, podendo receber *true* ou *false*, ou seja, dependendo da lógica de apresentação, o StackLayout pode ou não estar visível. Neste exemplo do IMC se os valores de altura (Height) ou peso (Weight) forem igual a zero e o usuário clicar no botão para fazer o cálculo, a propriedade IsVisible recebe o valor *false* e o elemento deixa de ser exibido na tela.

A Figura 11 contém todo o código do *code-behind* da View MainPage.xaml. Como toda a lógica de apresentação deve estar contida em uma ViewModel, a única instrução que a classe executa é informar qual é o seu BindingContext na linha 16, ou seja, qual a classe que a

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:IMCApp"
             x:Class="IMCApp.MainPage"
             Title="IMC App">
    <ContentPage.ToolbarItems>
        <ToolbarItem Order="Secondary" Text="Sair"></ToolbarItem>
    </ContentPage.ToolbarItems>

    <StackLayout Padding="20">
        <!--Título-->
        <Label Text="Cálculo IMC" Margin="20" HorizontalOptions="Center" FontSize="Large"/>

        <!--Controles-->
        <Label Text="Digite sua altura (em metros)"/>
        <Entry Keyboard="Numeric" Text="{Binding Height}"/>
        <Label Text="Digite seu peso (em quilos)"/>
        <Entry Keyboard="Numeric" Text="{Binding Weight}"/>
        <Button Text="Calcular" Command="{Binding CalculateCommand}"/>

        <!--Resultado-->
        <StackLayout HorizontalOptions="Center" IsVisible="{Binding IsVisible}" Orientation="Horizontal">
            <Label Text="Resultado: " />
            <Label Text="{Binding Result, StringFormat='{0:0.##}'}"/>
            <Label Text=" - "/>
            <Label Text="{Binding Message}"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

Figura 10: Trecho de código da View MainPage.xaml.

Fonte: Autoria própria

View deve associar as suas propriedades.

```
9   namespace IMCApp
10  {
        4 references
11      public partial class MainPage : ContentPage
12      {
            1 reference
13          public MainPage()
14          {
15              InitializeComponent();
16              BindingContext = new MainPageViewModel();
17          }
18      }
19  }
20
```

Figura 11: Trecho de código da classe MainPage.xaml.cs.

Fonte: Autoria própria

Na Figura 12 é mostrado o código da BaseViewModel.cs, uma classe que abstrai os métodos e propriedades que serão utilizados nas demais ViewModels. Para poder trabalhar com os *databindings* é necessário que seja implementada a interface INotifyPropertyChanged, na linha 11. Nessa interface temos o evento PropertyChangedEventHandler, na linha 15 que junto com o método *OnPropertyChanged()*, na linha 17, é responsável por monitorar as pro-

```
8
9   namespace IMCApp.ViewModels
10  {
        1 reference
11      public class BaseViewModel : INotifyPropertyChanged
12      {
            0 references
13          public string PageName { get; set; }
14
15          public event PropertyChangedEventHandler PropertyChanged;
16
            1 reference
17          protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
18          {
19              PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
20          }
21
            5 references
22          protected bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
23          {
24              if (EqualityComparer<T>.Default.Equals(storage, value))
25              {
26                  return false;
27              }
28
29              storage = value;
30              OnPropertyChanged(propertyName);
31              return true;
32          }
33      }
34  }
35
```

Figura 12: Trecho de código da classe BaseViewModel.cs.

Fonte: Autoria própria

priedades e informar quando elas foram alteradas. Na linha 22, o método *SetProperty()*, que deve ser incluído nos métodos setters das propriedades das ViewModels. Nele é executada a verificação para garantir que haja a mudança do valor apenas quando a propriedade for alterada, economizando processamento desnecessário do dispositivo.

No trecho de código da Figura 13 a classe MainPageViewModel.cs implementa a BaseViewModel.cs. Na definição das propriedades da classe, todas as que terão alguma alteração na exibição para o usuário devem chamar o método *SetProperty()* nos seus métodos setters. Só assim o Xamarin garante que o valor será sempre o mesmo, em qualquer um dos arquivos, sem que seja necessário criar um evento na interface que verifique se os valores foram alterados. Todas essas propriedades estão ligadas a um controle na View.

A Figura 14 exibe toda a lógica de apresentação da MainPageViewModel.cs para a MainPage.xaml. No construtor são definidos os parâmetros iniciais para a execução da aplicação. A propriedade IsVisible é iniciada com falso, ocultando o controle de layout da View, já que no início da aplicação nenhum cálculo foi executado e sendo assim não há nada a ser exibido.

O *CalculateCommand* é uma propriedade do tipo Command, presente no *framework*

```
9   namespace IMCApp.ViewModels
10  {
        2 references
11      public class MainPageViewModel : BaseViewModel
12      {
13          private double _height;
                4 references
14          public double Height
15          {
16              get { return _height; }
17              set { SetProperty(ref _height, value); }
18          }
19
20          private double _weight;
                3 references
21          public double Weight
22          {
23              get { return _weight; }
24              set { SetProperty(ref _weight, value); }
25          }
26
27          private double _result;
                7 references
28          public double Result
29          {
30              get { return _result; }
31              set { SetProperty(ref _result, value); }
32          }
33
34          private bool _isVisible;
                3 references
35          public bool IsVisible
36          {
37              get { return _isVisible; }
38              set { SetProperty(ref _isVisible, value); }
39          }
```

Figura 13: Trecho de código da classe MainPageViewModel.cs.

Fonte: Autoria própria

```
            1 reference
48          public ICommand CalculateCommand { get; private set; }
49
            1 reference
50          public MainPageViewModel()
51          {
52              IsVisible = false;
53              Height = 0;
54              Weight = 0;
55              CalculateCommand = new Command(ExecuteCommandCalculate)
56          }
57
            1 reference
58          void ExecuteCommandCalculate()
59          {
60              if (Height == 0 || Weight == 0)
61                  IsVisible = false;
62              else
63              {
64                  IsVisible = true;
65                  Result = (Weight / (Height * Height));
66                  if (Result < 18.5)
67                      Message = "Baixo peso";
68                  else if (Result >= 18.5 && Result < 25)
69                      Message = "Peso adequado";
70                  else if (Result >= 25 && Result < 30)
71                      Message = "Sobrepeso";
72                  else if (Result >= 30)
73                      Message = "Obesidade";
74              }
75          }
76      }
77  }
78
```

Figura 14: Trecho de código da classe MainPageViewModel.cs.

Fonte: Autoria própria

Xamarin.Forms, que é associado a um método o *ExecuteCommandCalculate()* e é nele onde é executado a lógica para a ação do botão CALCULAR da interface, que também está ligado a classe por *databindings*.

O método verifica primeiramente se as propriedades Height e Weight estão com valor zero, se verdadeiro, IsVisible recebe *false* e o método termina; caso contrário, o valor a ser atribuído será *true* e os valores de altura e peso serão calculados na fórmula. Para cada um dos valores existe uma saída em texto correspondente que será atribuído à propriedade Message, que está contida no controle StackLayout.

2.4 APPIUM E X-PATESCO

Appium é um *framework open source* para automatizar testes em *apps mobile*. Com ele é possível testar *apps* nativas, híbridas e Web *apps* . Além disso, ele é *cross-platform*, possibilitando realizar testes automatizados nas plataformas iOS e Android, utilizando o Selenium WebDriver API (SELENIUM, 2017). O testador utilizando o Appium consegue gerar *scripts* de teste que são gravados, gerando um projeto para o Microsoft Visual Studio, codificado em C# do qual também tem suporte para o Unit Testing Framework (MICROSOFT, 2017f). O projeto possui uma estrutura que pode ser utilizada para testar a mesma *app* em diversas configurações diferentes. Inclusive o *framework* é compatível com ambientes de teste em nuvem como o Amazon Device Farm (AMAZON, 2017), BitBar (BITBAR, 2017) e TestObject (TESTOBJECT, 2017). Estas plataformas disponibilizam uma grande quantidade de dispositivos reais dos quais é possível instalar as *apps* e validar o comportamento nas mais variadas configurações possíveis como resoluções de tela, diferentes SOs e capacidade de recursos disponíveis no dispositivo.

Como foi visto na Seção 2.2, existem várias formas de se desenvolver *apps*. Muitos *frameworks* agilizam o processo de desenvolvimento de uma *app*, possibilitando que com a mesma base de código o software esteja disponível em várias plataformas. Contudo a forma de testar continua trabalhosa, já que os testadores precisam escrever um *script* de teste diferente para cada plataforma ou mesmo quando os *scripts* apresentam problemas em diferentes configurações no mesmo SO. Com a motivação de melhorar o processo de testes automatizados os pesquisadores MENEGASSI; ENDO (2017) desenvolveram a ferramenta x-PATeSCO (*cross-Platform App Test Script reCOrder*). A ferramenta conecta-se a um servidor Appium e permite que sejam codificados casos de teste capazes de testar a mesma *app* em diferentes dispositivos, configurações e SOs. Ela tenta antecipar o trabalho do testador na revisão dos *scripts*. Muitas vezes quando uma aplicação recebe uma atualização, alguns componentes de interface são alterados ou simplesmente a sua disposição na tela é modificada. Isso para o software de

teste automatizado pode ser um problema, gerando falhas nos *scripts*. O x-PATeSCO produz um *script* que pode ser utilizado para testar a mesma *app* em várias configurações, com isso é possível encontrar inconsistências nas *apps* entre plataformas, ou até mesmo entre versões de um mesmo SO. Os detalhes do funcionamento da ferramenta podem ser vistos na Figura 15.

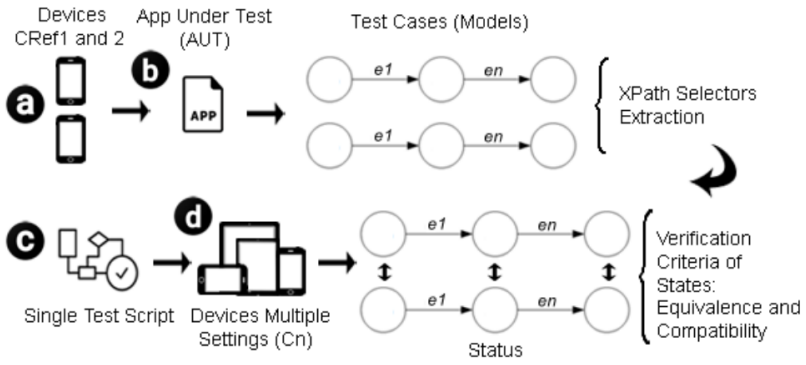


Figura 15: Funcionamento da ferramenta x-PATeSCO.
Fonte: Adaptado de (MENEGASSI; ENDO, 2017)

Para criação dos testes a ferramenta possui quatro passos, o primeiro é definir os dispositivos que serão utilizados. No passo dois será definido um modelo de caso de teste, definindo os eventos que serão testados da *app* e a busca pelos elementos da interface. No passo três obtêm-se como saída um caso de teste que pode ser usado para a mesma *app* em dispositivos com diferentes configurações. Finalmente o último passo consiste na execução do *script* gerado no passo três em configurações diferentes, verificando as inconsistências entre as várias configurações.

2.5 TRABALHOS RELACIONADOS

Alguns projetos já desenvolvidos apresentam relação com esta proposta. MENEGASSI; ENDO (2016) analisaram três *apps cross-platform* desenvolvidos em Apache Cordova/PhoneGap. O intuito do trabalho era verificar se cada *app* se comportava de maneira similar em dispositivos com diferentes configurações. Além disso o estudo propunha codificar um *script* de teste de interface para cada *app* e posteriormente fazer a validação. Os autores concluíram que nos testes manuais não houve nenhum problema. Entretanto, na execução dos testes automatizados foram percebidas algumas diferenças, como eventos que não foram executados, componentes de interface que se dispunham em lugares diferentes de acordo com a configuração do dispositivo.

Em outro projeto também dos autores MENEGASSI; ENDO, eles propõem a criação de uma ferramenta para auxiliar a criação de testes de interface. No projeto são utilizados quatro aplicativos desenvolvidos com o *framework* Apache Cordova. A ferramente foi intitulada x-PATeSCO (*cross-Platform App Test Script reCOrder*). O objetivo da criação é minimizar o tempo gasto arrumando os casos de testes quebrados dada alguma alteração na *app*. No estudo os aplicativos foram instalados em dispositivos com diferentes configurações e características. Foram definidos os casos de teste e a ferramenta gerava automaticamente os *scripts*. Após isso fez-se uso do *framework* de apoio, o Appium, para executar os *scripts* nas *apps*. Para identificar os elementos da tela a ferramenta faz uso de seletores XPath (W3C, 2016). Na ferramenta foram definidos três níveis, o primeiro é o *Cross-platform XPath* que encontra elementos independente da plataforma. No segundo e no terceiro nível são necessários dois XPaths, um para cada plataforma. No segundo eles são baseados no tipo de elemento e nos seus atributos chaves. E o último nível consiste nos XPaths baseados no caminho absoluto do elemento dentro da estrutura do XML. Cada controle é submetido a esses três níveis e quando encontrado por algum deles a ferramenta passa a buscar um outro elemento. Caso passe pelos três níveis e não seja encontrado, a ferramenta informa para o testador que aquele elemento não pôde ser localizado.

No geral a ferramenta foi capaz de localizar aproximadamente 90% dos controles de interface, sendo que 82% foram identificados já no primeiro nível, seguidos de 5% do segundo e 2% do terceiro. A quantidade de falhas foi próximo dos 9%.

3 PROPOSTA

Com base no conteúdo apresentado no Capítulo 2, o presente trabalho propõe verificar a eficácia da ferramenta x-PATeSCO, com o intuito de analisar se os *scripts* de testes gerados conseguem se adaptar a mudanças entre as versões de uma mesma *app*. Um diagrama da proposta pode ser visto na Figura 17.

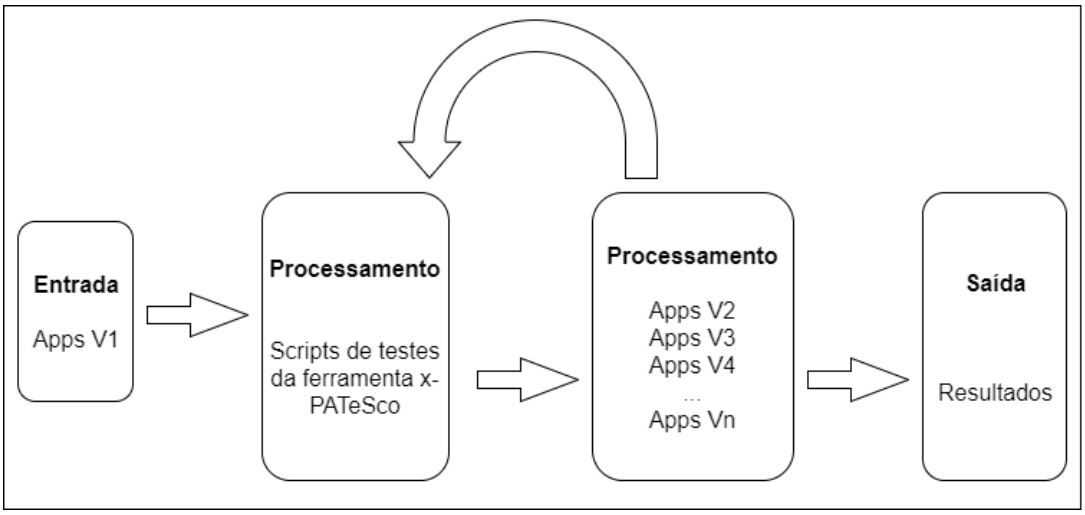


Figura 17: Arquitetura do desenvolvimento do trabalho proposto.
Fonte: Autoria própia

Para este estudo foram selecionadas três *apps* desenvolvidas em Xamarin. Elas serão identificadas por *APP1*, *APP2* e *APP3*. No início do estudo será utilizada o *.apk* da primeira versão publicada de cada *app*. Ela será instalada num dispositivo com SO Android e será submetida a ferramenta x-PATeSCO. Para cada *app*, um conjunto de casos de teste serão elaborados e gravados na ferramenta. Em um passo seguinte, os mesmos serão exportados como um projeto para o Visual Studio. Feito isso, o projeto de teste será aplicado a cada versão posterior para cada *app*, verificando a eficácia da ferramenta em lidar com as possíveis alterações de interface. A *APP1* possui vite e oito versões publicadas na Play Store, a *APP2* possui cinco e a *APP3* possui quatro.

O cronograma compreende o período de agosto de 2017 a junho de 2018.

Atividade 1: Definição do tema do trabalho.

Atividade 2: Estudar o *framework* Xamarin, utilizado para criação das *apps*.

Atividade 3: Estudar a ferramenta x-PATeSco que será utilizado para gerar os *scripts* de teste.

Atividade 4: Estudar o *framework* Appium que será o servidor para os testes nos dispositivos.

Atividade 5: Buscar trabalhos relacionados que apresentam alguma solução para realizar testes automatizados em *apps cross-platform*.

Atividade 6: Entrega e apresentação da proposta.

Atividade 7: Realizar correções na proposta sugeridas pela banca.

Atividade 8: Trabalhar na integração do *framework* Appium e da ferramenta x-PATeSco, artefato que será resultado deste trabalho.

Atividade 9: Condução do estudo de caso que irá validar o artefato gerado pelo trabalho.

Atividade 10: Elaboração do trabalho com apresentação dos resultados obtidos.

Atividade 11: Entrega e apresentação do trabalho final para a banca examinadora.

Atividade 12: Correções no trabalho final conforme sugestões da banca.

A Tabela 6 apresenta o cronograma planejado para a elaboração do trabalho.

ATIVIDADES	AGO 2017	SET 2017	OUT 2017	NOV 2017	DEZ 2017	JAN 2018	FEV 2018	MAR 2018	ABR 2018	MAI 2018	J 2
Definição do tema	X										
Estudar framework Xamarin		X	X								
Estudar ferramenta x-PATeSco		X	X								
Estudar framework Appium		X	X								
Buscar trabalhos relacionados		X									
Entrega e apresentação da proposta			X								
Correções na proposta				X							
Integrar Appium e a ferramenta x-PATeSco					X	X					
Condução do estudo de caso				X	X	X	X	X			
Elaboração do trabalho							X	X	X		
Entrega e apresentação do trabalho final										X	
Correções no trabalho final											

Tabela 6: Cronograma

Fonte: Autoria própria

REFERÊNCIAS

ADOBE. Adobe PhoneGap. 2017. Disponível em: <https://phonegap.com/>.

AMAZON. AWS Device Farm. 2017. Disponível em: <https://aws.amazon.com/pt/device-farm/>.

APACHE. Apache Cordova. 2017. Disponível em: <https://cordova.apache.org/>.

APPLE. App Store. 2017.

APPLE. iOS. 2017. Disponível em: <https://www.apple.com/br/>.

APPORTABLE. **Apportable**. 2017. Disponível em: <http://www.apportable.com/>.

BEIZER, B. Software testing techniques. Van Nostrand Reinhold Co., 1990.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54–57, 2008.

BITBAR. Mobile App Testing. 2017. Disponível em: <https://bitbar.com/testing/>.

BOUSHEHRINEJADMORADI, N. et al. Testing cross-platform mobile app development frameworks (t). In: **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 441–451.

C#. Microsoft C#. 2017. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/>.

CHANTELLE. **App Stores Start to Mature - 2016 Year in Review**. 2017. Disponível em: <http://blog.appfigures.com/app-stores-start-to-mature-2016-year-in-review/>.

CSS3. **W3C Cascading Style Sheet**. 2017. Disponível em: <https://www.w3.org/Style/CSS/Overview.en.html>.

ECMA. **ECMA International**. 2017. Disponível em: <http://www.ecma-international.org/>.

FACEBOOK. **Facebook React Native**. 2017. Disponível em: <https://facebook.github.io/react-native/>.

GOOGLE. **Android**. 2017. Disponível em: <https://www.android.com/>.

GOOGLE. Android Studio. 2017. Disponível em: <https://developer.android.com/studio/index.html?hl=pt-br>.

GOOGLE. **Google Play**. 2017. Disponível em: <https://play.google.com/store?hl=pt-br>.

HTML5. **W3C HyperText Markup Language**. 2017. Disponível em: <https://www.w3.org/html/>.

IBM. **IBM Mobile First**. 2017. Disponível em: <https://www.ibm.com/mobilefirst/br/pt/>.

IDC, I. D. C. **Smartphone OS Market Share, 2017 Q1**. 2017. Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>.

IONIC. IONIC. 2017. Disponível em: <https://ionicframework.com/>.

JSFOUNDATION. Appium. 2017. Disponível em: <http://appium.io/>.

MALDONADO, J. C. et al. Introdução ao Teste de Software. p. 1–49, 2004. Disponível em: <http://www.labes.icmc.usp.br/site/sites/default/files/NotaDidatica65.pdf>.

MALDONADO, J. C. et al. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. **(Publicação FEE)**, [sn], 1991.

MENEGASSI, A. A.; ENDO, A. T. Uma avaliação de testes automatizados para aplicações móveis híbridas. 2016.

MENEGASSI, A. A.; ENDO, A. T. Uma avaliação de testes automatizados para aplicações móveis híbridas (não publicado). 2017.

MICROSOFT. Microsoft. 2017. Disponível em: <https://www.microsoft.com/pt-br?SilentAuth=1&wa=wsignin1.0>.

MICROSOFT. **Microsoft Windows**. 2017. Disponível em: <https://www.microsoft.com/pt-br/windows/get-windows-10>.

MICROSOFT. MVVM Pattern. 2017. Disponível em: <https://msdn.microsoft.com/en-us/library/hh848246.aspx?f=255&MSPPError=-2147217396>.

MICROSOFT. **O que é um aplicativo da Plataforma Universal do Windows (UWP)?** 2017. Disponível em: <https://docs.microsoft.com/pt-br/windows/uwp/get-started/whats-a-uwp>.

MICROSOFT. **Silverlight**. 2017. Disponível em: <https://www.microsoft.com/silverlight/>.

MICROSOFT. Unit Testing Framework. 2017. Disponível em: <https://msdn.microsoft.com/en-us/library/ms243147%28vs.80%29.aspx?f=255&MSPPError=-2147217396>.

MICROSOFT. **Visual Studio**. 2017. Disponível em: <https://www.visualstudio.com/pt-br/vs/>.

MICROSOFT. WPF. 2017. Disponível em: <https://msdn.microsoft.com/pt-br/library/mt149842(v=vs.110).aspx>.

MONO. Mono. 2017. Disponível em: <http://www.mono-project.com/>.

MYAPPCONVERTER. **MyAppConverter**. 2017. Disponível em: <https://www.myappconverter.com/>.

MYERS, G. J. **77ie Art of Software Testing**. [S.l.]: New York: John Wiley and Sons, 1979.

.NETFRAMEWORK. .Net Framework. 2017. Disponível em: <https://www.microsoft.com/net/>.

NICOLI, L. **Entendendo Xamarin.Forms**. 2017. Disponível em: <https://www.lambda3.com.br/2017/04/entendendo-xamarin-forms/>. Acesso em: 09 de outubro de 2017.

ORSO, A.; ROTHERMEL, G. Software testing: a research travelogue (2000–2014). In: ACM. **Proceedings of the on Future of Software Engineering**. [S.l.], 2014. p. 117–132.

PRESSMAN, R. S. **Software Engineering A Practitioner's Approach 7th Ed - Roger S. Pressman**. [s.n.], 2009. 0 p. ISSN 1098-6596. ISBN 978-0-07-337597-7. Disponível em: <http://dinus.ac.id/repository/docs/ajar/RPL-7th_ed_software_engineering_a_practitioners_approach_by_roger_s._pressman_.pdf>.

ROSE, P. **Xamarin –Why we like it, Why Microsoft bought it**. 2016. Disponível em: <https://www.linkedin.com/pulse/xamarin-why-we-like-microsoft-bought-peter-rose/>. Acesso em: 9 de outubro de 2017.

SAÚDE, M. da. **IMC**. 2017. Disponível em: <http://portalsaude.saude.gov.br/dicas-de-saude/imc-em-adultos.html>.

SELENIUM. SeleniumHQ. 2017. Disponível em: <http://www.seleniumhq.org/docs/03_webdriver.jsp>.

TESTOBJECT. TestObject. 2017. Disponível em: <https://testobject.com/>.

W3C. XML Path Language (XPath). 2016. Disponível em: <https://www.w3.org/TR/1999/REC-xpath-19991116/>.

XAMARIN. **Xamarin**. 2017. Disponível em: <https://www.xamarin.com/>.

XAMARIN. **Xamarin Studio**. 2017. Disponível em: <https://developer.xamarin.com/guides/cross-platform/xamarin-studio/>.

XAMARIN.ANDROID. Xamarin.Android. 2017. Disponível em: <https://developer.xamarin.com/guides/android/>.

XAMARIN.FORMS. Xamarin.Forms. 2017. Disponível em: <https://developer.xamarin.com/guides/xamarin-forms/>.

XAMARIN.IOS. **Xamarin.iOS**. 2017. Disponível em: <https://developer.xamarin.com/guides/ios/>.

XAML. Xamarin XAML. 2017. Disponível em: <https://developer.xamarin.com/guides/xamarin-forms/xaml/xaml-basics/>.