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

LUCAS FERNANDES COELHO

TESTES AUTOMATIZADOS DE APLICAÇÕES MÓVEIS DESENVOLVIDAS USANDO O FRAMEWORK XAMARIN

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2017

LUCAS FERNANDES COELHO

TESTES AUTOMATIZADOS DE APLICAÇÕES MÓVEIS DESENVOLVIDAS USANDO O FRAMEWORK XAMARIN

Trabalho de Conclusão de Curso apresentada ... da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Bacharel em Engenharia de Software

Orientador: Prof. Doutor André Takeshi Endo

CORNÉLIO PROCÓPIO

2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio
Diretoria de Graduação e Educação Profissional
Programa de Graduação em ...
Engenharia de Software

TERMO DE APROVAÇÃO

Testes automatizados de aplicações móveis desenvolvidas usando o framework Xamarin

por

Lucas Fernandes Coelho

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "... em ..." e aprovado em sua forma final pelo Programa de Graduação em ... da Universidade Tecnológica Federal do Paraná.

Cornélio Procópio, XX/XX/XXXX

Prof. Doutor, André Takeshi Endo
Universidade Tecnológica Federal do Paraná

Prof. Titulação, Nome professor membro da banca
Universidade Tecnológica Federal do Paraná

Prof. Titulação, Nome professor membro da banca
Universidade Tecnológica Federal do Paraná

"A Folha de Aprovação assinada encontra-se na Coordenação do Curso"

RESUMO

Coelho, Lucas . Testes automatizados de aplicações móveis desenvolvidas usando o framework Xamarin. 21 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2017.

Texto do resumo (máximo de 500 palavras).

Palavras-chave: Palavra-chave 1, Palavra-chave 2, XXXXXXX, teste de software

ABSTRACT

Coelho, Lucas . Title in English. 21 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2017.

Abstract ... (maximum of 500 words).

Keywords: Keyword 1, Keyword 2, ...

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, muito espaço de armazenamento e uma variedade diversificada de sensores (BOUSHEHRINEJADMORADI et al., 2015). Eles são controlados por um sistema operacional (SO). Numa pesquisa realizada pela International Data Corporation (IDC, 2017) sobre o Market Share dos SOs por dispositivo do primeiro trimestre de 2017(1Q17), os sistemas mais utilizados são 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 usuários baixam e instalam em seus dispositivos. Em uma análise destas *app stores* feita pela appfigures (CHANTELLE, 2017), mostrou que a quantidade de *apps* ativos em 2016 na loja oficial do Android, a Google Play (GOOGLE, 2017c), contava com aproximadamente 2,81 milhões de aplicativos e a iOS App Store (APPLE, 2017a), da Apple, tinha algo em torno de 2,26 milhões. Esse *apps* são utilizados em todas as áreas, finanças, educação, jogos.

Com a crescente evolução dos sistemas computacionais, tem-se uma preocupação na 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 (**??**). Teste de software é uma das abordagens mais praticadas para avaliar a qualidade do software e é um dos temas mais pesquisados na engenharia de software (ORSO; ROTHERMEL, 2014). No desenvolvimento de *apps* deve haver um processo rigoroso de testes, já que caso algum defeito seja encontrado apenas quando o *app* já está nas *app stores*, isso pode influenciar a opinião do usuário sobre o *app*, classificando-o negativamente. Para fazer a correção o time de desenvolvimento precisar 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 [6].

1.1 MOTIVAÇÃO

No cenário de desenvolvimento de software convencional o time estuda um problema, encontra uma solução e em seguia a implementa (BERNARDO; KON, 2008). Somente todo o processo ter ocorrido é que o desenvolvedor realizará os testes no software. Testes manuais que consistem em designar uma ou mais pessoas para testar manualmente o que foi desenvolvido pela equipe. A execução desses testes visa encontrar falhas no sistema durante o processo de desenvolvimento, contudo é uma tarefa cansativa e que demanda muito tempo dos testadores. Neste cenário é comum que não sejam verificados novamente todos os casos de teste a cada mudança significativa no código. Isto pode gerar muitos erros no software, podendo atrasar a entrega do produto final ou no pior caso, o defeito ser encontrado pelo usuário já em produção. Todos esses problemas geram custo para a empresa. Ainda sobre os custos, conforme a quantidade de desenvolvedores na empresa cresce, o número de testadores deve acompanhar este crescimento.

Testes automatizados são programas ou *scripts* que tem a finalidade de testar as funcionalidades criadas. A vantagem da automatização dos 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 ferramenta poderosa para minimizar a quantidade de defeitos, uma vez que, quando o desenvolvedor produz uma funcionalidade ela deve ter um caso de teste automatizado correspondente. Assim se a equipe decidir incluir novas funcionalidades que alterem a estrutura dos códigos previamente desenvolvidos basta que o testador execute novamente todos os *scripts* e verificar se algum parou de funcionar. Além disso, omo os casos de teste são códigos que serão interpretados pela máquina, o testador pode incluir combinações de valores complexas que um ser-humano provavelmente não testaria.

Para este estudo será utilizado a ferramenta x-PATeSco (cross-Platform App Test Script reCOrder), desenvolvida por Menegassi e Endo (2017), não publicado. Baseada no *framework* Appium (JSFOUNDATION, 2017), esta ferramenta tem como objetivo principal realizar testes automatizados de sistema em *apps*. Tentando localizar automaticamente componentes de interface em dispositivos com as mais variadas configurações. Ela pode ser usada em ambos os sistemas, Android e iOS. A ferramenta visa a melhoria no processo desenvolvimento e testes de *apps cross-platform* (multiplataforma).

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*. Os testes serão gerados com base na *app* Forlogic Tasks (FORLOGIC, 2017) na sua primeira versão publicada na Google Play Store. Em posse desses *scripts* serão realizados novos testes nas versões subsequentes.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho foi dividido da seguinte forma: o Capítulo 2 apresenta o embasamento teórica 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 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 será explicado 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 2.3.1 serão abordadas as tecnologias Xamain.iOS e Xamarin.Androd. Na Seção 2.3.2 será introduzido o Xamarin.Forms, que será utilizado em todo o trabalho. Na Seção 2.3.3 será exibido um projeto de exemplo feito em Xamarin.Forms, explicando todas as suas características e funcionalidades. Na Seção 2.4 será apresentado o *framework* Appium. Na Seção 2.5 serão apresentados os trabalhos relacionados.

2.1 TESTES 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. Na primeira fase, 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, testa-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 conhecido 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: Grafo de Fluxo de Controle, Grafo de fluxo de dados e Teste de Mutação.

2.2 APLICAÇÕES MÓVEIS

Existem várias formas para desenvolver *apps* para as plataformas móveis e podem ser classificadas em três grupos: aplicações nativas, aplicações Web e aplicações híbridas (MALDONADO et al., 2004). Aplicações nativas são desenvolvidos 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 publicar a mesma *app* em outras lojas. 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 *frameworks* de desenvolvimento que abstraem todo o modo de criação de *apps*, utilizando apenas o que o *framework* fornece é possível criar aplicações para os SOs mais comuns 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 su-

portam totalmente todos os recursos dos dispositivos como câmera, microfone, agenda de contatos e preferências do telefone. Portanto para utilizar essas ferramentas 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, 2017). 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# (C#, 2017) para criar a lógica de negócio e XAML (XAML, 2017) para 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, chamando os controles nativos.

2.3 XAMARIN

Xamarin é um *framework* de desenvolvimento *cross-plataform* 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 técnologias 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 o mesmo *app* para iOS, obrigando o os desenvolvedores recriarem toda a interface e lógica de apresentação, reaproveitado apenas a lógica de aplicação, como pode ser visto na Figura 1.

Figura 1: Arquitetura de desenvolvimento de *apps* Xamarin.Android e Xamarin.iOS Fonte: (site)

Na figura 1, é possível notar que todo o código da lógica da aplicação é escrito em C# e utilizado em todas as plataformas (Shared C# App Logic), contudo a para cada uma das plataformas será necessário recriar a lógica de apresentação (Plataform-specific C#). As *apps* são escritas na linguagem C# e compilados e utilizando a 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 2, tanto a lógica de aplicação (Shared C# App Logic) quanto a lógica de apresentação e interface (Shared C# User Interface Code).

Figura 2: Arquitetura de desenvolvimento de aplicativos Xamarin.Forms Fonte (site)

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 (altura x altura, em metros) [25]. 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 3: Aplicativo exemplo ? Tela inicial Fonte: Autoria própria

2.3.3 ESTRUTURA DO PROJETO

A IDE necessária para codificar uma aplicação Xamarin é o Xamarin Studio [26] no Apple Mac OS [27] e o Visual Studio [28] no Microsoft Windows [29]. 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, 2017b), criado pela Microsoft e era anteriormente utilizado pelo WPF (MICROSFT, 2017) e Silverlight [32]. 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 imagem a seguir 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 temos 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 no final acrescenta-se ?ViewModel?. No exemplo a MainPage receberá o sufixo ViewModel, sendo nomeada como MainPageViewModel.cs. Outra classe importante para o projeto é a BaseViewModel; essa classe tem a abstração de alguns métodos ou atributos que serão utilizados em todas as outras ViewModels.

Figura 4: 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 5:

Figura 5: Trecho de código da classe App.xaml.cs Fonte: Autoria própria

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 plicação 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.

Figura 6: Trecho de código da View MainPage.xaml Fonte: Autoria própria

Na figura 6 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 para qual estiver desenvolvendo. É possível notar que alguns controles, como os as 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 verdadeiro ou falso, 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.

Figura 6: Trecho de código da classe MainPage.xaml.cs Fonte: Autoria própria

A figura 7 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 View deve associar as suas propriedades.

Figura 8: trecho de código da classe BaseViewModel.cs Fonte: Autoria própria

Na figura 8 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 PropertyChangedEcentHandler, na linha 15 que junto com o método OnPropertyChanged(), na linha 17, é responsável por monitorar as propriedades e informar quando elas foram alteradas. Já na linha 22 temos 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.

Figura 9: Trecho de código da classe MainPageViewModel.cs Fonte: Autoria própria

No trecho de código da figura 9 a classe MainPageViewModel.cs implementa a Base-ViewModel.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.

Figura 10: Trecho de código da classe MainPageViewModel.cs Fonte: Autoria própria

A figura 10 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 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

Appium é um *framework open source* para automatizar testes em *apps mobile*. Com ele é possível testar *apps* nativas, híbridas e *apps* Web. Além disso, ele é *cross-platform*, pos-

sibilitando 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, 2017c). O projeto possui uma estrutura que pode ser utilizado 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.

2.5 TRABALHOS RELACIONADOS

Artigos do André

3 PROPOSTA

Com base no conteúdo apresentado no Capítulo2, 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*.

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 que será utilizado para criação das *apps*.

Atividade 3: Estudar o *framework* 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 aplicativos *cross-platform*

Atividade 6: Entrega e apresentação da proposta.

Atividade 7: Realizar correções na proposta sugeridas pela banca.

Atividade 8: Elaboração do trabalho com apresentação dos resultados obtidos.

Atividade 9: Entrega e apresentação do trabalho final para a banca examinadora.

Atividade 10: Correções no trabalho final conforme sugestões da banca.

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/>.

FORLOGIC. **Forlogic Tasks**. 2017. Disponível em: <https://play.google.com/store/apps/details?id=com.fo

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.

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

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

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

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

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/>.

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>.

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

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

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

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/>.