CÂMPUS DE PALMAS

CURSO DE SISTEMAS DE INFORMAÇÃO

ESTUDO DE COMPATIBILIDADE ENTRE PROJETOS DE APLICATIVOS NATIVOS PARA ANDROID E IOS

Autor: SÁVIO MARTINS VALENTIM

Palmas – TO

2018

CÂMPUS DE PALMAS
CURSO DE SISTEMAS DE INFORMAÇÃO

ESTUDO DE COMPATIBILIDADE ENTRE PROJETOS DE APLICATIVOS NATIVOS PARA ANDROID E IOS

Autor: SÁVIO MARTINS VALENTIM

Trabalho apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Tocantins - UNITINS como parte dos requisitos para a obtenção do grau de Bacharel em Sistemas de Informação, orientado pelo Prof. Me. Silvano Maneck Malfatti.

Palmas, junho de 2018

CÂMPUS DE PALMAS
CURSO DE SISTEMAS DE INFORMAÇÃO

ESTUDO DE COMPATIBILIDADE ENTRE PROJETOS DE APLICATIVOS NATIVOS PARA ANDROID E IOS

Autor: SÁVIO MARTINS VALENTIM

Trabalho apresentado ao Curso de Sistemas de Informação da Universidade Estadual do Tocantins - UNITINS como parte dos requisitos para a obtenção do grau de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____

Prof. Me. Silvano Maneck Malfatti

_____

Prof. Me. Jânio Elias Teixeira Júnior

_____

Prof. Esp. Fredson Vieira Costa

DEDICATÓRIA

A minha família, pelo apoio incondicional.

AGRADECIMENTOS

Primeiramente agradeço a Deus por mais essa etapa de minha vida, por ter me dado força, perseverança e a coragem suficientes para concluir esse curso. Também agradeço aos meus pais e meus irmãos que sempre acreditaram em mim e fizeram de tudo para que e não desistisse ou desanimasse desse curso.

Agradeço também aos meus amigos que sempre me apoiaram e me ajudaram durante toda a minha vida. Agradeço aos colegas de curso que dividiram comigo todos os momentos vivenciados nessa jornada acadêmica e que hoje seguem outros caminhos.

Agradeço aos professores, especialmente ao meu orientador, por terem acreditado em mim e terem me ensinado o que eu sei, os quais lembrarei pela importância que tiveram em minha vida.

RESUMO: Este trabalho apresenta um estudo comparativo com o objetivo de analisar a compatibilidade de recursos entre projetos nativos de Android e iOS. Como estudo de caso, foi desenvolvido um aplicativo de localização de sala no qual um estudo foi realizado durante o desenvolvimento entre as duas plataformas, concluindo que as diferenças entre os sistemas dificultam a reutilização de recursos em um desenvolvimento nativo.

Palavras-chave: dispositivos móveis, Android, iOS, desenvolvimento de aplicativos nativo.

ABSTRACT: This work presents a comparative study in aim to analyze the compatibility of resources between native Android and iOS projects. As a case study, a room location application was developed in which a study was carried out during the development between the two platforms, concluding that differences between systems make it difficult to reuse resources in a native development.

Key-word: mobile devices, Android, iOS, native application development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE PALAVRAS ABREVIADAS

API: Application Programming Interface.

APP: Aplicativo.

iOS: iPhone Operation System.

SO: Sistema Operacional.

SGBD: Sistema de Gerenciamento de Banco de Dados.

MVC: Model-View-Controller.

GUI: Graphical User Interface.

KB: kilobyte.

OHA: Open Handset Alliance.

XML: eXtensible Markup Language.

IDE: Integrated Development Environment.

HTML: HyperText Markup Language.

GPS: Global Positioning System.

APK: Android Application Pack.

SUMÁRIO

1. INTRODUÇÃO

Os dispositivos móveis têm evoluído proporcionalmente junto a tecnologia, passando de simples aparelhos celulares usados em telefonemas e SMS para *smartphones* tão potentes quanto um computador.

Esses dispositivos são capazes de acessar a *internet* e facilitam a vida de milhões de usuários, junto a essa evolução abriu-se um novo para o mercado de desenvolvimento de aplicativos para esses dispositivos.

O objetivo desse mercado é desenvolver aplicações para dispositivos móveis que possam facilitar a vida dos usuários - aplicativos de banco, *e-mail*, leitores de código de barras, entre outros - melhorando dessa forma o trabalho e o lazer dos usuários.

O uso de dispositivos móveis vem crescendo consideravelmente no mundo, com o Brasil sendo um dos países que mais se destaca nesse assunto, com o acesso à *internet* por estes dispositivos superando o acesso por computadores desktop (Cryah, 2016).

Dos dispositivos utilizados em 2017 o Android dominou 85,9% do mercado e o iOS (*iPhone Operation System*) ficou com 14%, com o restante ficando com aparelhos menores como o Blackberry e o Windows Phone (Gartner, 2018).

Investir nessa área tem sido uma boa aposta para as empresas de desenvolvimento, pois além da *internet* móvel ser mais utilizada que em *desktops*, a venda de *smartphones* é crescente e as pessoas gastam mais tempo com um APP (Aplicativo) do que em sites ou utilizando o próprio computador (UNIFEIJr, 2017).

Os sistemas de *smartphones* predominantes no mercado do Brasil são o Android da Google e o iOS da *Apple* - que embarca apenas em seus produtos como *iPhones, iPads* e *iPods*.

O Android é o sistema operacional móvel mais utilizado no mundo e está disponível em várias plataformas, como *smarthphones*, *tablets*, TV, relógios, óculos e carros (Lecheta, 2016), o que demonstra que o Android é um sistema feito diversas plataformas não apenas os *smartphones* e *tablets*, incluindo sistemas embarcados e em acessórios, como relógios, TVs e até mesmo óculos entrando nessa lista.

A Apple não ficou atrás, lançando aparelhos que vão além dos tradicionais *Mac* e *Macbook*, aos inovadores *iPhone* e *iPads*, lançando o *Apple Watch*, um relógio com o sistema *watchOS* e o *Apple* TV.

Ambos são voltados para os usuários de aparelhos da empresa, espelhando conteúdo do *Mac*, do *iPhone* e do *iPad* nesses dispositivos.

O trabalho a seguir apresenta um estudo de um tipo de especifico de desenvolvimento: o nativo, que utiliza apenas recursos da própria plataforma para o qual o aplicativo está sendo desenvolvido, avaliando as diferenças e a possibilidade de reaproveitamento no desenvolvimento nativo entre o Android e iOS.

1.1. OBJETIVOS

O objetivo deste trabalho é apresentar o desenvolvimento de um aplicativo para *smartphones* com Android e iOS utilizando recursos, APIs (*Application Programming Interfa*ce), bibliotecas e *frameworks* nativos dos dois sistemas citados, a partir disto os objetivos específicos são:

- Analisar o desenvolvimento nativo do Android e iOS utilizando o mesmo projeto;
- Demonstrar as diferenças, semelhanças, reaproveitamentos e dificuldades entre os projetos.

1.2. PROBLEMA

O mercado *mobile* vem crescendo, com esses dispositivos se tornando uma parte essencial na vida das pessoas. Esse fator também se estende ao mundo corporativo, com as empresas adotando o uso de aplicativos para suas necessidades e aumentar sua produtividade.

Com esse cenário, o desenvolvimento mobile se tornou um mercado atrativo, mesmo com esses dispositivos – *tablets, smartphones e watchs* – possuindo uma arquitetura interna diferente de um computador pessoal.

Por exemplo, a quantidade de memória é menor, o processamento é mais lento e a bateria tem uso limitado, sendo esses alguns obstáculos encontrados pelos desenvolvedores quando se fala desse tipo de dispositivo.

Atualmente são dois os sistemas operacionais móveis que dominam o mercado: o Android da Google e o iOS da Apple. Normalmente, as aplicações são desenvolvidas separadamente para ambos e funcionam em dispositivos de diversos tamanhos e *hardwares* diferentes.

Porém, o sistema Android tem diferenças em relação ao sistema iOS e desenvolver em cada uma dessas plataformas exige conhecimentos e ferramentas diferentes com cada um dos sistemas tendo suas particularidades.

Por isso se tratando do mesmo escopo de projeto, atendendo aos mesmos requisitos e funcionalidades, existe alguma compatibilidade entre projetos de software para ambos os sistemas ou recursos que possam ser reaproveitados e compartilhados entre eles e ferramentas que permitam essa compatibilidade?

O desenvolvedor que deseja implementar um projeto para os dois sistemas pode escolher entre desenvolvimento nativo e o desenvolvimento híbrido, porém deve-se levar em consideração que o nativo precisará ser realizado duas vezes enquanto o híbrido não consegue alcançar o nível de performance do primeiro.

1.3. JUSTIFICATIVA

O mercado *mobile* tem crescido rapidamente e o desenvolvimento de sistemas para dispositivos móveis está acompanhando as tendências exigidas pelo mercado, por isso as empresas estão atentas e procuram cada vez mais mão de obra qualificada para as principais plataformas *mobile* disponíveis no mercado - o iPhone da Apple utiliza o iOS e a Google que utiliza o Android presente em grande parte dos modelos de *smartphones*.

Projetos de software que visam estar em múltiplas plataformas exigem uma atenção especial, pois o mesmo escopo de um projeto deve ser executado de variadas formas, detalhe este que pode ser a causa de muitos problemas principalmente entre desenvolvedores inexperientes.

Em projetos *mobile* não é diferente, um projeto para a plataforma Android tem diferenças significativas em comparação a um projeto para a plataforma o iOS, tanto empresas quanto desenvolvedores independentes realizam projetos para dispositivos móveis, e o fato do desenvolvimento nativo não possuir as mesmas tecnologias, ferramentas, métodos e conhecimento necessários nas duas plataformas representa um desafio o qual precisa ser superado por que quem deseja seguir no desenvolvimento nativo.

2. REFERENCIAL TEÓRICO

Com o surgimento do mercado *mobile* e a crescente importância que o mesmo vêm ganhando com o decorrer dos anos, as pesquisas sobre as plataformas e tipos de desenvolvimento para *smartphones* cresceram com os mais diversos objetivos, possibilitando que tanto acadêmicos quanto desenvolvedores buscassem informações sobre a plataforma e desenvolvimento dos referidos sistemas.

Da Fonseca e Beder (2015) comparam o desenvolvimento web e desenvolvimento nativo no Android, apontando as vantagens e as desvantagens de cada uma das alternativas apresentadas com base em um aplicativo desenvolvido para o Departamento de Computação da Universidade Federal de São Carlos.

Toda a estrutura do Android é bem definida, demonstrando que conhecer toda a estrutura do sistema como o ciclo de vida de uma *activity* e outros recursos são indispensáveis independentemente do tipo de desenvolvimento. O aplicativo do trabalho dos autores foi desenvolvido primeiro em padrões *web* sendo convertido para nativo.

Da Silva, Oires e Neto (2015) analisam e comparam os tipos de desenvolvimento de aplicativos móveis com foco em tecnologias para o iOS a partir dos conceitos e a arquitetura do sistema da Apple.

O trabalho desses autores trás as diferenças entre o desenvolvimento nativo, híbrido e web trazendo exemplos sem se focar em uma implementação especifica, demonstrando a dificuldade de se escolher o tipo de desenvolvimento, uma escolha que pode ser difícil para um desenvolvedor iniciante e o quanto é importante conhecer a diferença entre as soluções *mobile* apresentadas.

Bezerra e Schimiguel (2015) apresentam o desenvolvimento de aplicações hibridas utilizando o framework *Phonegap* da Adobe, criando um protótipo com o framework e visando a construção de uma aplicação para Android e Windows Phone com essa ferramenta utilizando suas APIs com um *web service*, demonstrando um exemplo de projeto real de curto prazo.

Há uma discussão sobre as diferenças entre o desenvolvimento híbrido e nativo, com o hibrido tendo como vantagens o menor custo e tempo de desenvolvimento em detrimento do desempenho por possuir uma linguagem interpretada sendo recomendado pra aplicativos mais simples, e o nativo tendo vantagem do desempenho e a robustez em detrimento do tempo de execução sendo recomendado para aplicativos mais críticos.

Este trabalho segue a mesma linha dos trabalhos apresentados anteriormente, baseado nas atividades a serem desenvolvidas com o Android e iOS planejando, desenvolvendo e comparando os dois projetos para atender à necessidade da proposta.

2.1. DESENVOLVIMENTO NATIVO

O desenvolvimento nativo é aquele que visa construir um APP ou sistema para um SO (Sistema Operacional) especifico, fazendo com que o APP desenvolvido seja compilado, instalado e interpretado diretamente pelo SO sem a necessidade de nenhum interpretador externo (Da Silva, Oires e Neto, 2015).

Por todas essas diferenças o desenvolvimento nativo para dispositivos móveis varia de acordo com a plataforma ao qual o APP se destina, pois cada plataforma possuí linguagens e ferramentas de desenvolvimento distintas (Bezerra e Schimiguel, 2015).

Diferente do desenvolvimento híbrido, um mesmo APP deve ser desenvolvido na linguagem nativa e compilado diretamente para o SO ao qual o APP se destina, aumentando o nível de complexidade do desenvolvimento.

2.2. ANDROID

O Android é um sistema operacional baseado em Linux, operando em *smartphones*, *tablets* e relógios o qual é desenvolvido pelo OHA (*Open Handset Alliance*), uma aliança entre várias empresas, dentre elas a Google (*Open Handset Alliance*, 2018).

O funcionamento do Android é idêntico a outros sistemas operacionais como Windows, iOS e Linux cuja função é gerenciar todos os processos dos aplicativos e do hardware da plataforma para que funcionem perfeitamente.

A diferença é que o Android foi impulsionado pela Google através da OHA para ser operado nos seus próprios dispositivos móveis, entrando na concorrência com outros sistemas operacionais dominantes no lançamento em 2008, como o Symbian (Nokia), iOS (Apple) e *Blackberry OS*.

Uma de suas características é a integração dos serviços Google a partir de uma conta do usuário, junto a isso trazendo a vantagem da *Play Store*, a loja oficial dos aplicativos que funciona como um repositório oferecendo assim muitos aplicativos gratuitos e pagos.

As linguagens de programação utilizadas são o Java e o C++, com o Kotlin anunciado em maio de 2017 como a nova linguagem oficial. Além disso, o Android é de código aberto, possibilitando aos desenvolvedores baixarem seu código fonte, realizar

testes, modificações no sistema e notícias sobre o andamento do desenvolvimento da plataforma (*Android Source*, 2018).

2.2.1. ARQUITETURA DO ANDROID

A arquitetura do sistema operacional Android é dívida em camadas, aonde cada parte é responsável por gerenciar os seus respectivos processos (Lecheta, 2016), a figura 1 demonstra graficamente como essa arquitetura está distribuída em 4 camadas e sempre que possível um desenvolvedor deve olhar primeiro as camadas superiores, pois nelas estão os serviços e *frameworks* que fornecem as abstrações para as camadas inferiores.

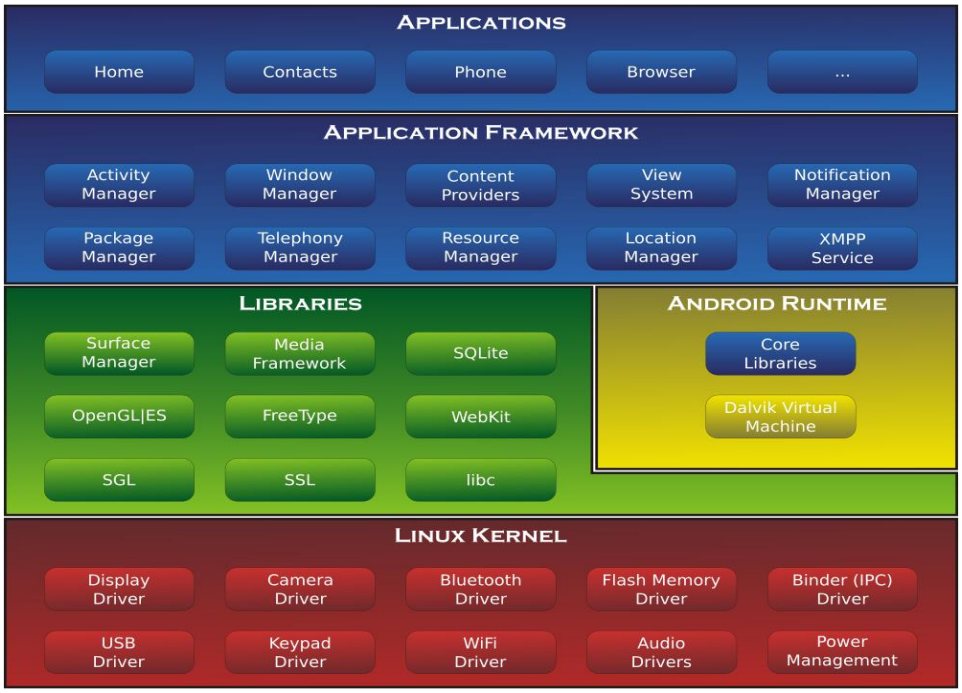


Figura 1: Arquitetura do Sistema Android

Fonte: https://www.devmedia.com.br (2018)

A primeira camada é a de aplicação, sendo nela onde os aplicativos são executados sobre o sistema operacional, tais como aplicativos de SMS, *e-mail*, navegador, mapas, calculadora e outros que são utilizados tanto pelo sistema quanto pelo usuário (*Android Developers*, 2018).

Na segunda é onde estão todas as bibliotecas do sistema, desde as bibliotecas Java e C/C++ que são utilizadas pelo sistema até as bibliotecas de funções de acesso a banco de dados *SQLite* (*Android Developers*, 2018).

A terceira é a *Runtime*, nela se localiza a máquina virtual criada para as aplicações executadas, possibilitando a integração do sistema com o *hardware* e executar vários

processos paralelamente, além de aperfeiçoar a coleta de lixo do sistema (*Android Developers*, 2018).

A última camada é a Linux Kernel, sendo a fundação de todo o sistema operacional e um derivado dos sistemas Linux, tendo herdado características similares do mesmo como a segurança, gerenciamento de memória e permissões (*Android Developers*, 2018).

2.2.2. CICLO DE VIDA DE UMA ACTIVITY

O ciclo de vida de um aplicativo Android está ligado diretamente ao ciclo de vida de uma *Activity*, pois quando um usuário navega através de uma aplicação Android ocorre uma série de eventos e à medida que um usuário navega, sai e volta para seu aplicativo, a *Activity* instância na transição do aplicativo por diferentes estados em seu ciclo de vida (*Android Developers*, 2018), a figura 2 mostra como é o ciclo de vida e a maneira que seus processos estão organizados.

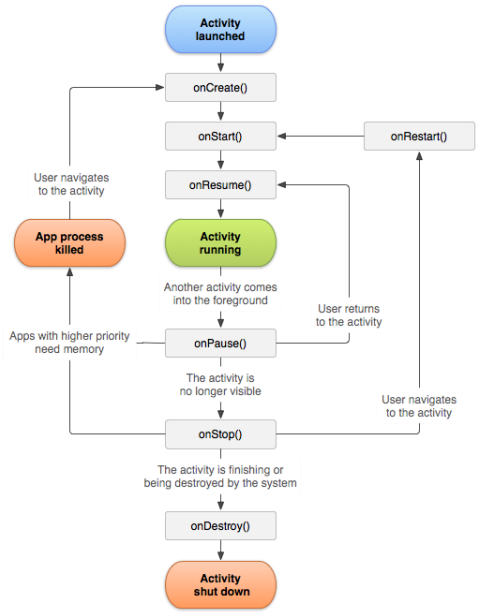


Figura 2: Ciclo de Vida de um Aplicativo Android.

Fonte: https:// developer.android.com (2018)

Existem sete processos de ciclo de vida em uma *activity* do Android (*Android Developpers*, 2018), são eles:

- ***onCreate*:** Chamado quando a *activity* é criada pela primeira vez. É o método principal sendo sempre o primeiro a ser chamado, necessário ser programado;
- ***onStart***: Chamado antes de a aplicação ficar visível para o usuário;
- ***onResume***: Chamado quando a aplicação irá interagir com o usuário;

- ***onPause*:** Chamado quando a aplicação estiver prestes a retomar ou mostrar outra *activity*;
- ***onStop*:** Chamado quando a aplicação não estiver mais sendo executada ou colocada em segundo plano;
- ***onRestart*:** Chamado, necessariamente, quando a aplicação estiver prestes a ser chamada de novo;
- ***onDestroy:*** Chamado antes de uma *activity* ou aplicativo serem finalizados.

2.3. iOS

O iOS é o sistema operacional que está nos dispositivos moveis da Apple, tanto os *iPhone* quanto os *iPad*, sendo da mesma forma que o Android, uma versão portátil de um sistema desktop, o iOS é a versão portátil do sistema da Apple do macOS, que está presente no *Mac*, *MacMini*, *MacBook* e outros dispositivos da empresa (*Apple Developer*, 2018).

O iOS é um sistema proprietário e roda apenas no hardware fornecido nos aparelhos da própria Apple, não sendo compatível com *hardwares* diferentes dos que saem da fábrica da empresa, diferente de outros sistemas como o Windows, Linux e FreeBSD, além disso o desenvolvimento do sistema fica restrito apenas a própria Apple, seguindo o mesmo exemplo de empresas como a Microsoft.

Foi anunciado em 2007 e a princípio usava como linguagem de programação base para seus aplicativos o Objective-C, mudando para a linguagem Swift em 2014, mantendo a compatibilidade para os aplicativos escritos em Objective-C, com a tendência de abandonar essa linguagem com o passar das versões a aprimoramentos.

2.3.1. ARQUITETURA DO iOS

O iOS possuí uma arquitetura distribuída em quatro camadas distintas, de maneira similar à do Android, mostrado na figura 3, com cada uma delas cumprindo uma função e oferecendo um conjunto de recursos diferentes que podem ser usados para auxiliar o desenvolvimento de aplicativos para o sistema da Apple, além de ser similar ao macOS (Rocha e Neto, 2011).

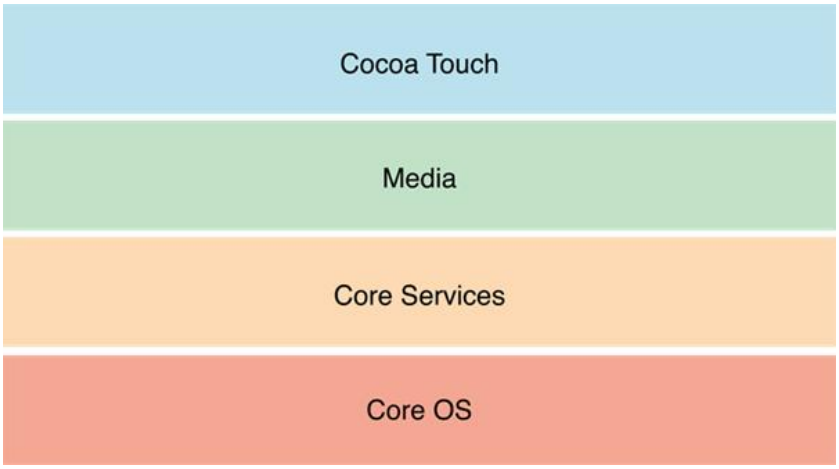


Figura 3: Arquitetura do Sistema iOS.

Fonte: https://www.devmedia.com.br/ (2018)

A primeira é a *Cocoa Touch*, onde estão os principais frameworks para a construção de aplicações, com ela definindo a infraestrutura para as tecnologias fundamentais, tais como aparência, *multi tasking*, *touch-based input*, notificações *push* e diversos serviços de alto nível do sistema (Rocha e Neto, 2011).

A segunda é a camada *Media*, que contém as tecnologias de gráfico, áudio e vídeo. As tecnologias dela foram projetadas para tomar mais fácil e prático a implementação de aplicativos multimídias (Rocha e Neto, 2011).

A terceira camada é a *Core Services*, que contém os recursos e os serviços fundamentais que são utilizados no desenvolvimento de aplicativos, como a *Foundation* e o *SQLite*, definindo os recursos que os aplicativos utilizam mesmo que indiretamente (Rocha e Neto, 2011).

Por último a camada *Core OS* é a mais baixa, sendo bastante utilizada pelas camadas de níveis superiores, mesmo que não seja utilizada diretamente ela de alguma forma é utilizada por outros *frameworks* de níveis superiores em algum momento, com sua utilização direta normalmente ocorrendo quando é preciso fazer uma negociação explicita de segurança ou uma comunicação com um hardware externo (Rocha e Neto, 2011).

2.3.2. CICLO DE VIDA DE UMA VIEW

O ciclo de vida no iOS é um evento que tem várias etapas do ponto de criação até o de exclusão em uma sequência de métodos em que o usuário acaba progredindo em seu uso (*Apple Developer*, 2018) conforme é demonstrado na figura 4 a organização e sequencia desses processos.

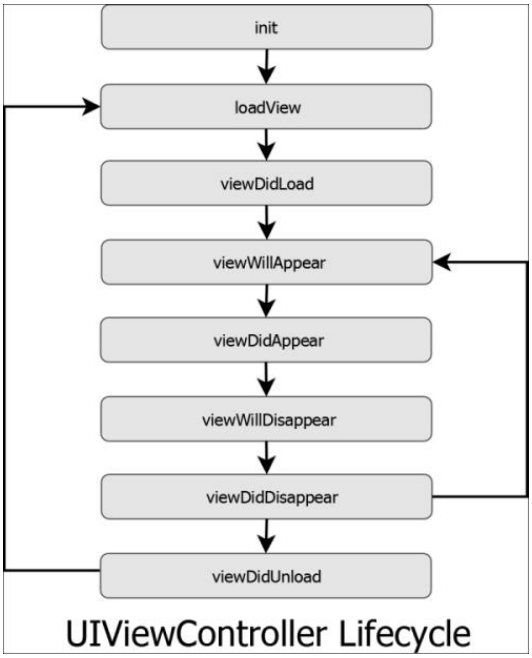


Figura 4: Ciclo de vida de um aplicativo iOS.

Fonte: https://www.safaribooksonline.com/ (2018)

São nove processos no ciclo de vida em uma *view controller* do iOS. São esses processos:

- ***init*:** É chamado para instanciar uma *view controller*, colocando os recursos que serão necessários durante a vida útil. É chamado apenas uma vez durante a vida do objeto, sendo o método construtor do ciclo;
- ***loadView*:** É chamado quando a *view controller* é criada sendo feito programaticamente. Quando a *storyboard* é usada esse método pode ser ignorado;
- ***viewDidLoad*:** É chamado apenas quando a *view* é criada e carregada na memória, sendo esse o método ideal para inicializar os objetos que a *view* irá usar. Pode ser comparado ao onCreate do Android;
- ***viewWillAppear*:** Notifica a *view controller* sempre que a *view* aparecer na tela, podendo executar ações nesse processo.
- ***viewDidAppear*:** É chamado quando a *view* estiver totalmente visível na tela;
- ***viewWillDisappear*:** É chamado quando a *view* é descartada, coberta ou escondida;
- ***viewDidDisappear*:** É chamado depois que *view* foi descartada, coberta ou escondida;
- ***didReceiveMemoryWarning*:** É chamado quando o aplicativo recebe um aviso de memória.

- ***viewDidUnload*:** É chamado quando a aplicação já terminou ou quando o sistema necessita finalizar a *view*.

2.4. WEB SERVICE

Um *web service* é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes, pois com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis (*SOAWebServices*, 2012).

Essencialmente um *web service* faz com que os recursos de uma aplicação de software estejam disponíveis sobre a rede de forma normalizada, sendo assim uma forma de interoperabilidade entre sistemas, podendo ser um recurso a ser reaproveitado entre diferentes plataformas.

2.5. SQLITE

O *SQLite* é uma biblioteca em linguagem C que implementa um banco de dados SQL embutido, permitindo aos programas que usam a biblioteca *SQLite* ter acesso a banco de dados SQL sem executar um processo de SGBD (Sistema de Gerenciamento de Banco de Dados) separado (*SQLite.org*, 2018).

O uso do *SQLite* é recomendado onde a simplicidade da administração, implementação e manutenção são mais importantes que incontáveis recursos que SGBDs mais voltados para aplicações complexas possivelmente implementam, sendo um banco perfeito para sistemas locais e sistemas embarcados, uma característica que se aplica aos sistemas *mobile*.

2.6. PADRÃO ARQUITETURAL MVC

O MVC (*Model-View-Controller*) é um padrão arquitetural de *software* bastante conhecido, usado principalmente nas aplicações web sendo o padrão de projeto escolhido para a estruturação deste projeto.

Foi inicialmente desenvolvido no intuito de mapear o método tradicional de entrada, processamento e saída que os diversos programas baseados em GUI (*Graphical User Interface*) (DevMedia, 2012), sendo essas partes separadas da seguinte forma:

- ***Model*:** representa o modelo de informações e dados do sistema;
- ***View:*** recebe entradas de dados e exibe os resultados para os usuários através da interface disponível;

- ***Controller:*** determina as ações do sistema, além de fazer o intermédio entre a *view* e o *model*.

Uma das vantagens deste padrão é o fato de que as partes são separadas podendo sofrer alterações sem afetar as outras, como mostra a figura 5 essa distribuição facilita não só a criação do código fonte, mas também sua manutenção pois não é necessária uma preocupação direta com as demais partes.

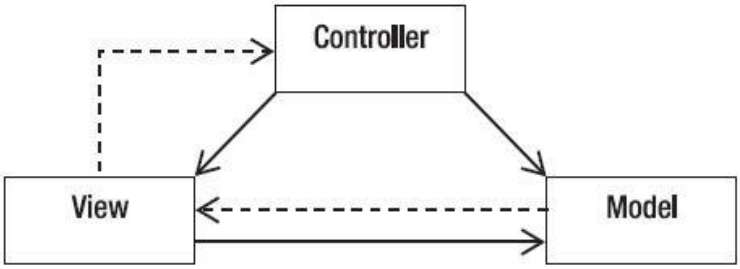


Figura 5: Objetos do MVC e suas iterações.

Fonte: www.devmedia.com.br (2018)

3. METODOLOGIA

Para realizar o comparativo foi desenvolvido e testado um aplicativo em ambos os sistemas, com a metodologia adotada no desenvolvimento visando à construção de um APP com o objetivo de prover aos alunos da UNITINS o acesso a informações sobre a universidade.

A função principal é permitir o acesso ao mapa de salas no semestre corrente, informando a sala da disciplina, o dia da semana e o horário em que ocorre, com as funções secundárias do aplicativo sendo o acesso ao endereço do campus do curso do estudante e visualizar as notícias da UNITINS, dando acesso a essas informações de maneira rápida e objetiva.

Definido o objetivo do APP, iniciou-se a modelagem do mesmo com o diagrama de classe, casos de uso, requisitos funcionais e não funcionais, *schema* de banco de dados, protótipo de telas e diagrama de arquitetura do sistema ao qual o APP será conectado para consumir os dados, com toda essa modelagem tendo por objetivo orientar o projeto em ambos os sistemas utilizando os componentes de um sistema *mobile*.

Para o desenvolvimento foi adotada uma sequência para demonstrar como é feito o aplicativo utilizando como base o padrão de projeto MVC citado na seção 5.5 com algumas adaptações, começando pela estruturação das camadas MVC, depois mostrando a estruturação da *view* com seus componentes e relacionamentos, abordando o uso de *web service*, mapas e terminando com a publicação.

Partes importantes do desenvolvimento foram detalhadas para melhores informações, o primeiro sistema a ter o aplicativo foi Android utilizando a linguagem Java, com a implementação em iOS utilizando linguagem Swift ocorrendo em seguida.

Durante o desenvolvimento no iOS foram testadas as ferramentas de conversão pesquisadas a fim de verificar a eficiência das mesmas no âmbito de reaproveitamento, possibilitando ao final do trabalho avaliar sua viabilidade.

Todo o trabalho ficou focado nas camadas superiores dos sistemas citados anteriormente (*Applications* e *Libraries* do Android e *Cocoa Touch* do iOS), pois são elas que tem importância para o desenvolvimento deste trabalho.

A seguir são elencadas as atividades no decorre deste projeto de comparativo de desenvolvimento:

1. Estudar os tipos de desenvolvimento para dispositivos móveis através de artigos científicos ou publicações buscando a definição e diferenças entre eles;
2. Pesquisar ferramentas de conversão que possam permitir o reaproveitamento de códigos e outros recursos para o iOS;
3. Modelar um projeto para ser utilizado no desenvolvimento nativo nos dois sistemas escolhidos;
4. Executar e documentar a execução do projeto no Android;
5. Repetir o passo anterior no iOS a partir do projeto realizado no Android com algumas comparações;
6. Realizar testes com ferramentas de conversão pesquisadas, a fim de verificar sua utilidade;
7. A partir dos resultados obtidos, chegar aos objetivos específicos deste trabalho.

3.1. INSTRUMENTOS

Durante toda a execução foi obedecida a ordem que foi estabelecida na descrição da metodologia, com o desenvolvimento sendo relatadas individualmente tanto no Android quanto no iOS. Os instrumentos utilizados durante essa etapa foram:

- Um Mac com XCode e simulador iOS instalados e atualizados;
- Um computador desktop ou notebook com o *Android Studio* e simulador instalados e configurados;

Com esses instrumentos a execução prosseguiu sem interrupções técnicas por conta de equipamentos ou falhas de software.

3.2. FERRAMENTAS DE DESENVOLVIMENTO

3.2.1. ANDROID STUDIO

O Android Studio é a IDE oficial do Android, criado para o desenvolvimento de aplicativos para todos os dispositivos com Android (*Android Developers*, 2018), substituindo outras IDEs como o Eclipse.

Está IDE possui ampla variedade de recursos, entre eles está à opção de utilizar um emulador Android para qualquer versão já fabricada, com o lançamento do Kotlin o Android Studio ganhou suporte para essa linguagem, bem como a possibilidade de converter automaticamente códigos em Java para a codificação Kotlin.

3.2.2. XCODE

O *XCode* é a IDE da Apple para o desenvolvimento de projetos voltados para os sistemas macOS, permitindo transformar códigos, ver alterações obter rapidamente detalhes sobre diferenças entre códigos (*Apple Developers*, 2018).

Assim como o *Android Studio* citado na seção anterior, o *XCode* conta com uma grande variedade de recursos, um dos mais notáveis sendo o *Playground* que testa código enquanto o mesmo é escrito, utilizando o emulador que é embutido no próprio macOS.

3.2.3. CONVERSORES

Foram pesquisadas algumas ferramentas de conversão utilizadas no teste de interoperabilidade do desenvolvimento dos sistemas. Essas ferramentas convertem códigos do *Android* para o iOS, conforme adotado na metodologia.

3.2.3.1. J2OBJC

O J2ObjC é uma ferramenta de código aberto da Google que traduz um código-fonte Java para *Objective-C* da plataforma iOS (*iPhone* e *iPad*).

Essa ferramenta permite que um código fonte possa fazer parte do código fonte de um aplicativo para iOS, com o objetivo é escrever o código não-UI (ou seja, que não faça parte do *layout*) de um aplicativo (como lógica de aplicativos e modelos de dados) em Java para aplicativos *Objective-C* do iOS (J2ObjC, 2018);

Sabe-se que o a *Apple* lançou a linguagem *Swift* para substituir o *Objective-C* em 2014, porém os mesmos ainda continuam tendo suporte a aplicações que foram escritas na linguagem antiga. Isso significa que essa linguagem pode ser usada como uma linguagem intermediaria para traduções de código *back-end* entre as plataformas.

3.2.3.2. SWIFTIFY

O *Swifify* é uma ferramenta de conversão web que possui uma versão gratuita, mais limitada, tanto quanto versões pagas que permitem maiores recursos de conversão (*Swifify*,2018).

Para se converter um código não é necessário cadastro ou qualquer tipo de registro, mas para se converter arquivos de projeto ou arquivos maiores que 1kB (Kilobite) é necessário um cadastro e de estar logado no site.

4. DESENVOLVIMENTO E RESULTADOS

Para que fosse possível comparar o desenvolvimento nas duas plataformas o projeto foi construído visando encaixa-lo como uma parte de outro sistema existente da UNITINS, no esquema da figura 6 é possível visualizar que o aplicativo consumirá os dados de outro sistema hospedado na UNITINS através de conexão pela internet, para que assim os acadêmicos possam ter acesso às informações propostas.

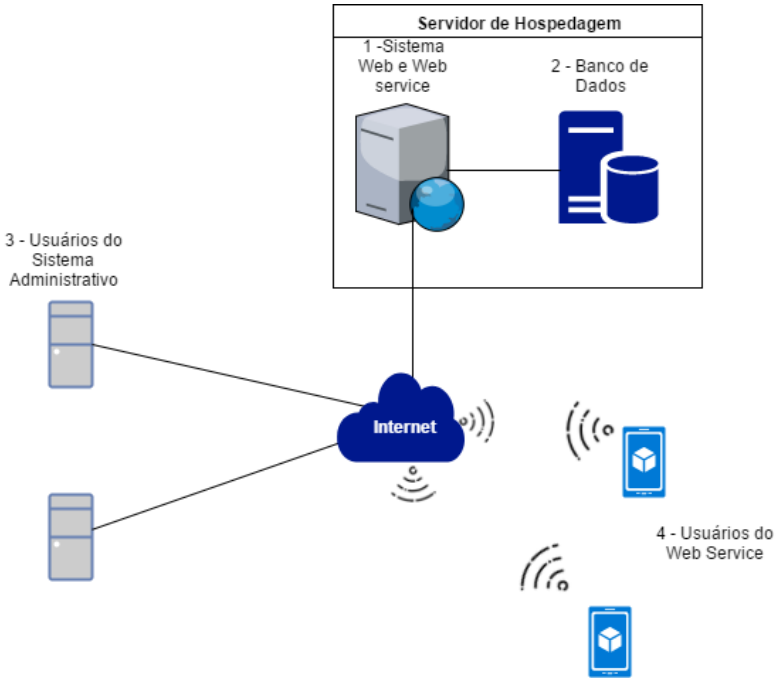


Figura 6: Arquitetura geral do sistema.

Fonte: autor (2018)

4.1. MODELAGEM

Primeiro foram modelados os casos de uso, que são um diagrama comportamental dos atores (usuários) do sistema servindo para representar como os casos de uso interagem entre si no sistema e com os usuários (atores), ou seja, como as funcionalidades se relacionarão umas com as outras e como serão utilizadas pelo usuário, durante o uso do sistema (VENTURA, 2016).

Conforme a figura 7, o usuário irá interagir com o sistema para realizar as consultas de horários, campus e obter um detalhamento sobre uma determinada alocação.

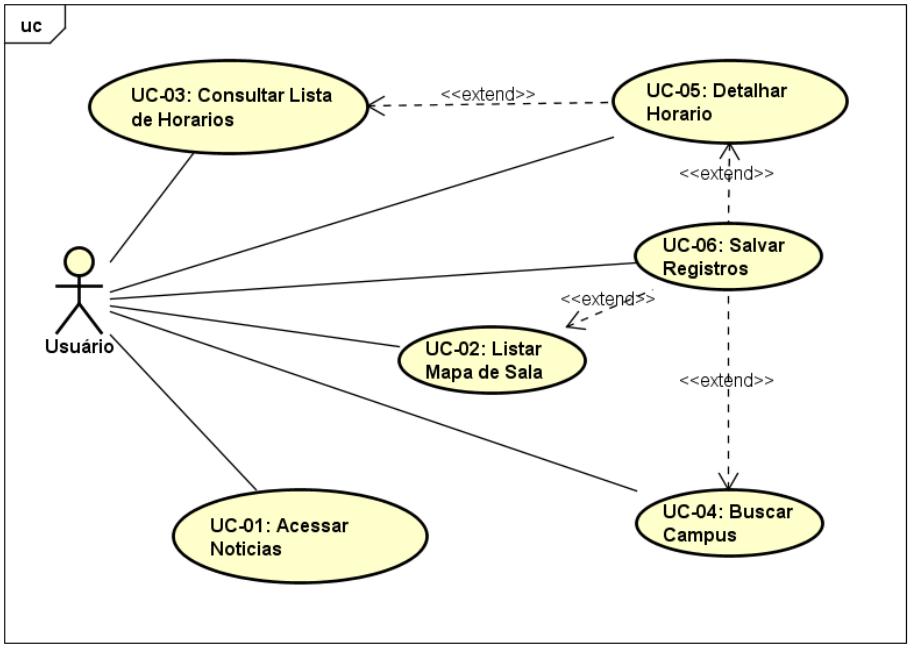


Figura 7: Diagrama de Casos de uso do APP e seus atores.

Fonte: elaborado pelo autor utilizando o software *Astah* (2018)

Após os casos de uso, foram levantados os requisitos funcionais e não-funcionais, com um requisito funcional pode ser descrito como a requisição de uma função que um *software* deverá atender/realizar (VENTURA, 2016).

Ou seja, é a exigência e a necessidade que um *software* deverá materializar para o seu usuário. Os requisitos funcionais do sistema estão na tabela 1, que possui vínculo com os casos de uso citados na figura 08.

Tabela 1: Requisitos Funcionais do APP

Requisito	**Caso de Uso Vinculado**
RF01: O app deve permitir ao usuário consultar uma lista de horários com as alocações de sala obtidas do sistema.	UC-02
RF02: O app deve salvar os dados mais recentes de alocação.	UC-06
RF03: O app deve permitir ao usuário obter um quadro personalizado sobre a alocação de salas da universidade.	UC-03
RF04: O app deve permitir ao usuário listar a alocação de salas da universidade	UC-02
RF05: O app deve permitir ao usuário buscar o endereço do campus.	UC-04
RF06: O app deve permitir ao usuário acessar as notícias da UNITINS.	UC-01

Fonte: autor (2018).

Requisitos não funcionais são as não funcionalidades do sistema que precisam ser realizadas para que o *software* atenda seu propósito, conforme explicado na tabela 2 eles atendem aos requisitos do sistema que não se referem a funcionalidades do negócio, mas que fazem parte do escopo do sistema, podendo ou não estar associado a um Requisito Funcional (VENTURA,2016).

Tabela 2: Requisitos Não Funcionais do Sistema

Requisito
RNF01: O app deve ser compatível com o Android 4.0 e iOS 5.0 ou superior.
RNF02: O app deve possuir acesso a redes TCP/IP para acesso ao *web servic*e de alocação e notícias.
RNF03: O app deve possuir um banco de dados local *SQLite* para armazenar informações relevantes.
RNF04: Integração pela internet com o sistema de alocação da Unitins via *web service* através da *internet*.
RNF05: A integridade dos dados deve ser garantida para evitar que os mesmos não sejam perdidos.

Fonte: autor (2018).

Com o objetivo de armazenar as informações mais importantes que vem do sistema hospedado foi modelado um banco de dados que se encontra no apêndice 1, a ideia é manter as informações para, caso não haja conexão com a internet, o usuário conseguir visualizar as informações desejadas,

Para exemplificar a estrutura do aplicativo foi elaborado um diagrama de classes que se encontra no apêndice 2, esse diagrama é a descrição das informações e estruturas usadas pelo aplicativo internamente, mostrando a comunicação com seus usuários, descrevendo as informações sem referência a qualquer implementação específica (VENTURA, 2016).

As telas foram desenhadas utilizando o *software* Pencil™®, sendo o conceito de interação de interface com o usuário e foram utilizadas como base para as telas de ambos os aplicativos desenvolvidos.

A figura 8 mostra como é a tela de início carregada com as informações básicas e instruções de uso para o usuário e a tela de informações sobre o desenvolvimento do aplicativo e o seu objetivo. Essas telas têm o objetivo apenas de informar o usuário sobre o uso do aplicativo e sobre o desenvolvedor.

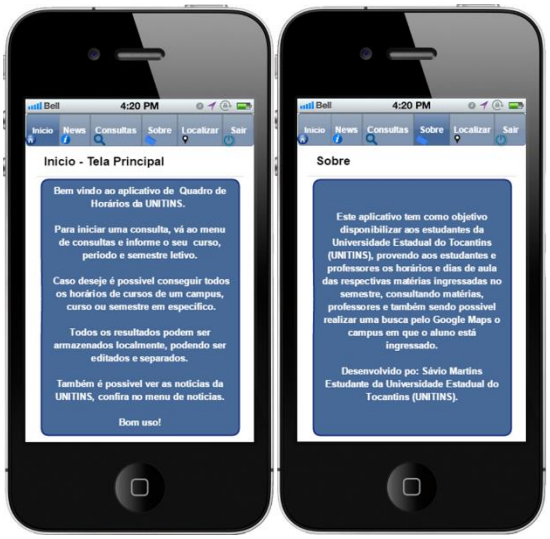


Figura 8: Telas de Início e Sobre.

Fonte: elaborado pelo autor utilizando o *software Pencil* (2018)

A figura 9 mostra como funcionará a parte de notícias, aonde o usuário irá obter uma listagem diretamente da UNITINS pela APP sem precisar abrir o seu navegador, ao clicar no título da notícia listada, o usuário será redirecionado para uma tela aonde será exibido o conteúdo da notícia.

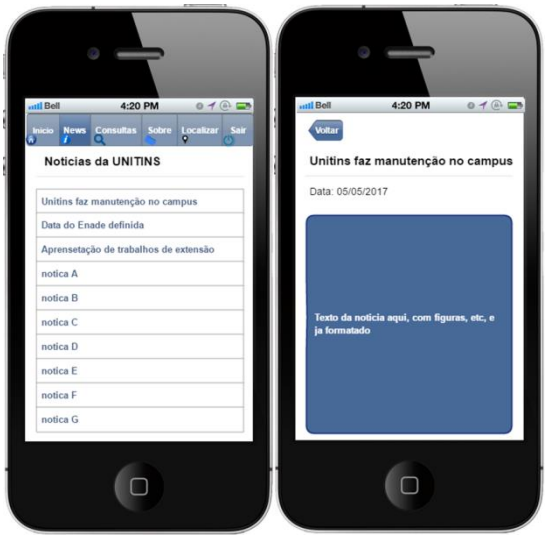


Figura 9: Telas de Listagem de Notícias e Detalhes de Noticia.

Fonte: elaborado pelo autor utilizando o *software Pencil* (2018)

A figura 10 mostra como funcionará a parte de alocações, aonde o usuário irá obter uma listagem das salas que estão alocadas no semestre selecionado, pois ao clicar no botão de pesquisa, o usuário será redirecionado para a tela de pesquisa aonde o mesmo poderá escolher os parâmetros desejados, diminuindo a listagem.

Figura 10: Telas de Listagem de Alocação e Pesquisa de Alocação.

Fonte: elaborado pelo autor utilizando o *software Pencil* (2018)

A figura 11 mostra como funcionará a listagem de alocações após a pesquisa citada na figura anterior, deixando a tela com informações de maneira mais resumida e dinâmica para o usurário, com a tela de detalhes do horário como serão exibidas as informações da alocação de maneira detalhada para o usuário.

Figura 11: Telas de Listagem de Alocação Resumida e Detalhes de uma Alocação.

Fonte: elaborado pelo autor utilizando o software *Pencil* (2018)

Por último a figura 12 mostra o modulo de localização, aonde o usuário poderá localizar o campus através de um mapa por coordenadas geográficas e a tela de sair do aplicativo.

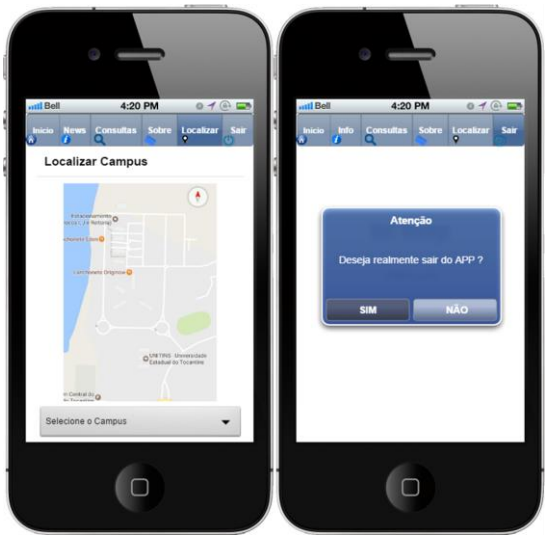


Figura 12: Telas de Localização por Mapa e Sair.

Fonte: elaborado pelo autor utilizando o software Pencil (2018)

4.2. DESENVOLVIMENTO NO ANDROID

4.2.1. ARQUITEURA DO PROJETO

Conforme citado na Metodologia toda a construção do projeto no Android foi baseada no padrão MVC citado na seção 5.5 com pequenas adaptações, o que na prática se resumiu ao projeto sendo dividido todo em pacotes com funcionalidades distintas.

A figura 13 mostra como esses pacotes foram organizados dentro do diretório *sources* que é o diretório principal de códigos fontes do Android.

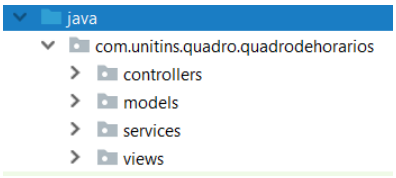


Figura 13: Estrutura MVC no Android.

Fonte: autor (2018)

As primeiras classes a serem construídas foram as do pacote *models*, com esses modelos sendo uma abstração das tabelas do banco de dados modelado na seção 7.1 possuindo os respectivos valores e métodos de acesso – *get* e *set*.

Todas as classes criadas são mostradas na figura 14, com esse pacote possuindo um subpacote para modelagem de dados do retorno do *web service* - que ainda será abordado no decorrer desta seção.

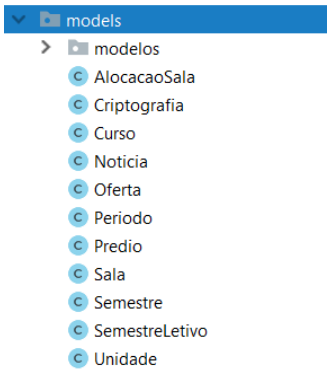


Figura 14: Pacote *Models* e suas classes no Android.

Fonte: autor (2018)

A adaptação citada no modelo MVC foi a criação de um pacote chamado *services*, esse pacote se fez necessário pois houveram classes que não se encaixavam em nenhuma outra classificação do modelo.

A figura 15 mostra como esse pacote ficou organizado, tendo um sub-pacote que foi utilizado para as classes de uso dos *web services* e que serão melhor abordadas no decorrer desta seção.

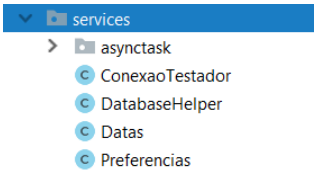


Figura 15: Pacote *services* no Android.

Fonte: autor (2018)

Uma das classes criadas nesse pacote foi criada a classe *DatabaseHelper* que serviu para acessar ao banco, criando um serviço interno de gerenciamento de banco de dados dentro no Android.

Como o banco nativo do Android é o *SQLite* citado na seção 5.4, a classe criada em *services* utilizou um script pronto que ficou alocado no diretório *assets*, sendo que no primeiro acesso ao banco do aplicativo o script é copiado para o diretório utilizado pelo APP – sendo representado por *data*/*data*/<*namespace* da aplicação>/*databases*/.

Todo esse acesso foi gerenciado pelas classes que ficaram no pacote *controllers* com todas essas classes utilizando a *DatabaseHelper* nesse serviço de acesso ao banco de dados, sendo um controlador para cada classe modelo.

Essas classes, que são mostradas na figura 16, possuem ao todo quatro métodos de acesso para cumprir esse papel, sendo dois de inserção – atualizar e inserir – e dois de

consulta – listar e buscaPorId – com exceção da AlocacaoSalaC que possuí um listar parametrizado.

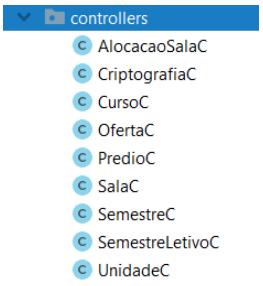


Figura 16: Pacote *controllers* no Android.

Fonte: autor (2018)

Com modelos, banco e controladores prontos foi possível criar as telas interativas do APP, porém é necessário observar que no Android a *view* se divide em três partes como mostrado na figura 17 com suas partes marcadas em vermelho, fazendo parte dela o *mainfest*, o pacote *views* e o diretório res.

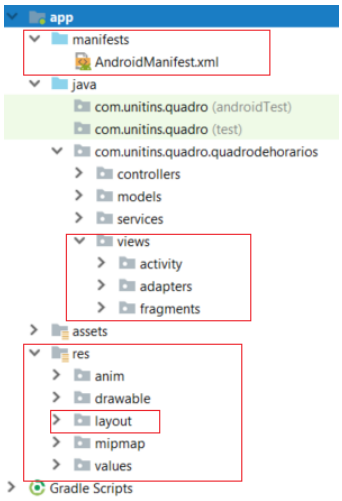


Figura 17: Distribuição da *view* no Android.

Fonte: autor (2018)

4.2.2. CONSTRUÇÃO DAS VIEWS

Ao criar uma *activity* e *fragment* são gerados dois arquivos, um é o XML do layout que fica em *layout*, o outro classe que se comunica com o código no diretório selecionado dentro de *source* em um dos sub-pacotes de *views*, e quando uma *activity* é criada ela é registrada no *mainfest*, sendo que uma delas precisa ser a principal

Para os dois arquivos se comunicarem é necessário o uso da classe R que acessa os componentes que estão em *resources* através do código fonte, uma classe reconhece o seu layout através da classe R e vice versa através do método *onCreate* da classe.

As *views* se dividem em três grupos como mostra a figura 18, o primeiro é o das *activitys* que é composto pela tela principal e as telas de detalhes do APP, agindo conforme o ciclo de vida citado na seção 5.1.2.

A tela principal foi feita utilizando um recurso de *tab bar* que necessita que outras telas sejam ligadas a ela, essas telas são as *fragments* – fragmentos de *activitys* – que ficam no grupo *fragments* e são a maioria das telas do aplicativo.

Essas *fragments* são infladas junto com os seus componentes para aparecer na *activity* principal conforme durante a execução possuíndo um funcionamento mais interno que o de uma *activity*, com o último grupo sendo o dos *adapters*, que serão abordadas melhor junto com os componentes de interface.

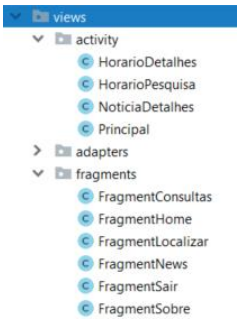


Figura 18: Pacote *views* e seus grupos no Android.

Fonte: autor (2018)

Como mostra a figura 19, todos os componentes visuais no Android podem ser adicionados estilo “arraste e solte”, como também é permitido mexer diretamente as linhas do XML, com cada componente pode ser selecionado por grupos no *Pallete* (1) pelo e organizados de maneira hierárquica pelo *Component Tree* (2).

Todos esses componentes precisam estar ligados a um estilo de *layout* principal sendo necessário sempre escolher um tipo de layout para a *activity* ou *fragment*, já as configurações ficam no menu *Attributes* (3) que permite selecionar os atributos sendo possível pesquisar uma configuração pelo nome clicando na lupa que fica nesse menu.

Ao centro pode ser exibida a tela com os componentes e também o *blueprint* (4), que é o esquema da tela selecionada mostrando como estão organizados os componentes em relação ao layout.

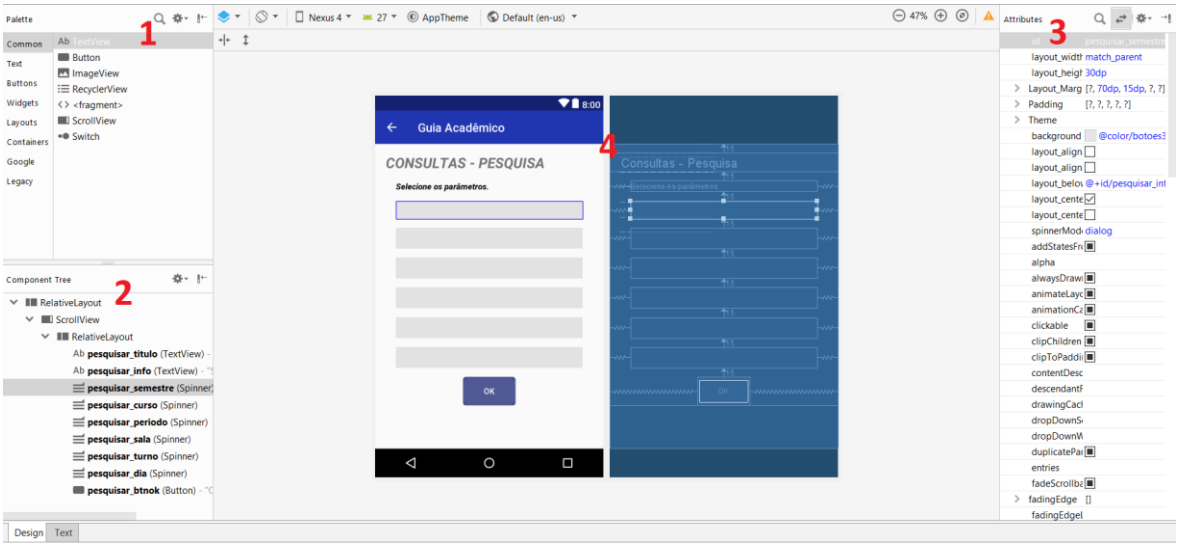


Figura 19: Edição no Android Studio.

Fonte: autor (2018)

4.2.2.1. USO DE COMPONENTES

Os componentes adicionados na tela que foram manipulados durante a execução – como botões e listas – precisaram de ter um ID vinculado, pois quando um componente é referenciado na classe controladora da *view* através da classe R é preciso um identificador.

Sem um identificador trabalhar com os componentes visuais no Android pode se tornar uma tarefa árdua e inviabilizar o desenvolvimento, com essas referências ocorrendo no *onCreate* com o identificador – o mesmo vale para as *fragments*.

De uma grande lista de componentes disponíveis no Android apenas alguns deles foram utilizados, podendo ser citados quatro utilizados no trabalho que possuem um comportamento diferenciado: botões, *list view*, *Spinner* e *web view*.

Um botão executa um ação ao se clicado durante a execução, para isso o botão que está na *activity* ou *fragment* precisa saber qual é o evento que lhe pertence.

No Android isso ocorre via protocolo, uma interface *onClickListener* que "escuta" as ações do botão, sendo recomendada para *fragments* ou em caso de vários botões em uma mesma tela.

A *list view* é o componente que foi responsável por exibir as informações de notícias e horários exibindo os itens de maneira vertical e rolável, sendo esse um dos componentes mais utilizados em aplicativos.

Para manipular uma lista foi necessária utilizar uma fonte de dados – um *ArrayList* – e um *adapter*, sendo possível realizar a manipulação – como listagem e clique – somente através dele e de seus métodos.

Além disso a lista precisa de uma célula personalizada para exibir as informações de maneira adequada, o layout dessa célula precisa ser criado separadamente e ter uma classe no sub-pacote *adapter* dentro de *views* como mostrado na figura 20.

Essas células precisam ser infladas e serem recicláveis via programação para não extrapolar a memória do *smartphone*.

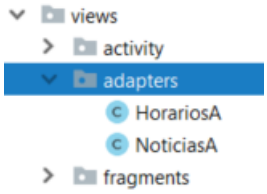


Figura 20: Célula reciclável no Android.

Fonte: autor (2018)

O *spinner* se assemelha a um *combo box* e funciona de maneira similar ao *list view*, exceto pela necessidade de um componente personalizado o *spinner* necessita também uma fonte de dados – *data source* – e de um *adapter* para controlar suas ações, incluindo a seleção de valores.

A *WebView* foi componente responsável por processar e exibir um texto HTML (*HyperText Markup Language*) dentro do aplicativo sem ter que chamar o navegador padrão do aparelho, abrindo links externos com o navegador padrão através de uma classe auxiliar que gerenciou o comportamento da *webview*.

4.2.2.2. RELACIONAMENTO

Após as *activitys* e *fragments* ficarem prontas com os seus componentes já configurados, foi necessário definir o relacionamento entre telas.

Toda a tela principal do aplicativo em Android se baseou em uma *tab bar*, com ícones dos menus e a possibilidade de se deslizar o menu com o dedo, todas as *fragments* tiveram que ser conectadas através da classe dessa tela via programação.

Além das telas relacionadas com a *tab bar* o Android utilizou dois tipos de relacionamento, o primeiro foi a *intent*, que é um objeto que pode ser usado para solicitar outro componente do aplicativo - sendo esse componente uma nova *activity*.

Uma *intent* pode passar informações entre as *activitys*, para isso foi necessário utilizar um dicionário de dados chamado de *bundle*, que é uma classe mapeada em chave/valor, funcionando como um container de transporte de dados primitivos.

Para retornar a tela que a chamou uma *intent* possuí um recurso chamado *intentResult*, que é uma *intent* de finalização de uma *activity*, com ela foi possível passar

informações como o resultado de alguma instrução realizada na *activity* chamada: dessa forma a *activity* que recebeu esse resultado pode realizar também alguma operação com base nos dados passados pelo resultado,

O outro relacionamento utilizado foi a *Parent Activity*, que funciona como um *up navigation* (navegação para cima em tradução livre) com uma seta na barra superior da *activity*.

Esse tipo de relacionamento foi feito na declaração de uma *activity* no arquivo *mainfest*, fazendo o sistema entender qual *activity* é o pai apropriado para cada *activity* filha, esse recurso permite que o sistema facilite esse padrão de navegação, pois o mesmo pode determinar a *activity* pai lógica somente partir do arquivo de *mainfest*.

4.2.3. USO DO WEB SERVICE

Com toda o aplicativo estruturado, com modelos, controladores, *views* e relacionamento de telas, foi possível utilizar realizar a função principal para o qual o projeto do *Android* foi designado: disponibilizar os horários do semestre, localização do campus e as notícias da UNITINS, conforme citado na seção 7.1.

Isso só foi possível com o uso dos *web services* disponibilizados do sistema de Alocação de Salas da Administração da UNITINS, pois todas as informações necessárias para se cumprir esse objetivo são todas disponibilizadas via *web service*.

Foi necessário primeiro adicionar as permissões para que o aplicativo pudesse acessar a internet, conforme mostrado na figura 21, sendo adicionadas no início do arquivo *mainfest*, antes de se declarar as *activitys*.

```
<!-- Permissão para usar a Internet -->
<uses-permission android:name="android.permission.INTERNET" />
<!-- Permissão para fazer Teste de Conexão -->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<!-- Permissão para usar o Localizador -->
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name="android.hardware.location.gps"/>
<!-- habilitar opengl -->
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

Figura 21: Permissões no Android.

Fonte: autor (2018)

O Android não permite que uma requisição *web* seja feita na *thread* principal do aplicativo, para isso é preciso utilizar as *AsyncTasks*, que são *threads* secundárias e assíncronas com a *thread* principal, toda requisição precisa ser feita através dessas *tasks*.

Como mostra a figura 22 as classes que fizeram a comunicação com o *web service* ficaram no pacote *services* no subpacote chamado *asynctask*.

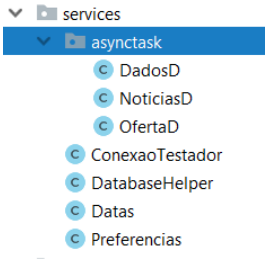


Figura 22: Célula reciclável no Android.

Fonte: autor (2018)

O *web service* foi usado em quatro momentos, o primeiro para baixar notícias, o segundo para baixar os dados de busca pela primeira vez, a terceira para realizar uma consulta e a quarta para sincronizar os dados do aplicativo com o do servidor.

Como o retorno do *web service* veio no padrão JSON, a requisição realizada utiliza a biblioteca *GSon*, uma biblioteca da própria Google para requisições nesse formato, transformando o texto recebido em objetos.

Conforme mostra a figura 23, essa biblioteca - e qualquer outra - precisa ser declarada no *dependencies* do *gradle*.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    compile 'com.android.support:design:25.3.1'
    compile 'com.android.support:support-v4:25.3.1'
    compile 'com.android.support:gridlayout-v7:25.3.1'
    compile 'com.google.android.gms:play-services-maps:10.2.1'
    testCompile 'junit:junit:4.12'
    compile 'com.google.code.gson:gson:2.8.0'
    compile 'com.stanfy:gson-xml-java:0.1.7'
    compile 'com.google.android.gms:play-services-maps:10.2.1'
}
```

Figura 23: Dependências declaradas no Gradle.

Fonte: autor (2018)

Para manipular o objeto *GSon* do retorno foi também necessário criar um modelo especifico igual ao do retorno do *web service*, esses modelos ficaram dentro do pacote *models* no sub-pacote modelo.

As classes criadas no modelo são mostradas na figura 24, sendo esses modelos compostos por atributos do retorno – um boleano, uma *string* e uma lista do modelo requisitado

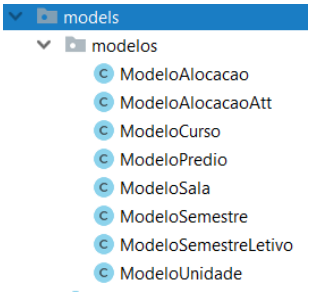


Figura 24: Modelos de retorno do *Web Service* no Android.

Fonte: autor (2018)

Para manipular o objeto *GSon* do retorno foi também necessário criar um modelo especifico igual ao do retorno do *web service*, esses modelos ficaram dentro do pacote *models* no sub-pacote modelo.

A exceção das notícias, todos os dados baixados são salvos após a execução da thread, porém por se tratar de uma tarefa assíncrona foi necessário fazer com que o aplicativo espere a os dados serem baixados.

Isso foi possível utilizando um *ProgressDialog*, essa caixa de diálogo é iniciada e finalizada junto da *task*, não importando se os dados foram baixados corretamente ou não, apenas evitando que o usuário não faça alguma ação que atrapalhe a execução da mesma.

Para evitar que o usuário não precise sempre repetir a mesma consulta quando abrir o aplicativo, os parâmetros informados na última consulta realizada são armazenados na memória interna do mesmo, essa memória é chamada de *SharedPreferences*.

O funcionamento é simples, sendo necessário importar a classe *SharedPreferences* do Android em uma classe, informar a identificação do banco utilizado e criar métodos estáticos no padrão *get* e *set*.

4.2.4. MAPA

O último componente do desenvolvimento do aplicativo foi o mapa de localização utilizado para o usuário localizar o campus através pelo GPS (*Global Positioning System*).

Para se utilizar o mapa como proposto foi necessário adicionar as permissões de localização e de *OpenGL* mostradas na figura 23, pois o dispositivo precisa acessar a localização a própria localização e renderizar o mapa utilizado dentro do aplicativo.

Além disso o aplicativo precisa de uma chave de API para acessar os servidores do Google Maps, essa chave é gratuita e pode ser utilizada por qualquer dispositivo Android que chama a API do Google Maps, para gera-la é necessário acessar Google API console no navegador e a colando no *mainfest* como mostra a figura 25.

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AIzaSyDs8R-GpFGIulXbdQNVncyfUZbXgh16Ws0" />
```

Figura 25: Chave para uso do Mapa no Android.

Fonte: autor (2018)

Com a chave assinada e as permissões o aplicativo ficou apto para utilizar esses recursos, internamente o mapa funciona com o *SupportMapFragment*, que nada mais é que um componente de mapa renderizado no próprio app, porem no aplicativo ele apenas coloca um pino no local do campus, não traçando a rota nem passando maiores informações internamente.

Isso ocorre por já existir a API de mapas do Google instalada no *Android* por padrão, portanto ao clicar no botão de localizar com o endereço selecionado, o aplicativo faz um *intent* para a API de mapas com os dados de partida e destino e deixa a cargo da API de realizar esse procedimento.

4.2.5. MENSAGENS PARA O USUÁRIO

Com todas as funcionalidades prontas, o aplicativo algumas vezes precisa comunicar o usuário o que ocorre durante a execução principalmente em casos de erro, uma das formas já foi citada que é o *ProgressDialog*, porém ele não é útil para mensagens rápidas.

Para um *feedback* dinâmico sobre uma operação com o usuário o Android possuí o a classe *Toast*, que oferece um pequeno pop-up que ocupa o espaço e desaparecem automaticamente após um tempo limite.

4.2.6. PUBLICAÇÃO

Após todo o desenvolvimento existe a possibilidade de se publicar o APP na *Google Play Store*, para isso é necessário criar um arquivo com extensão APK (*Android Application Pack*) com uma assinatura, que nada mais é que compilar o projeto e criar um instalador.

Esse procedimento é feito pelo próprio Android Studio, sendo necessário gerar uma chave quando o mesmo for criado, é preciso ir em *Build > Generate Signed APK*... e

selecionar a opção "*Create a new key*", preenchendo as informações que forem requisitadas. Essa chave é de extrema importância pois caso ela seja perdida não será possível atualizar o aplicativo na loja.

Na publicação, além de ter uma conta de desenvolvedor ativa, é preciso ficar atento a pelo menos 4 passos na publicação, que são o Versões de APPs, Detalhes do APP, Classificação de Conteúdo e Preços e distribuição.

Ou seja, é preciso preencher as informações necessárias, informando detalhes como a descrição e imagem das telas, preenchendo o formulário de classificação de conteúdo conforme as opções passadas e também se o aplicativo é pago ou tem alguma forma de propaganda.

4.3. DESENVOLVIMENTO NO iOS

4.3.1. ARQUITETURA DO PROJETO

A construção do projeto em iOS seguiu os mesmos passos da seção anterior, sendo baseado no padrão MVC conforme citado na Metodologia e na seção 5.5 com pequenas adaptações.

A figura 26 mostra como o projeto ficou organizado por grupos dentro da hierarquia de diretórios do projeto, não existindo um diretório principal para códigos fonte e sim o diretório principal do projeto.

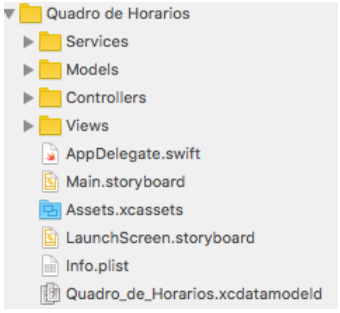


Figura 26: Estrutura MVC no iOS.

Fonte: autor (2018)

As primeiras classes a serem construídas foram as do grupo *models*, com esses modelos sendo uma abstração das tabelas do banco de dados modelado na seção 7.1, com a diferença de no iOS eles possuírem apenas os atributos sem a necessidade de métodos de acesso.

Todas as classes criadas são mostradas na figura 27, não sendo necessário um subgrupo para comportar novas classes como no Android.

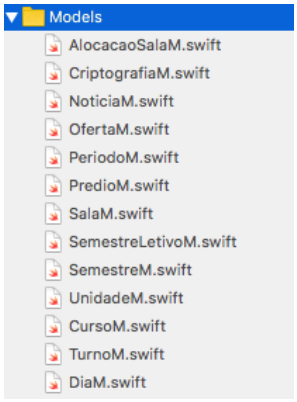


Figura 27: Grupo *Models* e suas classes no iOS.

Fonte: autor (2018)

A adaptação citada no modelo MVC no iOS seguiu o mesmo caminho com a criação de um grupo chamado *services*, da mesma forma houveram classes que não se encaixavam em nenhuma outra classificação do modelo.

A figura 28 mostra como esse grupo ficou organizado, tendo um sub-grupo que foi utilizado para as classes de uso dos *web services* e que serão melhor abordadas na seção 7.3.3.

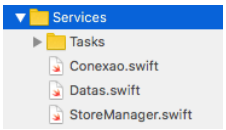


Figura 28: Grupo *Services* e suas classes no iOS.

Fonte: autor (2018)

Uma diferença no grupo *services* foi a ausência do banco de dados, pois no iOS ele é gerenciado pelo *Core Data*, um framework criado pela Apple para gerenciar o banco em forma de entidades utilizando o *SQLite* da seção 5.4 como base.

Para utiliza-lo foi necessário habilita-lo ao criar o projeto, sendo possível habilitar adicionando os *delegates* no arquivo principal do projeto chamado *AppDelegate*.

O c*ore data* trabalha com um arquivo *xcdatamodeld*, sendo criadas nesse arquivo como entidades que serão trabalhadas, como mostra a figura 29 seu funcionamento é similar ao de uma ferramenta de modelagem SGBD como o *MySql Workbench*, aonde é possível acrescentar relacionamentos e os atributos necessários.

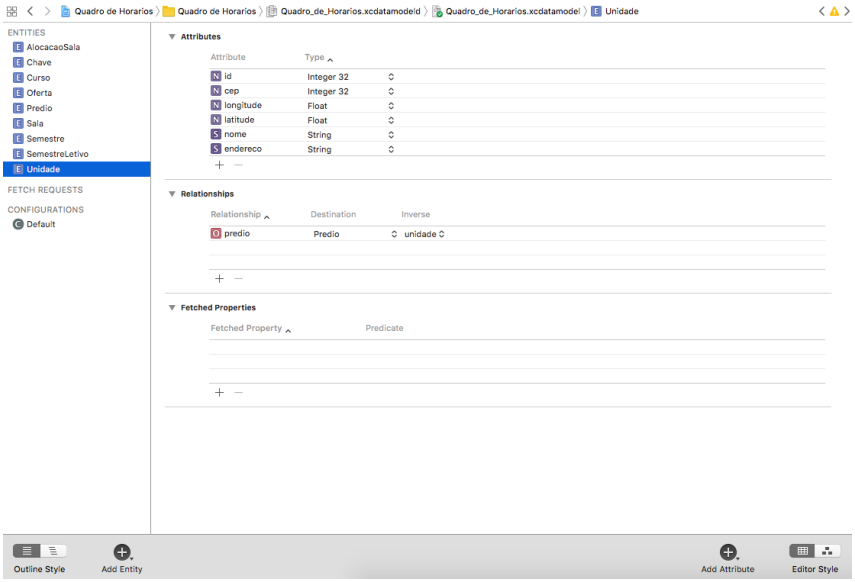


Figura 29: Entidades do *Core Data*.

Fonte: autor (2018)

Mesmo com o *core data* criando as classes automaticamente ao criar uma entidade, o acesso ao banco é gerenciado pelas classes que ficam no grupo *controllers*, isso se fez necessário por causa do uso do *web service* que será abordado no decorrer dessa seção.

As classes controladoras são mostradas na figura 30, possuindo ao todo cinco métodos de para gerenciar o banco junto às entidades do core data, sendo dois de inserção – atualizar e inserir – e três de consulta – listar, buscaPorId e buscaPorIdCD – tendo em todas elas o vínculo com a classe Contexto para auxiliar no uso das entidades Core Data.

A única exceção é o controlador AlocacaoSalaCD que possuí seis métodos, com um listar parametrizado para retornar a quantidade de dados que o usuário pesquisou.

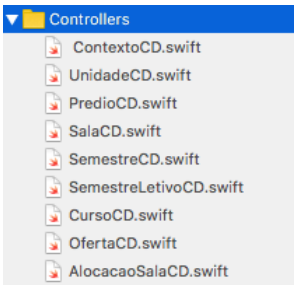


Figura 30: Grupo *controllers* no iOS.

Fonte: autor (2018)

Com estas três partes criadas foi possível iniciar a parte de telas do iOS, sendo a organização das *views* do iOS mais simples que a do Android, pois no iOS ela se divide em duas partes como mostrado na figura 31 com suas partes marcadas em vermelho, fazendo

parte dela o grupo de códigos *views* e os três arquivos de projeto – duas *storyboards* e o *assets*.

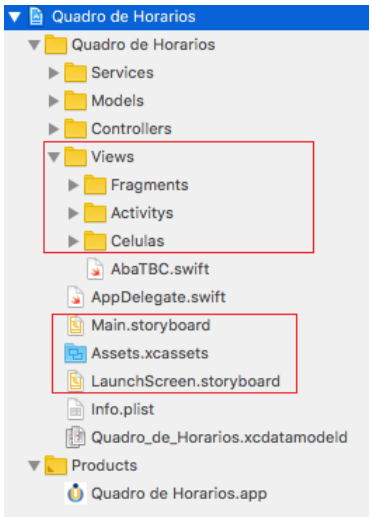


Figura 31: Distribuição da view no iOS.

Fonte: autor (2018)

4.3.2. CONSTRUÇÃO DAS VIEWS

Uma *view controller* é criada sempre eu duas partes, a primeira com o código Swift no grupo designado utilizando um dos *templates coccoa touch*, todos herdando de *view controller*, e a segunda é colocando uma *view controller* – também chamada de cena - do tipo correspondente dentro da *storyboard*.

Imagens ou ícones para serem utilizados no iOS são colocados no *assets.xcassets*, esse arquivo distribui todas as imagens para serem utilizadas durante a construção das *views*, incluindo o próprio ícone que foi feito utilizando um aplicativo.

Todas os layouts ficam dentro da *storyboard*, que é o que permite lidar com a construção e com a hierarquia de diversas *views* incluindo navegação entre elas sendo a *view controller* sua unidade mínima, precisando que uma delas seja indicada com a de entrada.

A única exceção é a *launch screen* que é colocada sozinha no *LauncScreen.storyboard*, sendo ela responsável apenas uma tela de entrada, sendo obrigatório colocar algum elemento visual nessa tela.

Outro detalhe é que a *storyboard* funciona por cima de um XML, mas é feita para que não seja necessário editar diretamente o XML, muito diferente do que ocorre no Android por possuir um número alto e mal organizado de opções nas configurações dos componentes e que torna necessário intervenções diretas no XML do arquivo.

Para a classe e a cena – como é chamada uma *view controller* na *storyboard* – se comunicarem é necessário utilizar a própria *stortyboard*, atráves de uma configuração pelo menu *Indentity Inspector* no campo *class* com o nome do arquivo, essas classes se dividem em três grupos como mostra a figura 32, o primeiro se chama *activitys* que é composto pelas *view controllers* que não estão vinculadas a tela principal.

Como cena principal foi utilizada a *tab bar view controller* que é uma variável de *view controller* que faz a navegação em abas, ligando as demais cenas via segue, não sendo necessária linha de código alguma, diferentemente do que ocorre no Android aonde é necessário programar toda a *tab bar*.

As cenas que são ligadas e possuem classes controladoras ficam no grupo *fragments* e também são a maioria das telas do aplicativo, diferente do Android essas telas não são infladas e são apenas chamadas quando as mesmas são selecionadas na *tab bar*.

O último grupo são as células, classes que possuem os atributos das células das *list view controllers* e serão abordadas junto com os componentes de interface. Esses nomes foram adotados nos grupos apenas para fins comparativos, uma vez que o iOS não usa tais componentes.

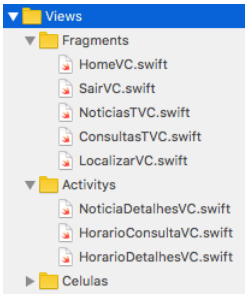


Figura 32: Grupo *views* e seus subgrupos no iOS.

Fonte: autor (2018)

Todos os componentes visuais no iOS podem também ser adicionados ao estilo "arraste e solte", não sendo necessário alterar o XML diretamente – as vezes, sequer é permitido acessa-lo – com cada componente podendo ser selecionado pelo *Object Library* (1) não sendo necessário que esses objetos estejam vinculados a uma layout como no Android.

As configurações ficam no *Identity Inspector* (2) podendo ser selecionadas outras configurações pelos ícones dependendo do que precisa ser alterado, já hierarquia dos

componentes de uma cena é informada no *Document Outline* (3) senda essa hierarquia meramente informativa quanto aos componentes que estão presentes,

Ao centro é fica o fluxo da *storyboard* com todas as cenas (4), incluindo os relacionamentos entre as telas conforme mostrado na figura 33.

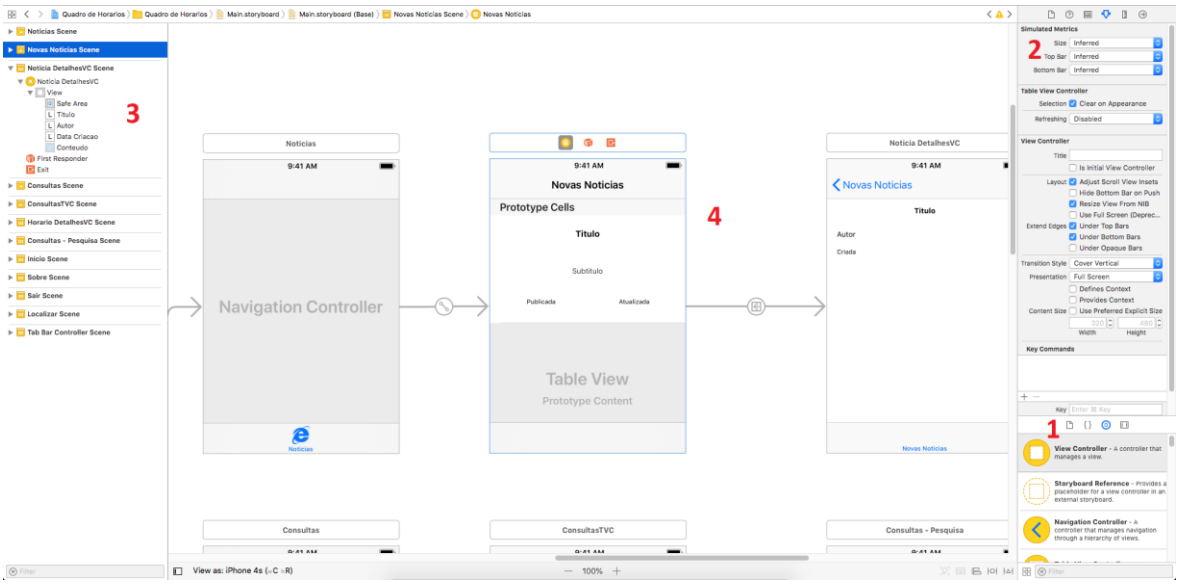


Figura 33: *Storyboard* no XCode.

Fonte: autor (2018)

4.3.2.1. USO DE COMPONENTES

Os componentes adicionados em uma tela que foram usados durante a execução – como botões e listas – precisaram de ter um vínculo com a classe responsável pela *view*, classe essa que teve o vínculo mostrado com a *storyboard* na seção anterior.

Nesse caso não foram necessários identificadores e sim das variáveis de acesso *IBOutlets* e *IBActions*, muito diferente do que ocorre no Android com o uso da classe R para todos os componentes.

Uma *IBOutlet* foi usada quando houve necessidade de enviar uma informação do código para uma cena na *storyboard*, enquanto o *IBAction* foi usado para fazer o inverso levando uma ação da *storyboard* para o código como nos protocolos do Android.

Essas variáveis foram conectadas conforme mostra a figura 34, mostrando mais uma vez como o uso da *storyboard* se difere do uso da variável R e dos protocolos utilizados pelo Android.

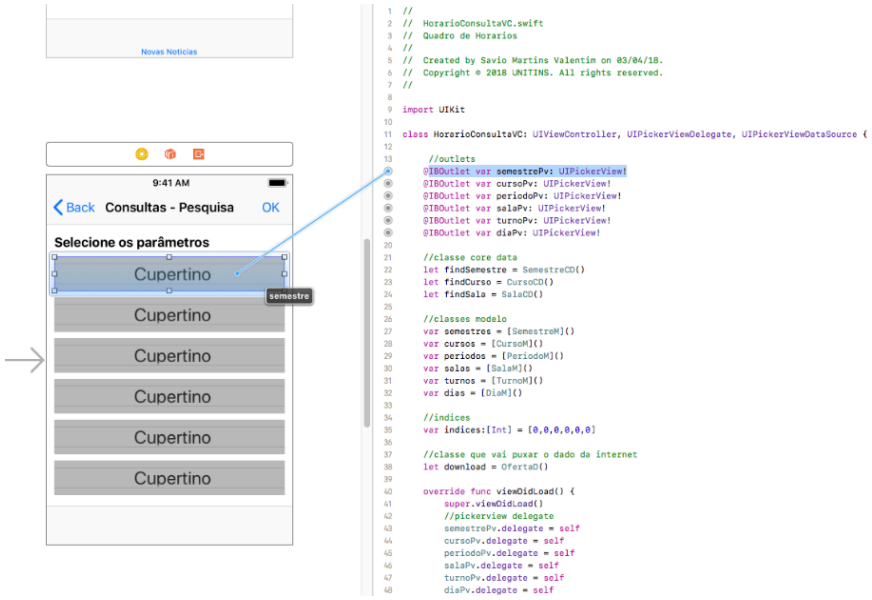


Figura 34: *IBOutlet* se conectando a classe no iOS.

Fonte: autor (2018)

Da mesma forma que no Android, o iOS possuí uma grande lista de componentes e apenas alguns deles foram utilizados, quatro deles no trabalho que possuem um comportamento diferenciado: botões, listas, *picker view* e *web view*.

Os botões foram usados para executar ações ao serem clicados durante a execução, para isso o botão precisou ser conectado a classe usando o *IBAction* sendo necessário uma variável para cada botão utilizado, como os botões não mudaram de aparência durante o uso não se fez necessário o uso das *IBOutlets*.

As listas foram responsáveis por exibir as notícias e os horários baixados com pequenas informações – como título, nome ou data de atualização – todas organizadas de maneira vertical e sendo manipuladas tanto para cima quanto para baixo.

Para manipular uma lista no iOS foi preciso uma fonte de dados – uma lista de objetos do modelo como Noticia ou Alocação – e dos métodos de *delegate* do iOS, esses *delegates* se assemelham ao *Adapter* do Android, com a diferença de que é necessário criar um método para cada função que se deseja gerenciar – como o clique na célula ou a contagem de células existentes.

Diferente do Android essas listas não são infladas, mas sim criadas conforme as mesmas estão visíveis ao usuário com o próprio *delegate* responsável pela criação da célula a tornando reutilizável.

Essas células já vieram na própria *list view controller* necessitando apenas de uma classe se comunicando com a *storyboard* e dos componentes a serem exibidos, com a

manipulação sendo encarregada dos *delegates* da lista a qual a célula pertence como mostra a figura 35.

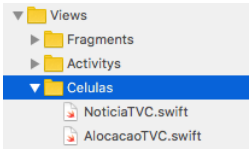


Figura 35: Células utilizadas no iOS.

Fonte: autor (2018)

O *picker view* se assemelha a um combo box e, assim como no Android, funciona de maneira semelhante a uma lista utilizando um *data source* e métodos *delegate* para controlar o seu comportamento, incluindo a mudança de fonte e seleção de valores e diferente do *spinner* o *picker view* não abre um menu *dropdown* e sim tem um aspecto de rolagem, o fazendo parecer um seletor de datas.

Por último o *WebView* que foi componente responsável por processar e exibir os textos HTML dentro do aplicativo sem ter que chamar um navegador padrão do aparelho, diferente do *Android* no iOS ele não necessita de uma classe extra e sim dos métodos *delegate* e do *IBOutlet*.

4.3.2.2. RELACIONAMENTO

Após todas as cenas ficarem prontas na *storyboard* com os seus componentes já configurados foi necessário definir o relacionamento entre elas, no iOS esse relacionamento acontece pelo componente chamado Segue, tendo como função gerenciar e suportar as transições visuais padrão disponíveis no *UIKit*.

No caso da tela principal, como já foi citado, ela é uma *Tab Bar View Controller*, o que significa que todos os componentes ligados a ela serão exibidos em forma de aba da mesma forma que foi feito no Android.

A diferença é que o relacionamento é mais simples e não necessita de uma linha de código sequer, utilizando apenas as *relationship segues* da *storyboard* sem a necessidade de programar esses relacionamentos.

Ao todo foram utilizadas três tipos de segue contando com a *relationship segue* (1) já citada, com a primeira do tipo *show detail* (2) aonde caso o app estiver no modo "*Master / Detail*" a *scene* será exibida na área de detalhes e a segunda do tipo *custom* (3) aonde a iteração pode ser controlada via programação com a figura 36 exemplificando esse uso.

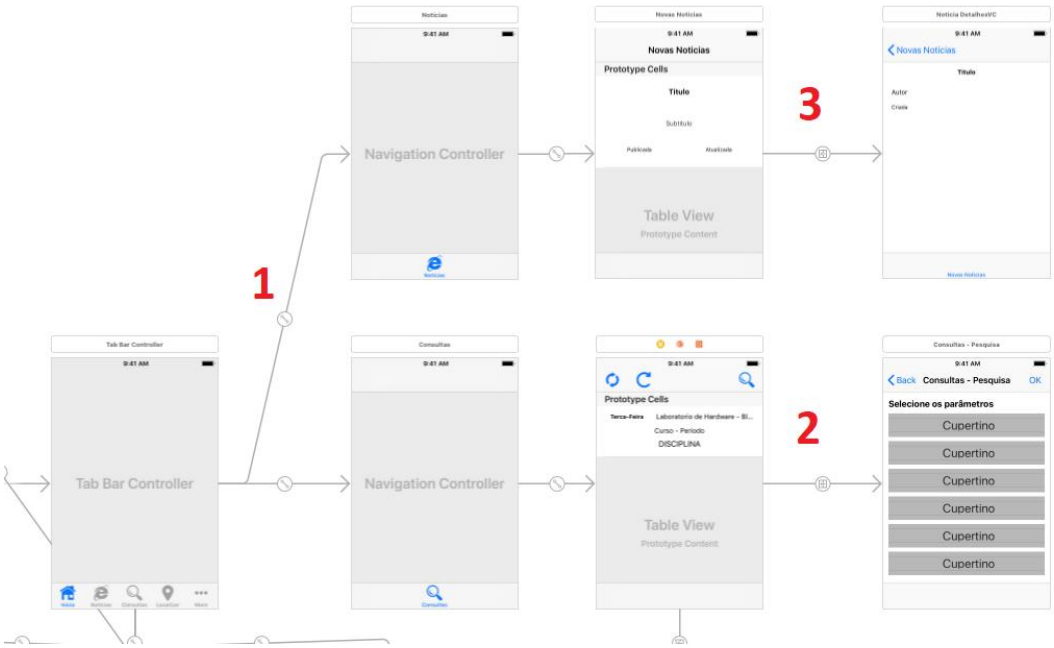


Figura 36: Segues na *storyboard* do iOS.

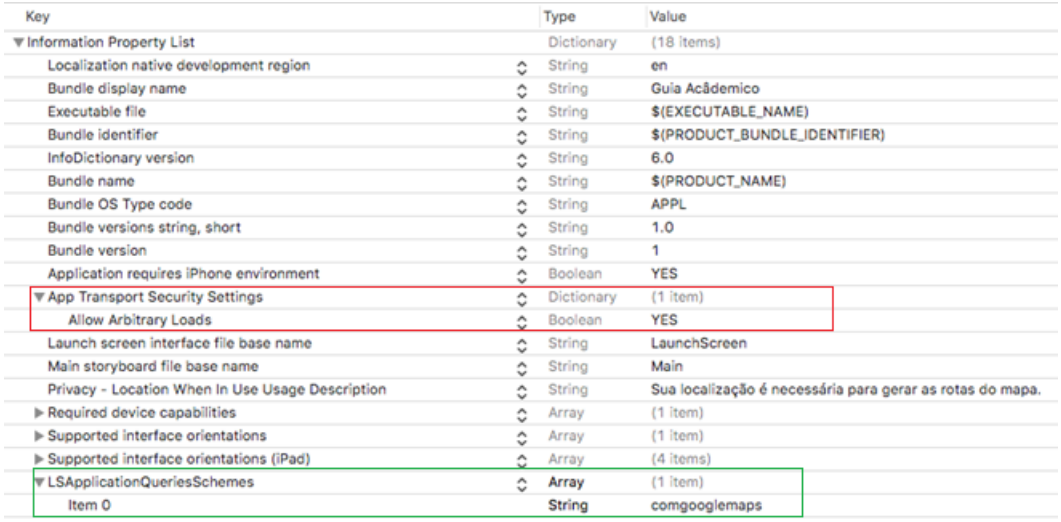
Fonte: autor (2018)

No caso da *custom segue* e *perform segue* foi necessário inserir um identificador, pois podem haver várias segues saindo de uma mesma cena. Tanto no evento de chamar quanto no evento de retorno, o identificador aponta qual a próxima instrução a ser executada.

Este tipo de segue foi utilizada para chamar a cena de detalhamento tanto de notícias quanto de horários, passando as informações necessárias a serem exibidas na *view controller*.

4.3.3. USO DO WEB SERVICE

Para se utilizar o *web service* no iOS também foi necessário adicionar as permissões no aplicativo para ter permissão para acessar a internet, essa permissão é concedida no arquivo *info.plist* como mostra na figura 37, sendo necessário adicionar uma nova linha com as permissões necessárias (em vermelho) – o mesmo vale para os mapas que serão abordados na seção seguinte (em verde).



Key	Type	Value
Information Property List	Dictionary	(18 items)
Localization native development region	String	en
Bundle display name	String	Guia Acâdemico
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Privacy - Location When In Use Usage Description	String	Sua localização é necessária para gerar as rotas do mapa.
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(1 item)
Supported interface orientations (iPad)	Array	(4 items)
LSApplicationQueriesSchemes	Array	(1 item)
Item 0	String	comgooglemaps

Figura 37: Permissões no iOS.

Fonte: autor (2018)

Assim como o Android, o iOS não permite que uma requisição *web* seja feita na *thread* principal do aplicativo, para isso foi preciso utilizar uma thread secundaria através de um recurso chamado *datatask*.

A operação ocorreu de forma diferente do Android, o *datatask* não é uma classe com métodos bem definidos como a *AsyncTask*, ele funciona como um empilhamento de processos chamados pela *URLSession* do iOS, enquanto no Android a *thread* é aberta e a requisição executada e como mostra a figura 38 essas classes ficaram no grupo *task* dentro de *services*.

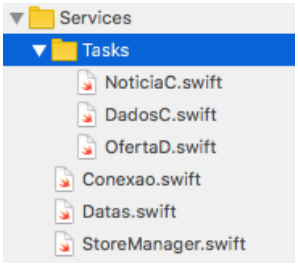


Figura 38: *Tasks* no iOS.

Fonte: autor (2018)

O *web service* também foi usado em quatro momentos: baixar notícias, baixar os dados de busca pela primeira vez, realizar uma consulta sincronizar os dados do aplicativo com o do servidor.

Para tratar o retorno não foi necessário importar nenhuma biblioteca como feito no Android nem necessitou de um modelo extra, para isso foi utilizada o *JSONSerialization* do iOS, transformando todo o retorno em um *Array* de *strings*, os modelos criados foram

muito utilizados nesse processo, pois foi necessário transformar os dados para seus respectivos tipos antes de realizar a persistência com o core data.

A exceção das notícias, todos os dados baixados também foram salvos após a execução da *datatask*, porém por se tratar de uma tarefa assíncrona foi necessário fazer com que o aplicativo espere a os dados serem baixados.

O iOS não possuí o *progress dialog*, portanto visualmente não foi possível um elemento indicando progresso enquanto o aplicativo baixa e carrega as informações dos *web services* utilizados.

Da mesma forma que no Android o iOS possuí uma memória interna para armazenar tipos primitivos de dados, esse recurso é chamado de *StoredProprietes* e foi usado com o intuito de armazenar as consultas realizadas pelo usuário.

O funcionamento é simples, sendo necessário criar uma classe – que está em *services* - com funções estáticas que gravam e retornam as informações, sendo necessário apenas informar na entrada as informações salvas e tratar o retorno ao chama-las.

4.3.4. MAPA

O último componente do desenvolvimento do aplicativo foi o mapa de localização utilizado para o usuário localizar o campus através pelo GPS, com as permissões necessárias sendo adicionadas como mostrado figura 37 para permitir acessar a localização correta.

Diferente do Android não é necessário uma chave assinada para o acesso aos mapas, mas internamente o processo é o mesmo sendo necessário utilizar um *MKMapView* atráves de uma *IBOutlet* na classe controladora.

No aplicativo foi feito como no Android, apenas colocando um pino no local do campus, não traçando a rota nem passando maiores informações internamente, para isso foi chamada a API de mapas do Google, nesse caso a mesma foi aberta em um website ao clicar no botão de localizar com o endereço selecionado, passando os dados de partida e destino e deixa a cargo da API de realizar esse procedimento.

4.3.5. MENSAGENS PARA O USUÁRIO

Da mesma forma que o Android, o iOS algumas vezes precisa comunicar o usuário o que ocorre durante a execução principalmente em casos de erro, e para realizar essa tarefa o sistema possuí um componente chamado *Alert*, que gera uma caixa de texto com a mensagem desejada – muito parecido com o que acontece em páginas web.

Nesse caso é necessário que o usuário confirme a visualização, pois diferente do *Toast* o *Alert* não desaparece sozinho, o que pode ser um incomodo mesmo com o mesmo ocupando pouco espaço na tela.

4.3.6. PUBLICAÇÃO

Após todo o desenvolvimento também existe a possibilidade de se publicar o APP na *App Store* da Apple, porém para realizar a publicação é necessário possuir uma conta de desenvolvedor ativa, sendo o maior problema o preço: uma conta da Apple custa 99 dólares anuais por tipo de dispositivo.

Uma característica que deve ser levada em conta na publicação de um APP para iOS é a verificação realizada pela Apple em todos os aplicativos que se candidatam a entrar em sua loja, esse controle de qualidade é uma vantagem que a Apple tem em relação a Google pois a mesma não realiza a mesma verificação que a Apple faz, porém essa verificação pode levar uma semana ou mais e pode ser interpretada como um problema – enquanto na realidade não é.

Por esse motivo foi dito nas seções anteriores desse desenvolvimento que era necessário incluir os ícones e a tela inicial do aplicativo pois sem esses itens o aplicativo é considerado como invalido ou inapropriado.

4.4. RESULTADOS OBTIDOS

Os dois projetos mostrados na seção 7.2 e 7.3 estão disponíveis no apêndice 3 para consultas mais detalhadas, por esse motivo o código não foi destacado nos desenvolvimentos citados.

A maior parte das diferenças entre os projetos que foram destacadas no decorrer da seção 7.3 foram diferenças na construção dos componentes e na maneira de se realizar alguma parte do desenvolvimento, isso ocorreu por conta da diferença de tecnologias utilizadas que interferiram no processo.

Essas diferenças começaram pela IDE que são apropriadas para cada uma das plataformas, outro fator é que pouco importa qual o sistema operacional o Android Studio pode ser utilizado em qualquer um dos principais sistemas disponíveis do mercado incluindo o próprio macOS.

Já para Xcode foi obrigatório o uso de um dispositivo Apple pois a IDE só funciona no próprio macOS, o que pode ser considerado um problema quando se leva em conta os

custos, já que o preço dos computadores da Apple no Brasil são muito mais elevados que o de um computador convencional.

O fato de as ferramentas serem diferentes também fez com as configurações ficassem organizadas de formas diferenciadas, com o *mainfest* no Android que recebendo as configurações de permissão junto a hierarquia de *activitys*, e o iOS com o *info.plist* apenas com as informações e configurações do APP.

Outro diferença que causou impacto foi o desenvolvimento da parte visual, foi a primeira parte em que a diferença de recursos empregados em cada um dos sistemas ficou bem evidente, principalmente pela proposta apresentada pelo iOS.

Uma a*ctivity* no Android é construída em três lugares distintos com os arquivos de layout no diretório *resources*, a declaração hierárquica no *mainfest* e as classes controladores no diretório *sources*, enquanto no iOS o funcionamento é dividido em dois lugares com todas as *view controllers* estando na s*toryboard* e as classes controladoras estando no grupo *views*.

Essas características interferiram diretamente no relacionamento dos componentes visuais com as classes responsáveis pelo comportamento do layout, pois enquanto o iOS o fez pela *storyboard* o Android utilizou os protocolos da classe R.

Uma desvantagem notada ao usar a classe R foi a necessidade de atribuir um identificador aos componentes, nesse caso o id exige uma organização de nomes para que os mesmos não fiquem difíceis de serem encontrados ao ligar o componente ao código,

O mesmo não ocorreu no iOS pois ao criar a *IBOutlet* e *IBAction* pois o nome é dado na hora que o vínculo é feito, porem o mesmo problema que ocorre nos Ids do Android pode ocorrer no iOS em caso de nomes mal escolhidos para as variáveis citadas.

O relacionamento entre as telas segue o mesmo caminho, mais uma vez no Android tudo foi feito via programação sendo necessário programar a *intent*, o retorno da *intent* e o resultado que essa intente gerou – quando houve necessidade – sendo preciso manter o controle de fluxo de forma externa a IDE.

No iOS o uso da *storyboard* facilitou nesse aspecto, pois todos os relacionamento ficaram visíveis como em um diagrama, a única vez que foi necessário fazer um relacionamento via programação foi em *perform segues* customizadas em que houve troca de informações entre as mesmas.

As linguagens de programação utilizadas também foram outro fator de diferença critico, o Swift e o Java são muito diferentes em questões de sintaxe, semântica e de

funções utilizadas, mudando muito a forma de se desenvolver e construir um código, sendo necessárias abordagens diferenciadas para cada uma.

Junto as diferenças de IDE e linguagem de programação, houve também diferenças no uso do banco de dados mesmo com ambos utilizando o *SQLite* como banco nativo, isso ocorreu por conta do uso do core data no iOS.

No Android apesar de existir um *adapter* parar gerenciar o banco - chamado Cursor - é necessário programar esse acesso e criar todo o banco no Android atráves de um script, porém o mesmo não ocorre no iOS com o core data pois ele cria toda as tabelas sozinho e se comporta como entidades, sendo necessário apenas chama-lo no acesso.

Essas diferenças e outras citadas na seção 7.3 podem ser melhores visualizadas conforme mostra na tabela 3.

Tabela 3: Diferenças entre Android e iOS

Diferença em	Android	iOS
IDE	Utiliza o Android Studio em qualquer SO.	Utiliza o Xcode apenas no macOS.
Diretórios	Separados por diretórios diferentes, sendo os principais o src, res e assets.	Separados por diretórios apenas as imagens dos códigos fonte, exceto ao criar um grupo com diretório.
Banco de Dados	SQLite nativo, usa o banco a partir de um script.	Utiliza core data que trata as tabelas como entidades.
Telas	Acitivity, Fragment e TabBar com fragment inflada dentro de uma activity.	Tab Bar Controller, Navigation Controller, View Controller e Table View Controller.
Distribuição da View	Diretório res (e todos os subdiretórios), mainfest.xml e pacote views.	LaunchScreen.storyboard, Main.Storyboard, Assets.xcassets e grupo views.
Comunicação entre View e código	Id capturado pela classe R e componentes com variável de classe com id capturado pela classe R. Método onCreate obrigatório.	Classe selecionada via StoryBoard, componentes via Outlets e Actions. Método viewDidLoad opcional.
Layout	RelativeLayout e GridLayout.	Position View com autoresizing.
Relacionamento entre Telas	Intent, intent result e parent activity.	Segue e PerformSegue.
Mapas	Necessita de uma chave gerada.	Apenas a permissão para o aplicativo.
Publicação	Conta de desenvolvedor vitalícia ao custo de 25 dólares para qualquer dispositivo. APP não demora a ficar disponível na loja.	Conta de desenvolvedor anual ao custo de 100 dólares para um tipo de dispositivo. APP demora a ficar disponível na loja.

Fonte: autor (2018).

Apesar das diferenças citadas o desenvolvimento nativo das duas plataformas apresentou semelhanças, muitas delas foram semelhanças conceituais aonde ambos os dois desenvolvimentos possuem ideias parecidas.

Uma dessas semelhanças foi a utilização do mesmo padrão de projeto, o uso do MVC ajudou a diminuir as diferenças de organização no desenvolvimento dos dois sistemas, uma vez que a arvore de diretórios do Android é organizada de maneira distinta a do iOS.

O MVC também possibilitou organizar melhor o desenvolvimento, isso deixou o trabalho mais fácil de ser feito e de ser comparado, uma vez que a organização da parte visual é completamente diferente adotar o mesmo padrão

Outra semelhança notada foi o ciclo de vida de uma *activity* e de uma *view controller*, apesar de o ciclo da *activity* refletir diretamente ao ciclo de vida do aplicativo como um todo e o da *view controller* ser um ciclo de vida interno, os dois possuem funções que são parecidas entre si que podem facilitar no uso de alguma instrução.

Essa semelhança está na figura 39, aonde os dois ciclos de vida são postos lado a lado e os métodos que fazem funções similares são circulados pelas cores azul, verde, amarela e vermelha. Nota-se que mesmo com o nome diferente, as funções são iguais.

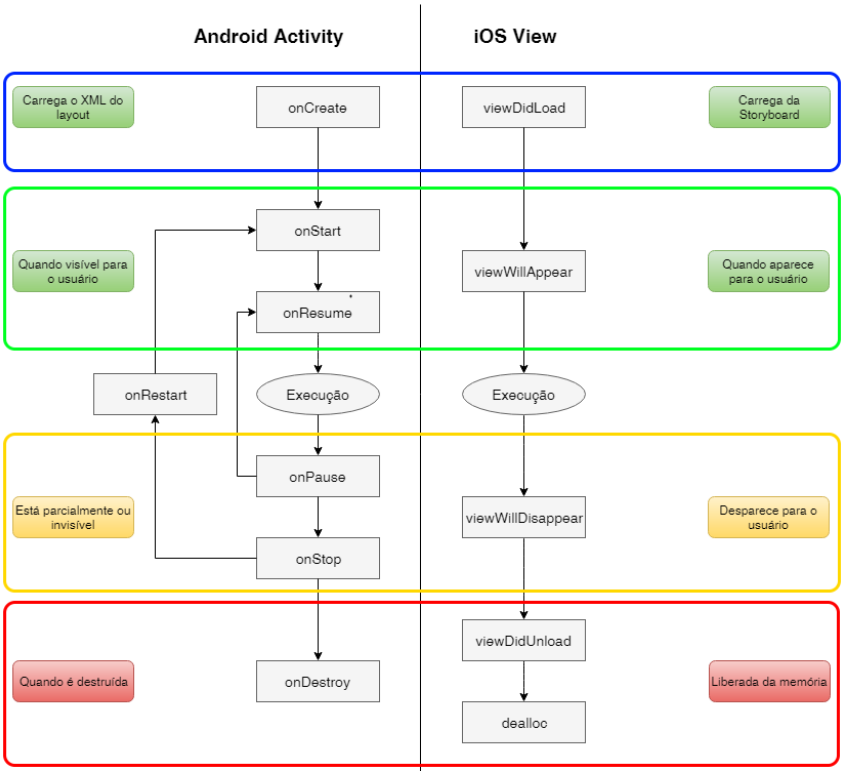


Figura 39: Semelhanças de ciclo de vida.

Fonte: http://vardhan-justlikethat.blogspot.com/ - adaptado pelo autor (2018)

Outra semelhança foi o uso de *web services*, ambos utilizaram o mesmo serviço disponibilizado para alimentar as informações do aplicativo sem necessidade de alterar o endereço utilizado, com os mesmos parâmetros passados e retornos no mesmo formato. A forma de uso também foi parecida, ambos os sistemas não permitem abrir uma conexão na thread principal, sendo diferente a forma de construção e não o conceito utilizado.

O mesmo serviu para os componentes visuais, tanto Android quanto iOS possuem diferentes componentes em suas listas, mas essas diferenças não são um problema pois existem componentes que executam funções similares, sendo necessário conhecer ao menos os componentes mais elementares de cada um como mostra a tabela 4.

Tabela 4: Semelhanças de componentes entre Android e iOS

Semelhança em	Android	iOS
Padrão de projeto	MVC separado por pacotes, havendo a necessidade de importar uma classe de um pacote externo.	MVC separado por grupos, não é necessário importar uma classe ao se utilizar fora do grupo.
Views	Utiliza XML e permite a personalização de componentes.	Utiliza storyboard e permite a personalização de componentes.
Metodos de componentes visuais	Adapter como controlador e ArrayList como fonte de dados	Delegates como controladores e Datasource como fonte de dados.
Tela: texto	TextView	UILabel
Tela: listagem	List View com célula personalizada	List View Controller com table view cell.
Tela: caixa de seleção	Spinner	PickerView
Tela: ações	Button e Floatting Action Button.	UIButton e UITabbarbutton.
Tela: web	Webview com webview cliente.	UIWebview com delegate
Tela: mapa	Fragment inflada exibindo uma mapview, direcionando para a biblioteca de mapas do google traçar a rota	MKMapView, direcionando para o site do google maps para traçar uma rota.
Preferencias	Classe estática SharedPreferences.	Classe estática StoreManager.
Web Service	Executa fora da thread principal via AsyncTask utilizando biblioteca GSon.	Executa fora da thread principal via DataTask;
Testes	Android Simulator.	iOS Simulator.

Fonte: autor (2018).

Como definido na metodologia, foi realizada a tentativa de conversão de códigos de Android para iOS, convertendo o Java para Swift utilizando o Objective-C como linguagem intermediaria por meio das ferramentas J2ObjC e Swifty, essa conversão foi feita após o desenvolvimento no Android ficar pronto.

No primeiro passo – de Java para Objective-C – as tentativas de conversão foram realizadas por pacotes separados para uma melhor observação dos resultados, pois o conversor gera um arquivo com a extensão h e outro com a extensão m, com a quantidade duplicada de arquivos a organização poderia ficar comprometida.

Os resultados ficaram abaixo do esperado, o conversor converteu apenas os arquivos que não tinham bibliotecas especificas do Android, ou seja, os arquivos dos pacotes *view*, *asynctask* e *controller* não puderam ser convertidos como mostra a figura 40.

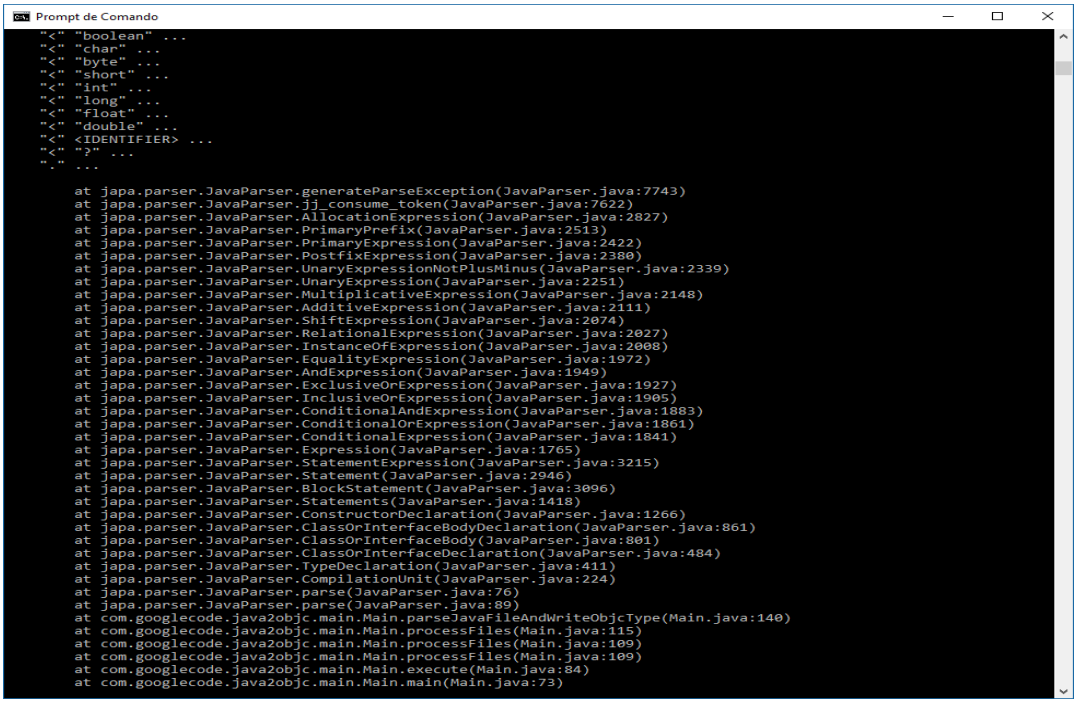


Figura 40: Conversão de Java para Objective-C.

Fonte: Autor (2018)

Os modelos foram os arquivos que tiveram a melhor conversão no primeiro passo, com as classes de *services* – excluindo o *asynctask* - com a segunda melhor taxa de aproveitamento. Porém, ao tentar converter para Swift a ferramenta apresentou outra série de limitações, como a necessidade de uma assinatura para permitir a conversão de um projeto completo e os arquivos serem restritos a 1kB a versão *free*.

Como mostra a figura 41 o código convertido apresentou algumas falhas estruturais e trechos desnecessários, com a necessidade constante de correções a utilização desses códigos ficou inviável e os mesmos foram descartados.

Figura 41: Conversão de Objective-C para Swift.

Fonte: Autor (2018)

Mesmo com a falha na conversão foi possível seguir todas as especificações sem realizar nenhuma adaptação, com o mesmo escopo de projeto sendo seguido e mantendo os dois projetos indo pelo mesmo caminho de desenvolvimento.

Outro ponto foi o banco de dados, apesar da diferença de implementação o mesmo Schema foi seguido sem causar nenhuma mudança na integridade do dados, sendo necessário apenas ter atenção em relação a forma como cada sistema os utilizou.

As telas seguem o mesmo exemplo, mesmo que não tenham sido reaproveitadas elas seguiram o mesmo protótipo tanto no iOS quanto no Android. Juntamente com o Schema de banco de dados, o uso do mesmo protótipo de telas demonstra que utilizar ferramentas de modelagem e de apoio são uma excelente alternativa na falta de conversores.

Por último o reaproveitamento de *web service* foi essencial para que os aplicativos pudessem chegar ao mesmo objetivo, os endereços utilizados foram os mesmos e os parâmetros passados seguiram exatamente o mesmo padrão – get para o de notícias e post para os restantes.

Esse é o principal recurso compatível entre as plataformas, sendo o ponto de interoperabilidade entre os sistemas. A possibilidade de reaproveitamento e compatibilidade ficam melhor compreendidos com a tabela 5 com todos esses aspectos resumidos.

Tabela 5: Reaproveitamento e Compatibilidade de recursos entre Android e iOS.

Recurso	Reaproveitamento/Compatibilidade
Conversão de código	Não. O uso é Ineficiente e o tempo gasto é superior ao de fazer do início.
Requisitos e Especificações	Sim. Podem ser seguidos sem necessidade de adaptações.
Banco de dados	Sim. Apenas o mesmo Schema pode ser utilizado.
Telas	Sim. O mesmo protótipo pode ser seguido, não sendo possível a conversão entre as telas dos sistemas.
Web service	Sim. O mesmo serviço pode ser utilizado sem nenhuma limitação via get/post.

Fonte: autor (2018).

5. CONCLUSÕES

Após os estudos realizados neste trabalho ficou claro que a escolha do desenvolvimento nativo nas plataformas Android e iOS deve levar em conta que o mesmo projeto deverá sempre ser executado duas vezes, pois é necessário compilar o código separadamente para cada plataforma ao qual o aplicativo se destina.

O desenvolvedor deverá entender como cada desenvolvimento funciona e isso inclui conhecer a linguagem de programação, saber como cada um lida com as *views* principalmente na parte de relacionamento entre telas e de componentes, bem como o uso de banco de dados.

Tais diferenças trazem dificuldades para desenvolver ao mesmo tempo para as duas plataformas, não apenas pelos recursos tecnológicos e ferramentas de desenvolvimento utilizada separadamente em cada um, mas também pelo baixo reaproveitamento de recursos entre essas plataformas.

Usar a conversão de códigos se mostrou um método ineficiente com as ferramentas utilizadas, outro ponto é que não foi possível converter as *views* por estas serem estruturadas de maneiras distintas, levando isso em conta é a melhor forma é fazer tudo do zero a partir da mesma especificação e o aplicativo desenvolvido neste trabalho reforça esse ponto.

Essas dificuldades podem ser amenizadas ao se utilizar os mesmos desenhos de telas e seguir a mesma especificação como exemplificado durante todo esse trabalho, além disso ao adotar um padrão de projeto como o MVC essa diferença de arquitetura diminuiu como foi mostrado nas seções 7.2 e 7.3, os projetos se tornaram idênticos facilitando este comparativo, inclusive permitindo detectar semelhanças conceituais e de componentes que fazem funções similares entre as plataformas.

Outro facilitador entre os projetos foi a adoção dos *web services*, mesmo nos dispositivos móveis eles ainda são um dos principais recursos que permitem a interoperabilidade entre sistemas e sem eles o aplicativo criado para o comparativo não teria nenhuma utilidade.

Com o tempo a nova linguagem oficial do Android deve ganhar mais espaço em relação ao Java, cada vez mais os desenvolvedores verão o Kotlin como opção principal e não mais como uma linguagem nova recém chegada, uma vez que o que foi visto no iOS entre Swift e Objective-C acabará acontecendo também no Android.

O mesmo comparativo poderá ser feito utilizando o Kotlin e as novas atualizações do Android Studio como base, em vista que a tendência da Google de aproximar as características de desenvolvimento do Android as características do iOS fica cada vez mais evidente.

A longo prazo, com o intuito de melhorar a aproximação entre as duas plataformas e visando diminuir as dificuldades encontradas, novas ferramenta de apoio de modelagem ou de conversão podem ser pesquisada, uma vez que as diferenças entre os dois projetos tende a se tornar menor com o passar dos anos.

Dessa forma é possível concluir que os projetos aqui desenvolvidos são incompatíveis no aspecto tecnológico trazendo grandes dificuldades para aqueles que pretendem utilizar o desenvolvimento nativo, mas possuem compatibilidade quanto aos conceitos utilizados podendo inclusive possuir um padrão arquitetural igual.

REFERÊNCIAS

Android Developers, 2018. **Site Oficial**. Disponível em: < *https://developer.android.com* >. Acesso em: 3 mar. 2018.

Android Source, 2018. **Site Oficial**. Disponível em: < *https://source.android.com* >. Acesso em: 3 mar. 2018.

Apple Developer, 2018. **Site Oficial**. Disponível em: < *https://developer.apple.com* >. Acesso em: 4 mar. 2017.

Bezerra. P. T., Schimiguel, J., 2015. **Desenvolvimento de aplicações mobile cross-platform utilizando Phonegap.** Disponível em: < *http://www.eumed.net/cursecon/ecolat/br/16/phonegap.html* >. Acesso em: 3 mar. 2018.

Cryah, 2016. **Crescimento do mercado Mobile no Brasil e no mundo**. Disponível em: < *https://cryah.com.br/crescimento-do-mercado-mobile-no-brasil-e-no-mundo* >. Acesso em: 10 fev. 2018

Da Fonseca. M. R., Beder, D. M., 2015. **Aplicativos Android: desenvolvimento nativo x uso de ferramentas baseadas em padrões web.** Disponível em: < *http://revistatis.dc.ufscar.br/index.php/revista/article/view/302/102* >. Acesso em: 1 mar. 2018.

Da Silva. L. L .B, Pires, D. F., Neto. S. C., 2015. **Desenvolvimento de Aplicações para Dispositivos Móveis: Tipos e Exemplo de Aplicação na plataforma iOS**. Disponível em: < *http://www.lbd.dcc.ufmg.br/colecoes/wicsi/2015/004.pdf* >. Acesso em: 1 mar. 2018.

DevMedia, 2012. **Introdução ao Padrão MVC**. Disponível em: < *http://www.devmedia.com.br/introducao-ao-padrao-mvc/29308* >. Acesso em: 6 mar. 2018.

Gatner, 2018. **Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017**. Disponível em: < *https://www.gartner.com/newsroom/id/3859963* >. Acesso em: 15 mar. 2018.

J2ObjC, 2018. **Site Oficial**. Disponível em: < *https://developers.google.com/j2objc* >. Acesso em: 20 mar. 2018.

LECHETA, Ricardo R., 2016. **Google Android – Aprenda a criar aplicações para dispositivos móveis com o Android SDK**, São Paulo: Editora Novatec, 2016. 1068 p. ISBN 978-85-7522-468-7.

Open Handset Alliance, 2018. **Site Oficial.** Disponível em: < *https://www.openhandsetalliance.com* >. Acesso em: 3 mar. 2018.

Rocha. A. M., Neto, R.M.F., 2011. **Introdução à Arquitetura Apple iOS**. Disponível em: < *http://studylibpt.com/doc/896343/introdu%C3%A7-%CB%9Cao--a-arquitetura-apple-ios* >. Acessado em: 5 mar. 2018.

SOAWebService, 2012. **Como funciona.** Disponível em:< *http://www.soawebservices.com.br/como-funciona.aspx* >. Acesso em: 8 mar. 2018

SQLite, 2018. **Site Oficial.** Disponível em:< *https://www.sqlite.org* >. Acesso em: 5 mar. 2018.

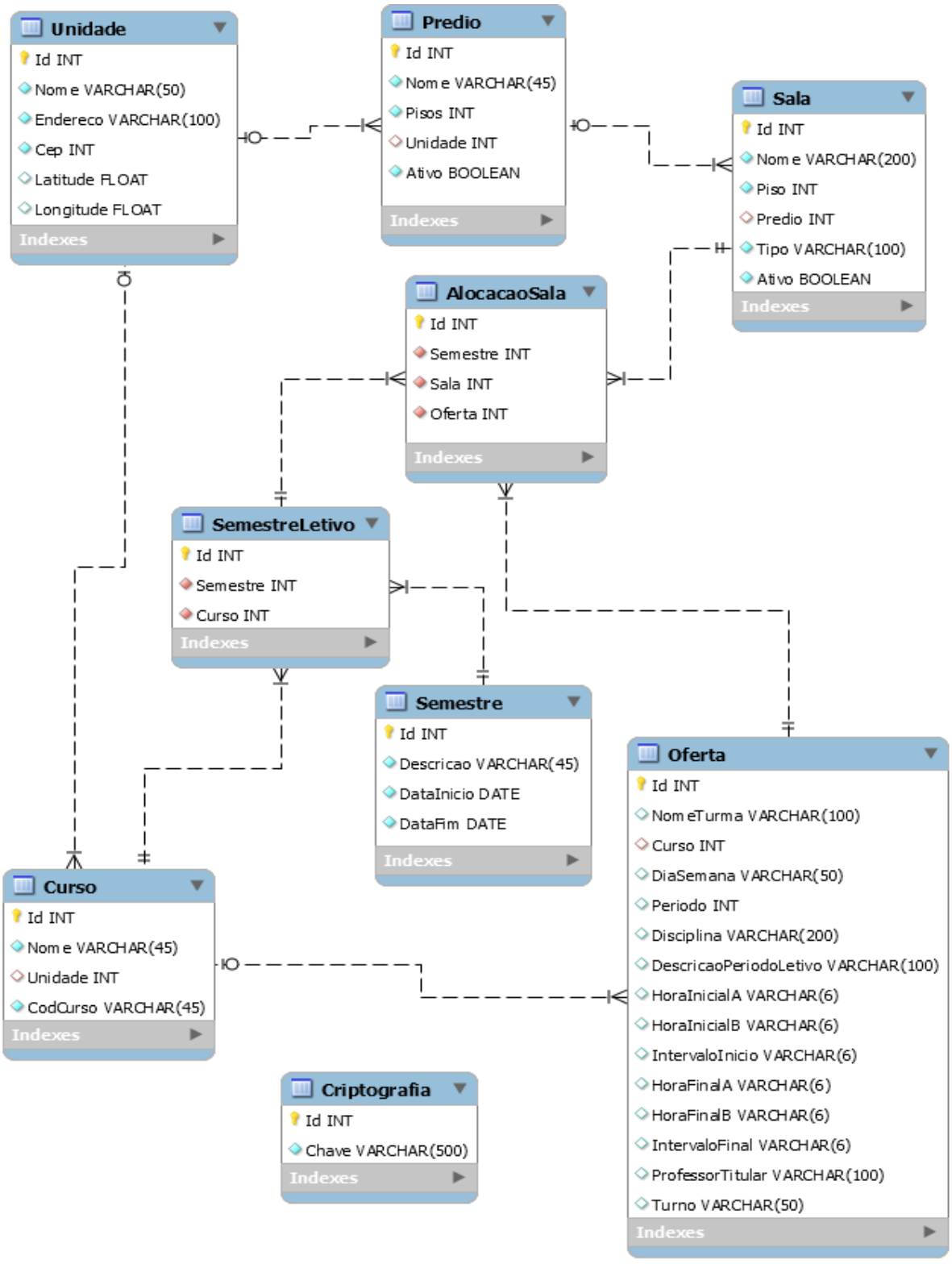
SwiftiFy, 2018. **Site Oficial.** Disponível em:<*https://objectivec2swift.com* >. Acesso em: 20 mar. 2018.

UNIFEIJr, 2017. **6 motivos para investir no mercado de aplicativos.** Disponível em: < *http://unifeijr.com.br/blog/mercado-de-aplicativos* >. Acesso em: 18 fev. 2018.

VENTURA, Plinio., 2016. **Requisitos Funcionais, Casos de Uso em Engenharia de Software.** Disponível em:< *http://www.ateomomento.com.br* >. Acesso em: 7 mar. 2018.

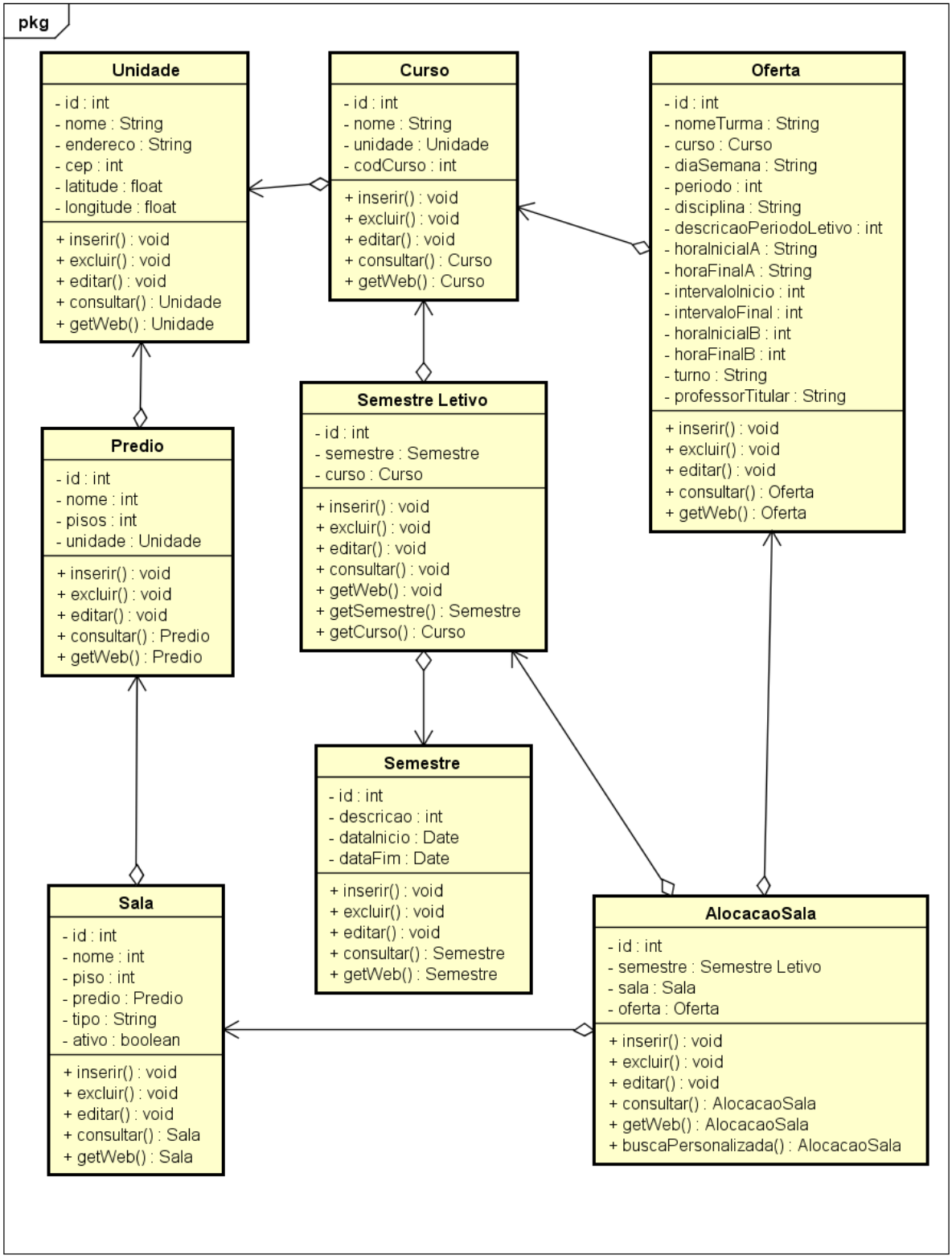
APÊNDICE I

Schema do Banco de Dados



APÊNDICE II

Diagrama de Classes



APÊNDICE III

Códigos fonte dos aplicativos.

Android: disponível em https://github.com/saviossmg/Projeto_TCC_Android
iOS: disponível em https://github.com/saviossmg/Projeto_TCC_IOS